Arm® A32/T32 Instruction Set Architecture

Armv8, for Armv8-A architecture profile

arm

Copyright © 2010-2021 Arm Limited (or its affiliates). All rights reserved.
DDI 0597 (ID061021)

Arm A32/T32 Instruction Set Architecture
Armv8, for Armv8-A architecture profile

Copyright © 2010-2021 Arm Limited (or its affiliates). All rights reserved.

Release Information

For information on the change history and known issues for this release, see the **Release Notes** in the **A32/T32 ISA XML for Armv8.7 (2021-06)**.

Proprietary Notice

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

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

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

This document may include technical inaccuracies or typographical errors.

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

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

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

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

Copyright © 2010-2021 Arm Limited (or its affiliates). All rights reserved.

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

(LES-PRE-20349 version 21.0)

Confidentiality Status

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

Product Status

The information in this document is at Beta quality. Beta quality means that all major features of the specification are included, some details might be missing.

Web Address

`http://www.arm.com`

Progressive Terminology Commitment

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

Previous issues of this document included terms that can be offensive. We have replaced these terms. If you find offensive terms in this document, please contact `terms@arm.com`.

Copyright © 2010-2021 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Copyright © 2010-2021 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

AArch32 -- Base Instructions (alphabetic order)

ADC, ADCS (immediate): Add with Carry (immediate).

ADC, ADCS (register): Add with Carry (register).

ADC, ADCS (register-shifted register): Add with Carry (register-shifted register).

ADD (immediate, to PC): Add to PC: an alias of ADR.

ADD, ADDS (immediate): Add (immediate).

ADD, ADDS (register): Add (register).

ADD, ADDS (register-shifted register): Add (register-shifted register).

ADD, ADDS (SP plus immediate): Add to SP (immediate).

ADD, ADDS (SP plus register): Add to SP (register).

ADR: Form PC-relative address.

AND, ANDS (immediate): Bitwise AND (immediate).

AND, ANDS (register): Bitwise AND (register).

AND, ANDS (register-shifted register): Bitwise AND (register-shifted register).

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

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

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

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

B: Branch.

BFC: Bit Field Clear.

BFI: Bit Field Insert.

BIC, BICS (immediate): Bitwise Bit Clear (immediate).

BIC, BICS (register): Bitwise Bit Clear (register).

BIC, BICS (register-shifted register): Bitwise Bit Clear (register-shifted register).

BKPT: Breakpoint.

BL, BLX (immediate): Branch with Link and optional Exchange (immediate).

BLX (register): Branch with Link and Exchange (register).

BX: Branch and Exchange.

BXJ: Branch and Exchange, previously Branch and Exchange Jazelle.

CBNZ, CBZ: Compare and Branch on Nonzero or Zero.

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_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ADC, ADCS (immediate)

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

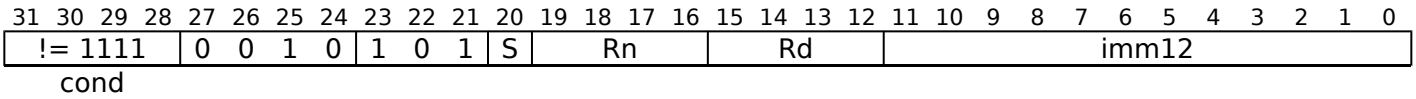
If the destination register is not the PC, the ADCS variant of the instruction updates the condition flags based on the result.

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

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

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

A1



ADC (S == 0)

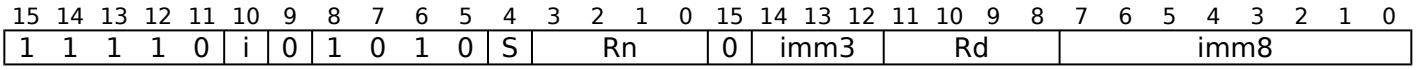
```
ADC{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

ADCS (S == 1)

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

```
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = A32ExpandImm(imm12);
```

T1



ADC (S == 0)

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

ADCS (S == 1)

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

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

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

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<Rd> For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. Arm deprecates using the PC as the destination register, but if the PC is used:

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

For encoding T1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.

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

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

<const> For encoding A1: an immediate value. See *Modified immediate constants in A32 instructions* for the range of values.

For encoding T1: an immediate value. See *Modified immediate constants in T32 instructions* for the range of values.

Operation

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

Operational information

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

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

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ADC, ADCS (register)

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

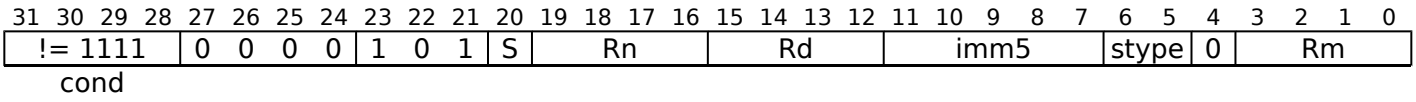
If the destination register is not the PC, the ADCS variant of the instruction updates the condition flags based on the result.

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

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

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

A1



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

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

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

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

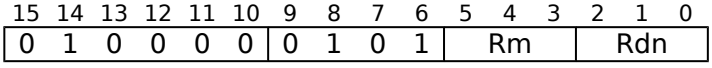
ADCS, rotate right with extend (S == 1 && imm5 == 00000 && stype == 11)

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

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

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

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

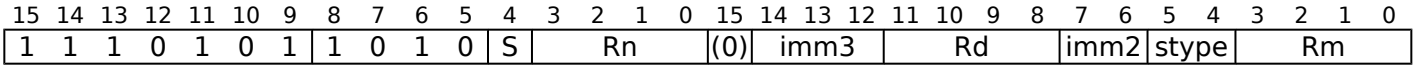
T1



T1

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

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

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

T2



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

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

ADC, shift or rotate by value (S == 0 && !(imm3 == 000 && imm2 == 00 && stype == 11))

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

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

ADCS, rotate right with extend (S == 1 && imm3 == 000 && imm2 == 00 && stype == 11)

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

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

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

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

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

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

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

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

<Rd> For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. Arm deprecates using the PC as the destination register, but if the PC is used:

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

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

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

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

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

For encoding T1 and T2: is the second general-purpose source register, encoded in the "Rm" field.

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

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

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

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

In T32 assembly:

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

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

Operation

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

Operational information

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

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

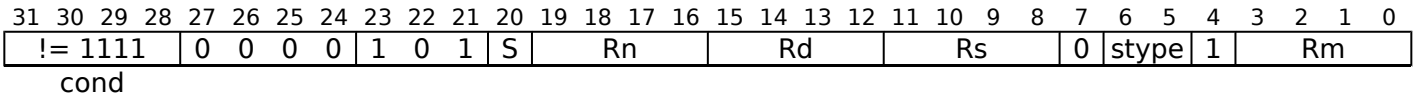
Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ADC, ADCS (register-shifted register)

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

A1



Flag setting (S == 1)

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

Not flag setting (S == 0)

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

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

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

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<Rd> Is the general-purpose destination register, encoded in the "Rd" field.

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

<Rm> Is the second general-purpose source register, encoded in the "Rm" field.

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

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

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

Operation

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

Operational information

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

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

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ADD (immediate, to PC)

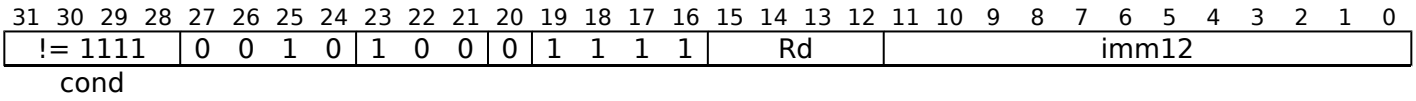
Add to PC adds an immediate value to the Align(PC, 4) value to form a PC-relative address, and writes the result to the destination register. Arm recommends that, where possible, software avoids using this alias

This is a pseudo-instruction of ADR. This means:

- The encodings in this description are named to match the encodings of ADR.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of ADR gives the operational pseudocode for this instruction.

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

A1



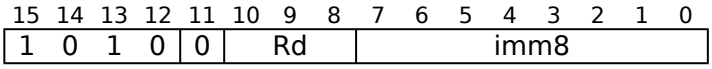
A1

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

is equivalent to

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

T1



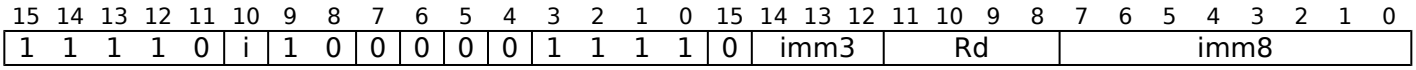
T1

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

is equivalent to

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

T3



T3

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

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

is equivalent to

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

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<Rd> For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If the PC is used, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*.

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

<label> For encoding A1: the label of an instruction or literal data item whose address is to be loaded into <Rd>. The assembler calculates the required value of the offset from the Align(PC, 4) value of the `ADR` instruction to this label.

If the offset is zero or positive, encoding A1 is used, with imm32 equal to the offset.

If the offset is negative, encoding A2 is used, with imm32 equal to the size of the offset. That is, the use of encoding A2 indicates that the required offset is minus the value of imm32.

Permitted values of the size of the offset are any of the constants described in *Modified immediate constants in A32 instructions*.

For encoding T1: the label of an instruction or literal data item whose address is to be loaded into <Rd>. The assembler calculates the required value of the offset from the Align(PC, 4) value of the `ADR` instruction to this label. Permitted values of the size of the offset are multiples of 4 in the range 0 to 1020.

For encoding T3: the label of an instruction or literal data item whose address is to be loaded into <Rd>. The assembler calculates the required value of the offset from the Align(PC, 4) value of the `ADR` instruction to this label.

If the offset is zero or positive, encoding T3 is used, with imm32 equal to the offset.

If the offset is negative, encoding T2 is used, with imm32 equal to the size of the offset. That is, the use of encoding T2 indicates that the required offset is minus the value of imm32.

Permitted values of the size of the offset are 0-4095.

<imm8> Is an unsigned immediate, a multiple of 4, in the range 0 to 1020, encoded in the "imm8" field as <imm8>/4.

<imm12> Is a 12-bit unsigned immediate, in the range 0 to 4095, encoded in the "i:imm3:imm8" field.

<const> An immediate value. See *Modified immediate constants in A32 instructions* for the range of values.

Operation

The description of ADR gives the operational pseudocode for this instruction.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ADD, ADDS (immediate)

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

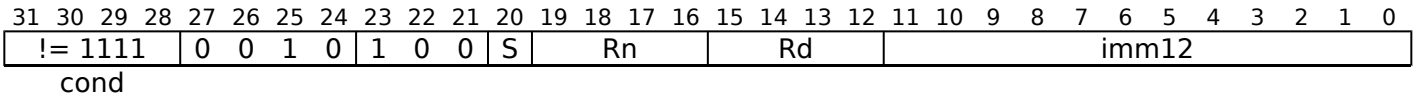
If the destination register is not the PC, the ADDS variant of the instruction updates the condition flags based on the result.

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

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

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

A1



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

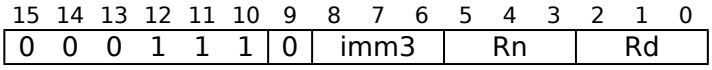
```
ADD{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

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

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

```
if Rn == '1111' && S == '0' then SEE "ADR";
if Rn == '1101' then SEE "ADD (SP plus immediate)";
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = A32ExpandImm(imm12);
```

T1

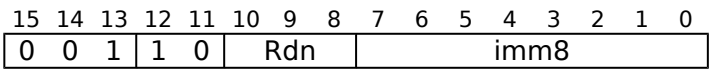


T1

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

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

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

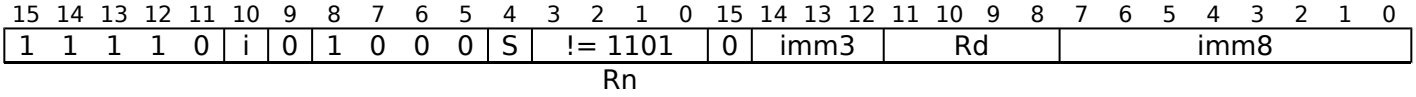
T2



T2

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

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

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

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

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

T3



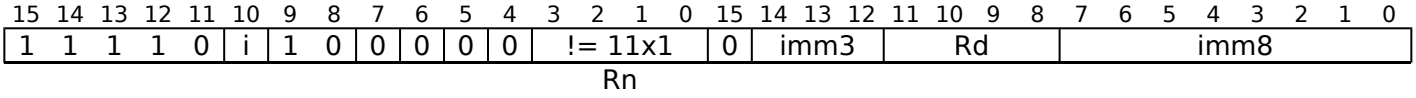
ADD (S == 0)

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

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

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

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

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

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

T4



T4

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

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

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

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

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rdn>	Is the general-purpose source and destination register, encoded in the "Rdn" field.
<imm8>	Is a 8-bit unsigned immediate, in the range 0 to 255, encoded in the "imm8" field.

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

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

For encoding T1, T3 and T4: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.

<Rn> For encoding A1 and T4: is the general-purpose source register, encoded in the "Rn" field. If the SP is used, see *ADD (SP plus immediate)*. If the PC is used, see *ADR*.

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

For encoding T3: is the general-purpose source register, encoded in the "Rn" field. If the SP is used, see *ADD (SP plus immediate)*.

<imm3> Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "imm3" field.

<imm12> Is a 12-bit unsigned immediate, in the range 0 to 4095, encoded in the "i:imm3:imm8" field.

<const> For encoding A1: an immediate value. See *Modified immediate constants in A32 instructions* for the range of values.

For encoding T3: an immediate value. See *Modified immediate constants in T32 instructions* for the range of values.

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

Operation

```
if CurrentInstrSet() == InstrSet_A32 then
    if ConditionPassed() then
        EncodingSpecificOperations();
        (result, nzcv) = AddWithCarry(R[n], imm32, '0');
        if d == 15 then          // Can only occur for A32 encoding
            if setflags then
                ALUExceptionReturn(result);
            else
                ALUWritePC(result);
        else
            R[d] = result;
            if setflags then
                PSTATE.<N,Z,C,V> = nzcv;
else
    if ConditionPassed() then
        EncodingSpecificOperations();
        (result, nzcv) = AddWithCarry(R[n], imm32, '0');
        R[d] = result;
        if setflags then
            PSTATE.<N,Z,C,V> = nzcv;
```

Operational information

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

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

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ADD, ADDS (register)

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

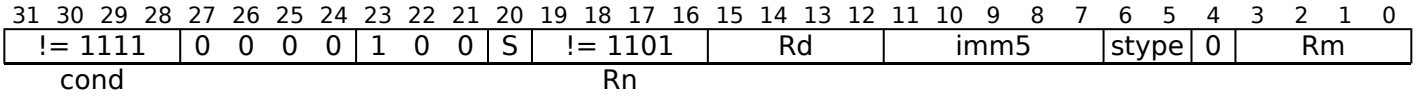
If the destination register is not the PC, the ADDS variant of the instruction updates the condition flags based on the result.

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

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

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

A1



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

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

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

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

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

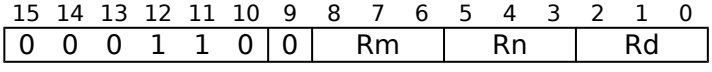
```
ADDS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

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

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

```
if Rn == '1101' then SEE "ADD (SP plus register)";
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm5);
```

T1

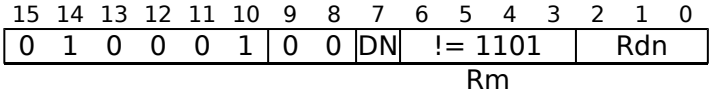


T1

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

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

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);
```

T2

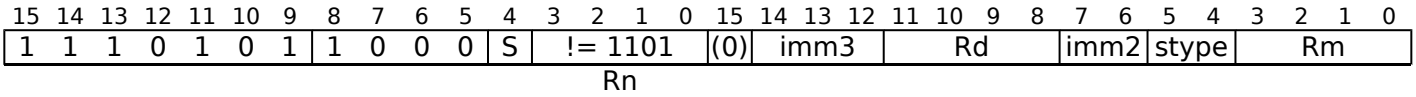


T2 (!(DN == 1 && Rdn == 101))

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

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

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

T3



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

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

ADD, shift or rotate by value (S == 0 && !(imm3 == 000 && imm2 == 00 && stype == 11))

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

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

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

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

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

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

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

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

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

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

Assembler Symbols

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

<Rdn> Is the general-purpose source and destination register, encoded in the "DN:Rdn" field. If the PC is used, the instruction is a branch to the address calculated by the operation. This is a simple branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*.

The assembler language allows <Rdn> to be specified once or twice in the assembler syntax. When used inside an IT block, and <Rdn> and <Rm> are in the range R0 to R7, <Rdn> must be specified once so that encoding T2 is preferred to encoding T1. In all other cases there is no difference in behavior when <Rdn> is specified once or twice.

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

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

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

When used inside an IT block, <Rd> must be specified. When used outside an IT block, <Rd> is optional, and:

- If omitted, this register is the same as <Rn>.
- If present, encoding T1 is preferred to encoding T2.

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

<Rn> For encoding A1: is the first general-purpose source register, encoded in the "Rn" field. The PC can be used. If the SP is used, see *ADD (SP plus register)*.

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

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

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

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

For encoding T2: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used.

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

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

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

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

Inside an IT block, if ADD<c> <Rd>, <Rn>, <Rd> cannot be assembled using encoding T1, it is assembled using encoding T2 as though ADD<c> <Rd>, <Rn> had been written. To prevent this happening, use the .W qualifier.

Operation

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

Operational information

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

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

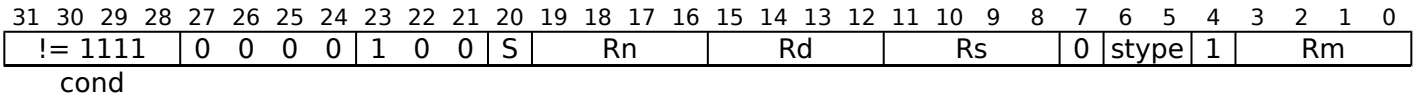
Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ADD, ADDS (register-shifted register)

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

A1



Flag setting (S == 1)

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

Not flag setting (S == 0)

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

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

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

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<Rd> Is the general-purpose destination register, encoded in the "Rd" field.

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

<Rm> Is the second general-purpose source register, encoded in the "Rm" field.

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

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

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

Operation

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

Operational information

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

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

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ADD, ADDS (SP plus immediate)

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

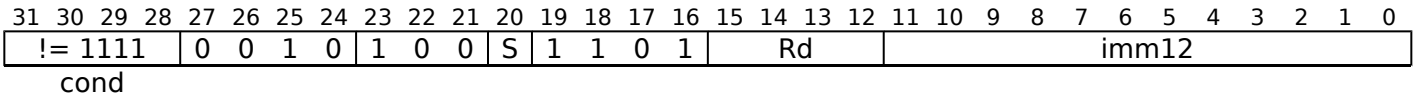
If the destination register is not the PC, the ADDS variant of the instruction updates the condition flags based on the result.

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

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

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

A1



ADD (S == 0)

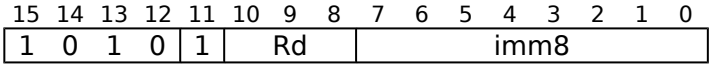
```
ADD{<c>}{<q>} {<Rd>,} SP, #<const>
```

ADDS (S == 1)

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

```
d = UInt(Rd);  setflags = (S == '1');  imm32 = A32ExpandImm(imm12);
```

T1

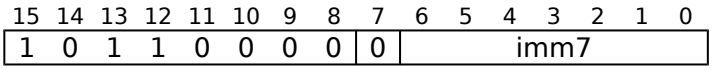


T1

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

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

T2

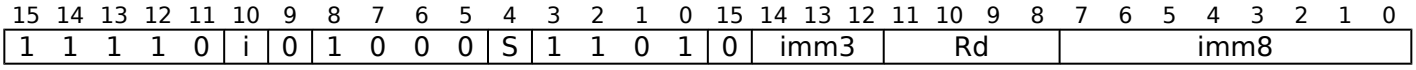


T2

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

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

T3



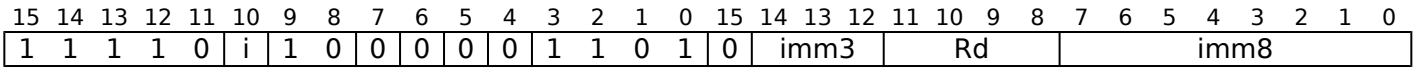
ADD (S == 0)

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

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

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

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

if Rd == '1111' && S == '1' then SEE "CMN (immediate)";
d = UInt(Rd);  setflags = (S == '1');  imm32 = T32ExpandImm(i:imm3:imm8);
if d == 15 && !setflags then UNPREDICTABLE;
```

T4



T4

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

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

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

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

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
SP,	Is the stack pointer.
<imm7>	Is the unsigned immediate, a multiple of 4, in the range 0 to 508, encoded in the "imm7" field as <imm7>/4.
<Rd>	For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the SP. Arm deprecates using the PC as the destination register, but if the PC is used: • For the ADD variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*. • For the ADDS variant, the instruction performs an exception return, that restores *PSTATE* from SPSR_<current_mode>. For encoding T1: is the general-purpose destination register, encoded in the "Rd" field. For encoding T3 and T4: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the SP.
<imm8>	Is an unsigned immediate, a multiple of 4, in the range 0 to 1020, encoded in the "imm8" field as <imm8>/4.
<imm12>	Is a 12-bit unsigned immediate, in the range 0 to 4095, encoded in the "i:imm3:imm8" field.
<const>	For encoding A1: an immediate value. See *Modified immediate constants in A32 instructions* for the range of values. For encoding T3: an immediate value. See *Modified immediate constants in T32 instructions* for the range of values.

Operation

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

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ADD, ADDS (SP plus register)

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

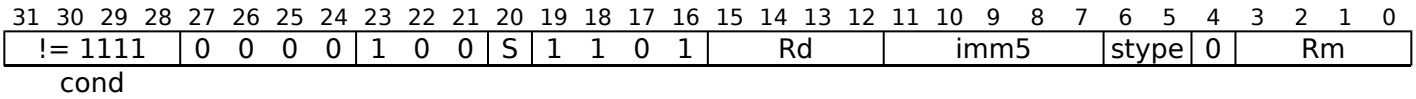
If the destination register is not the PC, the ADDS variant of the instruction updates the condition flags based on the result.

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

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

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

A1



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

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

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

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

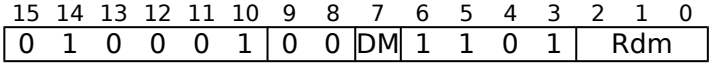
ADDS, rotate right with extend (S == 1 && imm5 == 00000 && stype == 11)

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

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

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

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

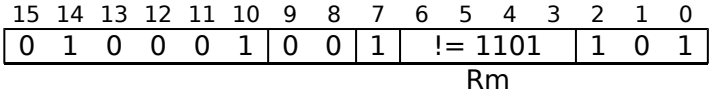
T1



T1

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

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

T2

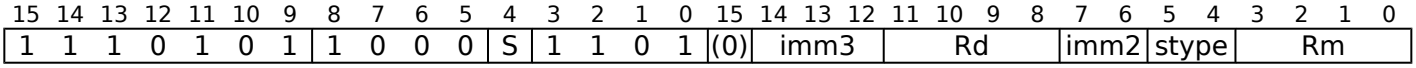


T2

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

```
if Rm == '1101' then SEE "encoding T1";
d = 13;  m = UInt(Rm);  setflags = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
```

T3



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

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

ADD, shift or rotate by value (S == 0 && !(imm3 == 000 && imm2 == 00 && stype == 11))

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

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

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

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

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

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

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

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

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
SP,	Is the stack pointer.
<Rdm>	Is the general-purpose destination and second source register, encoded in the "Rdm" field. If omitted, this register is the SP. Arm deprecates using the PC as the destination register, but if the PC is used, the instruction is a branch to the address calculated by the operation. This is a simple branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*.
<Rd>	For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the SP. Arm deprecates using the PC as the destination register, but if the PC is used: • For the ADD variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*.

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

For encoding T3: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the SP.

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

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

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

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

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

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

Operation

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

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ADR

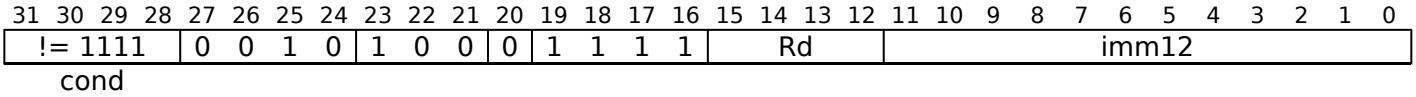
Form PC-relative address adds an immediate value to the PC value to form a PC-relative address, and writes the result to the destination register.

This instruction is used by the alias SUB (immediate, from PC).

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

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

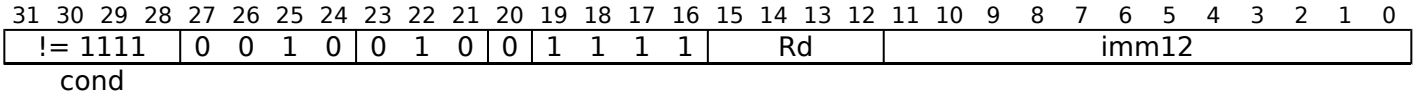
A1



A1

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

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

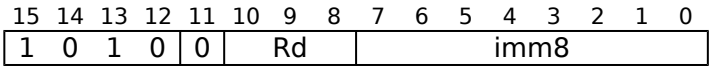
A2



A2

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

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

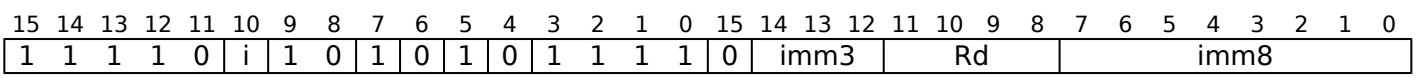
T1



T1

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

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

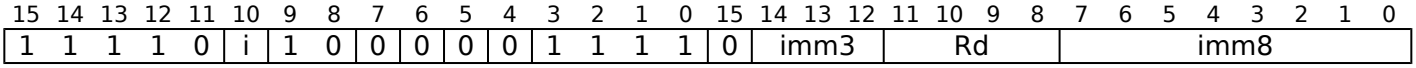
T2



T2

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

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

T3



T3

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

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

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

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

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<Rd> For encoding A1 and A2: is the general-purpose destination register, encoded in the "Rd" field. If the PC is used, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*.

For encoding T1, T2 and T3: is the general-purpose destination register, encoded in the "Rd" field.

<label> For encoding A1 and A2: the label of an instruction or literal data item whose address is to be loaded into <Rd>. The assembler calculates the required value of the offset from the Align(PC, 4) value of the ADR instruction to this label.

If the offset is zero or positive, encoding A1 is used, with imm32 equal to the offset.

If the offset is negative, encoding A2 is used, with imm32 equal to the size of the offset. That is, the use of encoding A2 indicates that the required offset is minus the value of imm32.

Permitted values of the size of the offset are any of the constants described in *Modified immediate constants in A32 instructions*.

For encoding T1: the label of an instruction or literal data item whose address is to be loaded into <Rd>. The assembler calculates the required value of the offset from the Align(PC, 4) value of the ADR instruction to this label. Permitted values of the size of the offset are multiples of 4 in the range 0 to 1020.

For encoding T2 and T3: the label of an instruction or literal data item whose address is to be loaded into <Rd>. The assembler calculates the required value of the offset from the Align(PC, 4) value of the ADR instruction to this label.

If the offset is zero or positive, encoding T3 is used, with imm32 equal to the offset.

If the offset is negative, encoding T2 is used, with imm32 equal to the size of the offset. That is, the use of encoding T2 indicates that the required offset is minus the value of imm32.

Permitted values of the size of the offset are 0-4095.

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

Alias Conditions

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

Operation

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

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

AND, ANDS (immediate)

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

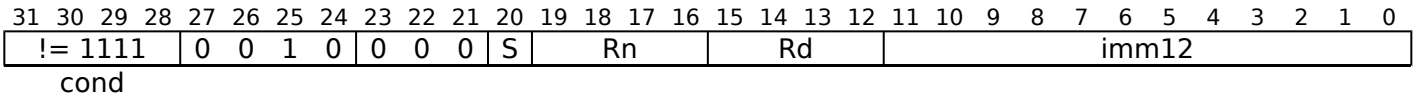
If the destination register is not the PC, the ANDS variant of the instruction updates the condition flags based on the result.

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

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

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

A1



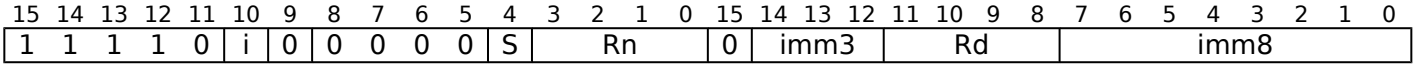
AND (S == 0)

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

ANDS (S == 1)

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

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

T1



AND (S == 0)

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

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

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

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

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

Assembler Symbols

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

Operation

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

Operational information

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

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

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

AND, ANDS (register)

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

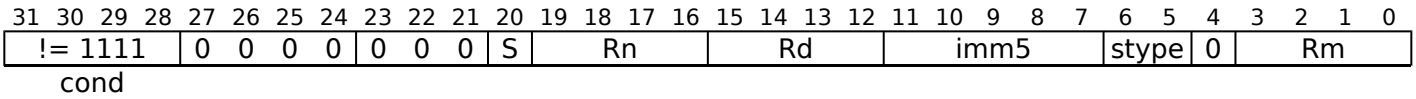
If the destination register is not the PC, the ANDS variant of the instruction updates the condition flags based on the result.

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

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

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

A1



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

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

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

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

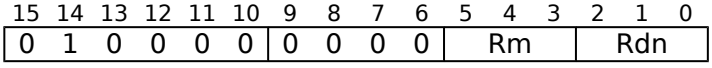
ANDS, rotate right with extend (S == 1 && imm5 == 00000 && stype == 11)

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

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

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

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

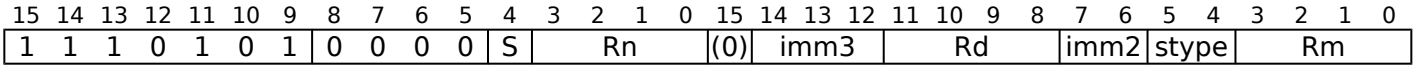
T1



T1

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

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

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

T2



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

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

AND, shift or rotate by value (S == 0 && !(imm3 == 000 && imm2 == 00 && stype == 11))

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

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

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

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

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

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

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

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

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

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rdn>	Is the first general-purpose source register and the destination register, encoded in the "Rdn" field.
<Rd>	For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. Arm deprecates using the PC as the destination register, but if the PC is used: • For the AND variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*. • For the ANDS variant, the instruction performs an exception return, that restores *PSTATE* from SPSR_<current_mode>. For encoding T2: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.
<Rn>	For encoding A1: is the first general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated. For encoding T2: is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	For encoding A1: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated. For encoding T1 and T2: is the second general-purpose source register, encoded in the "Rm" field.
<shift>	Is the type of shift to be applied to the second source register, encoded in “stype”:

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

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

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

In T32 assembly:

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

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

Operation

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

Operational information

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

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

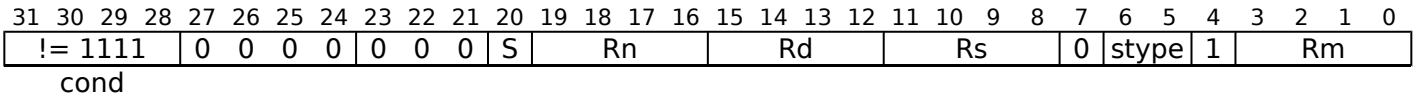
Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

AND, ANDS (register-shifted register)

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

A1



Flag setting (S == 1)

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

Not flag setting (S == 0)

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

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

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

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<Rd> Is the general-purpose destination register, encoded in the "Rd" field.

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

<Rm> Is the second general-purpose source register, encoded in the "Rm" field.

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

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

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

Operation

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

Operational information

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

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

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ASR (immediate)

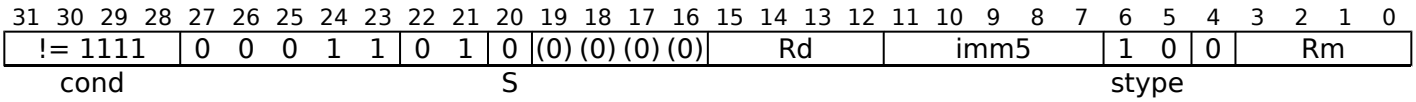
Arithmetic Shift Right (immediate) shifts a register value right by an immediate number of bits, shifting in copies of its sign bit, and writes the result to the destination register.

This is an alias of MOV, MOVS (register). This means:

- The encodings in this description are named to match the encodings of MOV, MOVS (register).
- The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.

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

A1



MOV, shift or rotate by value

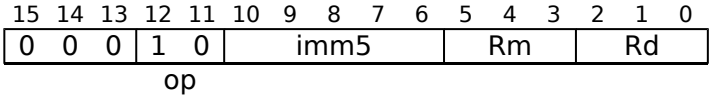
```
ASR{<c>}{<q>} {<Rd>,} <Rm>, #<imm>
```

is equivalent to

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

and is always the preferred disassembly.

T2



T2

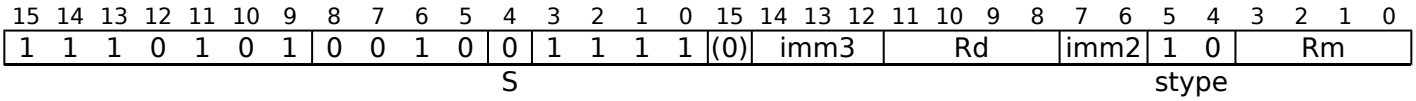
```
ASR<c>{<q>} {<Rd>,} <Rm>, #<imm> // (Inside IT block)
```

is equivalent to

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

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

T3



MOV, shift or rotate by value

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

is equivalent to

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

and is always the preferred disassembly.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rd>	For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. Arm deprecates using the PC as the destination register, but if the PC is used, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*.
	For encoding T2 and T3: is the general-purpose destination register, encoded in the "Rd" field.
<Rm>	For encoding A1: is the general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.
	For encoding T2 and T3: is the general-purpose source register, encoded in the "Rm" field.
<imm>	For encoding A1 and T2: is the shift amount, in the range 1 to 32, encoded in the "imm5" field as <imm> modulo 32.
	For encoding T3: is the shift amount, in the range 1 to 32, encoded in the "imm3:imm2" field as <imm> modulo 32.

Operation

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

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ASR (register)

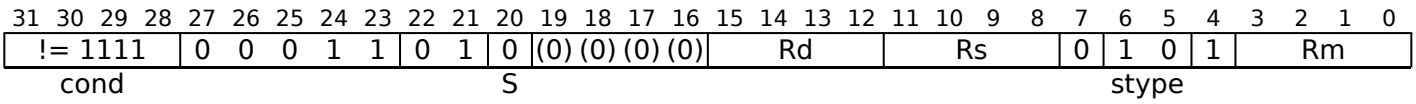
Arithmetic Shift Right (register) shifts a register value right by a variable number of bits, shifting in copies of its sign bit, and writes the result to the destination register. The variable number of bits is read from the bottom byte of a register

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

- The encodings in this description are named to match the encodings of MOV, MOVS (register-shifted register).
- The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.

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

A1



Not flag setting

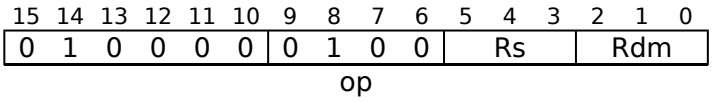
```
ASR{<c>}{<q>} {<Rd>,} <Rm>, <Rs>
```

is equivalent to

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

and is always the preferred disassembly.

T1



Arithmetic shift right

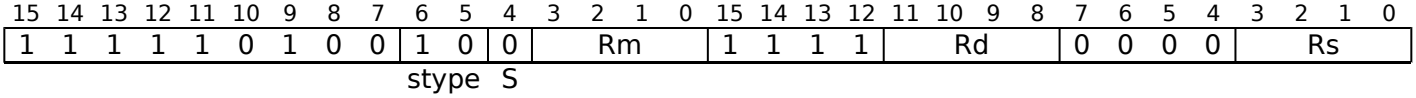
```
ASR<c>{<q>} {<Rdm>,} <Rdm>, <Rs> // (Inside IT block)
```

is equivalent to

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

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

T2



Not flag setting

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

is equivalent to

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

and is always the preferred disassembly.

Assembler Symbols

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

Operation

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

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ASRS (immediate)

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

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

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

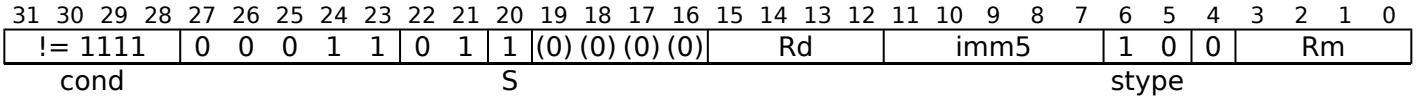
- The PE branches to the address written to the PC, and restores *PSTATE* from SPSR_<current_mode>.
- The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state*.
- The instruction is UNDEFINED in Hyp mode.
- The instruction is CONSTRAINED UNPREDICTABLE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

This is an alias of MOV, MOVS (register). This means:

- The encodings in this description are named to match the encodings of MOV, MOVS (register).
- The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.

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

A1



MOVS, shift or rotate by value

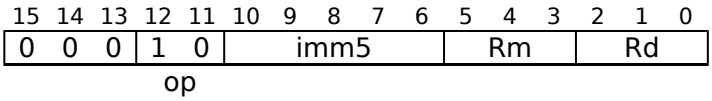
```
ASRS{<c>}{<q>} {<Rd>,} <Rm>, #<imm>
```

is equivalent to

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

and is always the preferred disassembly.

T2



T2

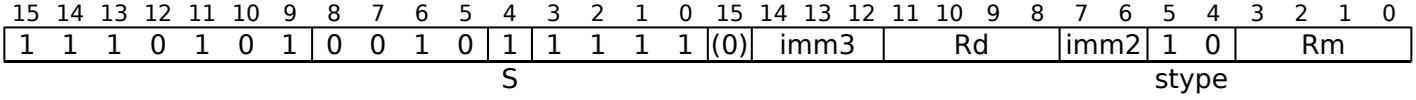
```
ASRS{<q>} {<Rd>,} <Rm>, #<imm> // (Outside IT block)
```

is equivalent to

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

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

T3



MOVS, shift or rotate by value

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

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

is equivalent to

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

and is always the preferred disassembly.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rd>	For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. Arm deprecates using the PC as the destination register, but if the PC is used, the instruction performs an exception return, that restores *PSTATE* from SPSR_<current_mode>.
	For encoding T2 and T3: is the general-purpose destination register, encoded in the "Rd" field.
<Rm>	For encoding A1: is the general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.
	For encoding T2 and T3: is the general-purpose source register, encoded in the "Rm" field.
<imm>	For encoding A1 and T2: is the shift amount, in the range 1 to 32, encoded in the "imm5" field as <imm> modulo 32.
	For encoding T3: is the shift amount, in the range 1 to 32, encoded in the "imm3:imm2" field as <imm> modulo 32.

Operation

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

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ASRS (register)

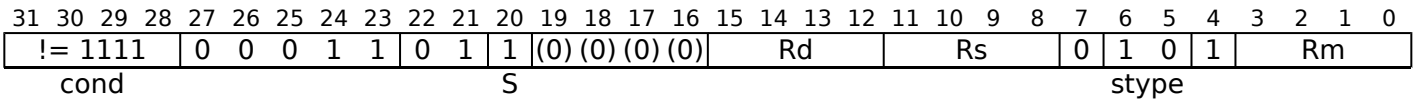
Arithmetic Shift Right, setting flags (register) shifts a register value right by a variable number of bits, shifting in copies of its sign bit, writes the result to the destination register, and updates the condition flags based on the result. The variable number of bits is read from the bottom byte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

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

- The encodings in this description are named to match the encodings of MOV, MOVS (register-shifted register).
- The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.

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

A1



Flag setting

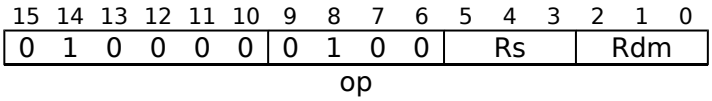
```
ASRS{<c>}{<q>} {<Rd>,} <Rm>, <Rs>
```

is equivalent to

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

and is always the preferred disassembly.

T1



Arithmetic shift right

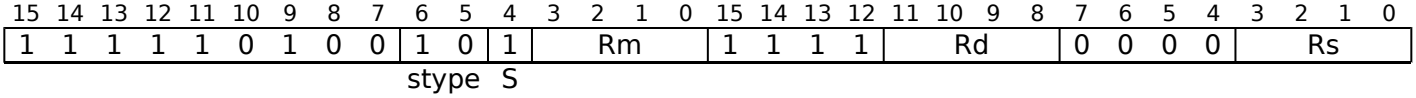
```
ASRS{<q>} {<Rdm>,} <Rdm>, <Rs> // (Outside IT block)
```

is equivalent to

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

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

T2



Flag setting

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

is equivalent to

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

and is always the preferred disassembly.

Assembler Symbols

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

Operation

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

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

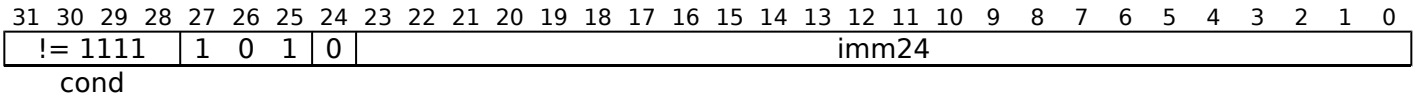
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

B

Branch causes a branch to a target address.

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

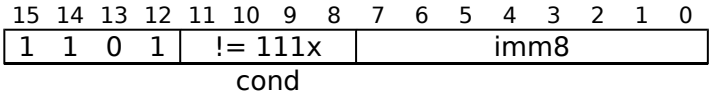
A1



A1

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

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

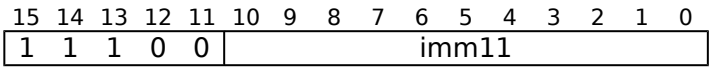
T1



T1

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

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

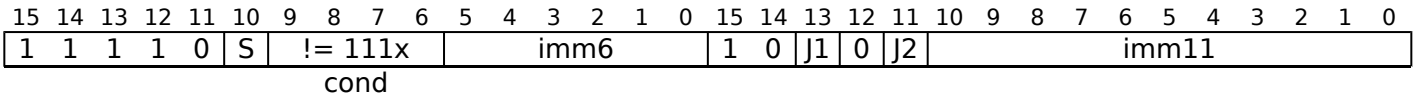
T2



T2

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

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

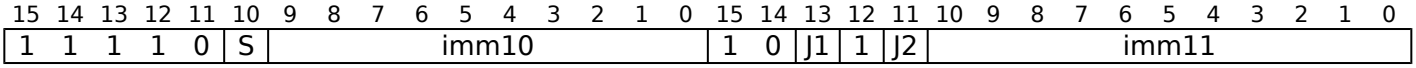
T3



T3

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

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

if cond<3:1> == '111' then SEE "Related encodings";
imm32 = SignExtend(S:J2:J1:imm6:imm11:'0', 32);
if InITBlock() then UNPREDICTABLE;
```

T4



T4

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

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

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

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

Assembler Symbols

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

Operation

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

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

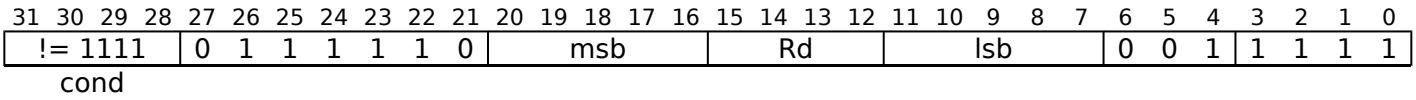
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

BFC

Bit Field Clear clears any number of adjacent bits at any position in a register, without affecting the other bits in the register.

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

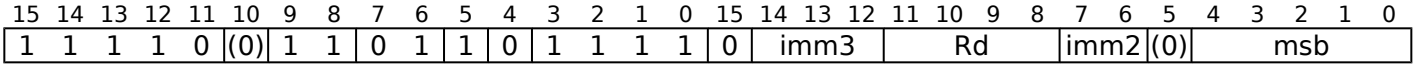
A1



A1

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

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

T1



T1

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

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

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

Assembler Symbols

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

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if msbit >= lsbit then
        R[d]<msbit:lsbit> = Replicate('0', msbit-lsbit+1);
        // Other bits of R[d] are unchanged
    else
        UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

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

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

Operational information

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

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

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

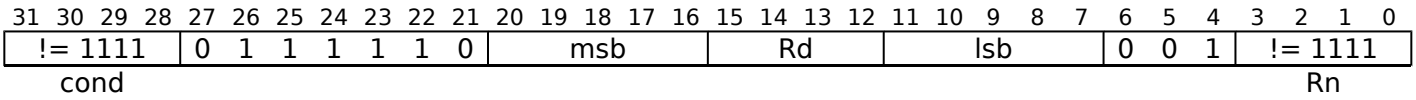
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

BFI

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

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

A1

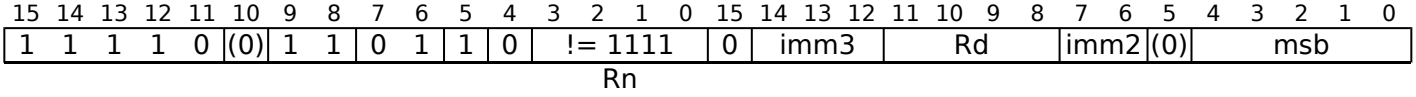


A1

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

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

T1



T1

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

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

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

Assembler Symbols

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

Operation

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

CONSTRAINED UNPREDICTABLE behavior

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

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

Operational information

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

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

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

BIC, BICS (immediate)

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

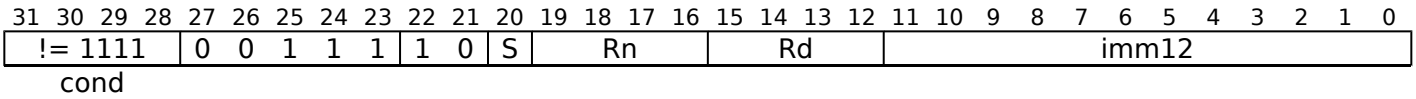
If the destination register is not the PC, the BICS variant of the instruction updates the condition flags based on the result.

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

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

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

A1



BIC (S == 0)

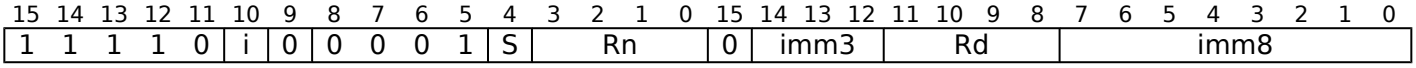
```
BIC{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

BICS (S == 1)

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

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

T1



BIC (S == 0)

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

BICS (S == 1)

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

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

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

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<Rd> For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. Arm deprecates using the PC as the destination register, but if the PC is used:

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

For encoding T1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.

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

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

<const> For encoding A1: an immediate value. See *Modified immediate constants in A32 instructions* for the range of values.

For encoding T1: an immediate value. See *Modified immediate constants in T32 instructions* for the range of values.

Operation

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

Operational information

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

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

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

BIC, BICS (register)

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

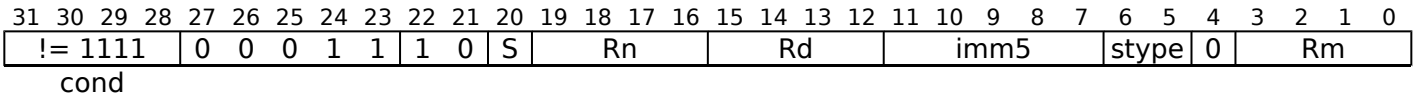
If the destination register is not the PC, the BICS variant of the instruction updates the condition flags based on the result.

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

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

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

A1



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

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

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

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

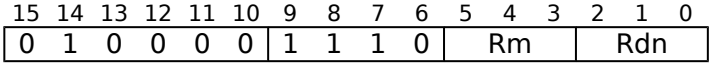
BICS, rotate right with extend (S == 1 && imm5 == 00000 && stype == 11)

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

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

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

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

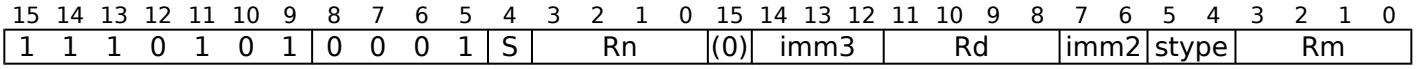
T1



T1

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

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

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

T2



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

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

BIC, shift or rotate by value (S == 0 && !(imm3 == 000 && imm2 == 00 && stype == 11))

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

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

BICS, rotate right with extend (S == 1 && imm3 == 000 && imm2 == 00 && stype == 11)

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

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

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

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

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

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

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

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

<Rd> For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. Arm deprecates using the PC as the destination register, but if the PC is used:

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

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

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

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

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

For encoding T1 and T2: is the second general-purpose source register, encoded in the "Rm" field.

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

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

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

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

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
    result = R[n] AND NOT(shifted);
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.N = result<31>;
            PSTATE.Z = IsZeroBit(result);
            PSTATE.C = carry;
            // PSTATE.V unchanged
```

Operational information

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

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

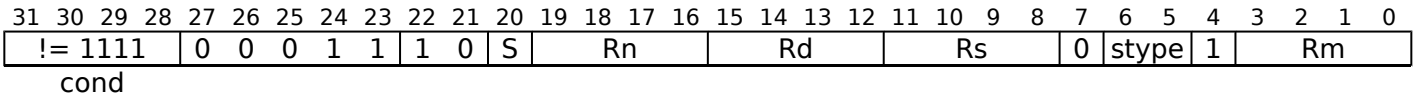
Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

BIC, BICS (register-shifted register)

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

A1



Flag setting (S == 1)

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

Not flag setting (S == 0)

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

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

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

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<Rd> Is the general-purpose destination register, encoded in the "Rd" field.

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

<Rm> Is the second general-purpose source register, encoded in the "Rm" field.

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

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

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

Operation

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

Operational information

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

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

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

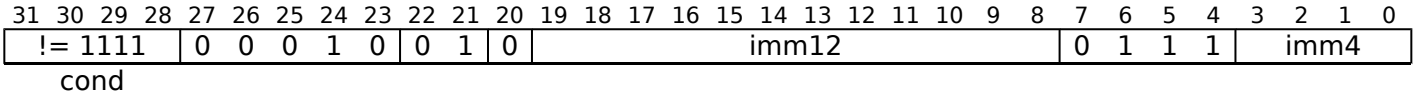
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

BKPT

Breakpoint causes a Breakpoint Instruction exception.
Breakpoint is always unconditional, even when inside an IT block.

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

A1



A1

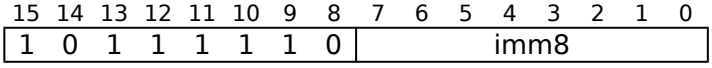
```
BKPT{<q>} {#}<imm>

imm16 = imm12:imm4;
if cond != '1110' then UNPREDICTABLE;  // BKPT must be encoded with AL condition
```

CONSTRAINED UNPREDICTABLE behavior

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

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes unconditionally.
- The instruction executes conditionally.

T1



T1

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

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

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

Assembler Symbols

<q>	See *Standard assembler syntax fields*. An `BKPT` instruction must be unconditional.
<imm>	For encoding A1: is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm12:imm4" field. This value: • Is recorded in the Comment field of ESR_ELx.ISS if the Software Breakpoint Instruction exception is taken to an exception level that is using AArch64. • Is ignored otherwise. For encoding T1: is a 8-bit unsigned immediate, in the range 0 to 255, encoded in the "imm8" field. This value: • Is recorded in the Comment field of ESR_ELx.ISS if the Software Breakpoint Instruction exception is taken to an exception level that is using AArch64. • Is ignored otherwise.

Operation

```
EncodingSpecificOperations();
AArch32.SoftwareBreakpoint(imm16);
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

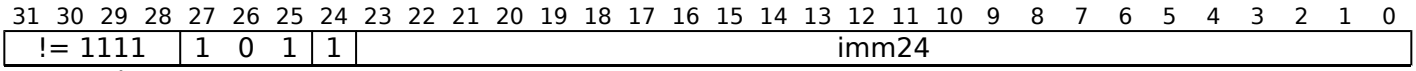
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

BL, BLX (immediate)

Branch with Link calls a subroutine at a PC-relative address, and setting LR to the return address.
Branch with Link and Exchange Instruction Sets (immediate) calls a subroutine at a PC-relative address, setting LR to the return address, and changes the instruction set from A32 to T32, or from T32 to A32.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2) .

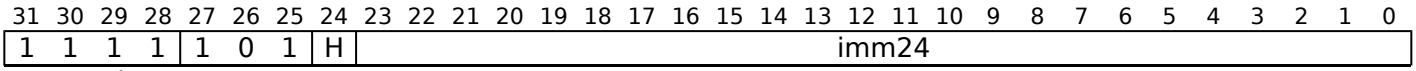
A1



A1

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

imm32 = SignExtend(imm24:'00', 32);  targetInstrSet = InstrSet_A32;
```

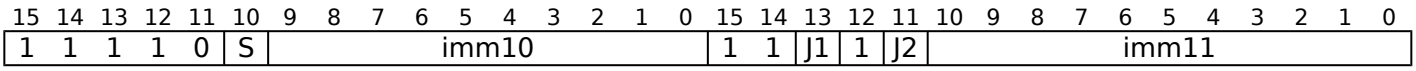
A2



A2

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

imm32 = SignExtend(imm24:H:'0', 32);  targetInstrSet = InstrSet_T32;
```

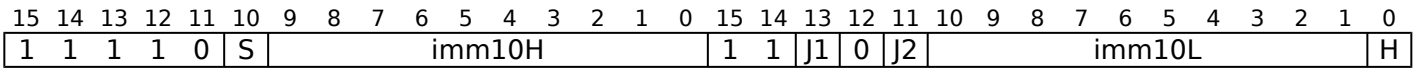
T1



T1

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

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

T2



T2

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

if H == '1' then UNDEFINED;
I1 = NOT(J1 EOR S);  I2 = NOT(J2 EOR S);  imm32 = SignExtend(S:I1:I2:imm10H:imm10L:'00', 32);
targetInstrSet = InstrSet_A32;
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

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

Assembler Symbols

<c>	For encoding A1, T1 and T2: see *Standard assembler syntax fields*. For encoding A2: see *Standard assembler syntax fields*. <c> must be AL or omitted.
<q>	See *Standard assembler syntax fields*.
<label>	For encoding A1: the label of the instruction that is to be branched to. The assembler calculates the required value of the offset from the PC value of the BL instruction to this label, then selects an encoding that sets imm32 to that offset. Permitted offsets are multiples of 4 in the range –33554432 to 33554428. For encoding A2: the label of the instruction that is to be branched to. The assembler calculates the required value of the offset from the PC value of the BLX instruction to this label, then selects an encoding with imm32 set to that offset. Permitted offsets are even numbers in the range –33554432 to 33554430. For encoding T1: the label of the instruction that is to be branched to. The assembler calculates the required value of the offset from the PC value of the BL instruction to this label, then selects an encoding with imm32 set to that offset. Permitted offsets are even numbers in the range –16777216 to 16777214. For encoding T2: the label of the instruction that is to be branched to. The assembler calculates the required value of the offset from the Align(PC, 4) value of the BLX instruction to this label, then selects an encoding with imm32 set to that offset. Permitted offsets are multiples of 4 in the range –16777216 to 16777212.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if CurrentInstrSet() == InstrSet_A32 then
        LR = PC - 4;
    else
        LR = PC<31:1> : '1';
    if targetInstrSet == InstrSet_A32 then
        targetAddress = Align(PC,4) + imm32;
    else
        targetAddress = PC + imm32;
    SelectInstrSet(targetInstrSet);
    BranchWritePC(targetAddress, BranchType_DIRCALL);
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

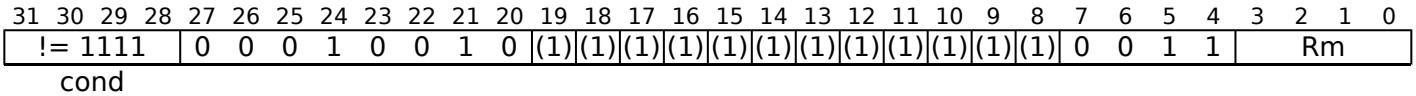
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

BLX (register)

Branch with Link and Exchange (register) calls a subroutine at an address specified in the register, and if necessary changes to the instruction set indicated by bit[0] of the register value. If the value in bit[0] is 0, the instruction set after the branch will be A32. If the value in bit[0] is 1, the instruction set after the branch will be T32.

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

A1

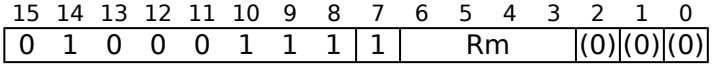


A1

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

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

T1



T1

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

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

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

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rm>	Is the general-purpose register holding the address to be branched to, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    target = R[m];
    if CurrentInstrSet() == InstrSet_A32 then
        next_instr_addr = PC - 4;
        LR = next_instr_addr;
    else
        next_instr_addr = PC - 2;
        LR = next_instr_addr<31:1> : '1';
    BXWritePC(target, BranchType_INDCALL);
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

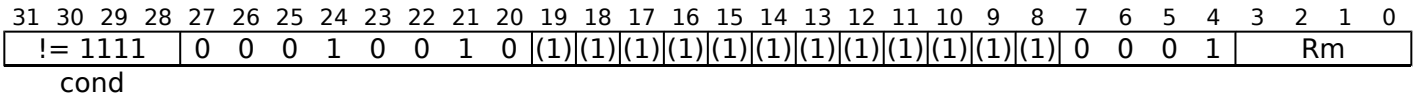
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

BX

Branch and Exchange causes a branch to an address and instruction set specified by a register.

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

A1

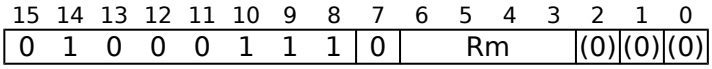


A1

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

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

T1



T1

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

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

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

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rm>	For encoding A1: is the general-purpose register holding the address to be branched to, encoded in the "Rm" field. The PC can be used. For encoding T1: is the general-purpose register holding the address to be branched to, encoded in the "Rm" field. The PC can be used. If <Rm> is the PC at a non word-aligned address, it results in UNPREDICTABLE behavior because the address passed to the BXWritePC() pseudocode function has bits<1:0> = '10'.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    BXWritePC(R[m], BranchType_INDIR);
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

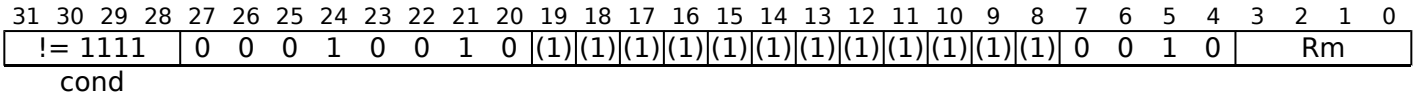
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

BXJ

Branch and Exchange, previously Branch and Exchange Jazelle.
In Armv8, BXJ behaves as a BX instruction, see *BX*. This means it causes a branch to an address and instruction set specified by a register.

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

A1

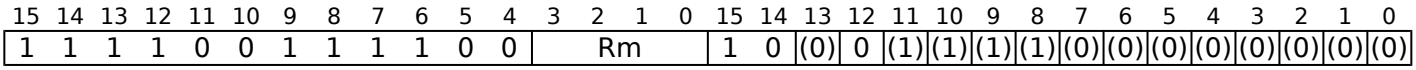


A1

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

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

T1



T1

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

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

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

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rm>	Is the general-purpose register holding the address to be branched to, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    BXWritePC(R[m], BranchType_INDIR);
```

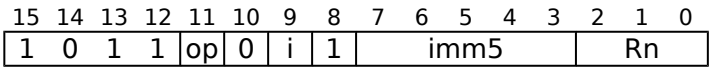
Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

CBNZ, CBZ

Compare and Branch on Nonzero and Compare and Branch on Zero compare the value in a register with zero, and conditionally branch forward a constant value. They do not affect the condition flags.

T1



CBNZ (op == 1)

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

CBZ (op == 0)

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

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

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

Assembler Symbols

<q>	See *Standard assembler syntax fields*.
<Rn>	Is the general-purpose register to be tested, encoded in the "Rn" field.
<label>	Is the program label to be conditionally branched to. Its offset from the PC, a multiple of 2 and in the range 0 to 126, is encoded as "i:imm5" times 2.

Operation

```
EncodingSpecificOperations();
if nonzero != IsZero(R[n]) then
    CBWritePC(PC + imm32);
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

CLREX

Clear-Exclusive clears the local monitor of the executing PE.

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

A1

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

A1

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

// No additional decoding required
```

T1

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

T1

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

// No additional decoding required
```

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

Assembler Symbols

<c>	For encoding A1: see *Standard assembler syntax fields*. Must be AL or omitted. For encoding T1: see *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.

Operation

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

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

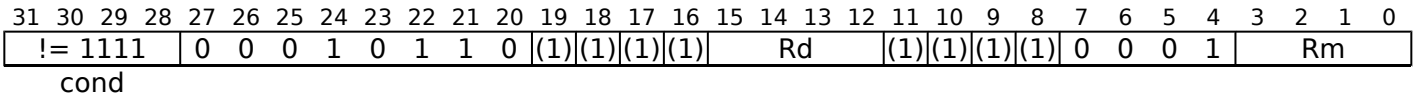
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

CLZ

Count Leading Zeros returns the number of binary zero bits before the first binary one bit in a value.

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

A1

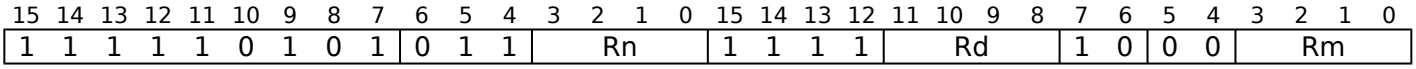


A1

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

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

T1



T1

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

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

CONSTRAINED UNPREDICTABLE behavior

If `m != n`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as described, with no change to its behavior and no additional side effects.
- The instruction executes with the additional decode: m = UInt(Rn);.
- The value in the destination register is UNKNOWN.

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

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rm>	For encoding A1: is the general-purpose source register, encoded in the "Rm" field. For encoding T1: is the general-purpose source register, encoded in the "Rm" field. It must be encoded with an identical value in the "Rn" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = CountLeadingZeroBits(R[m]);
    R[d] = result<31:0>;
```

Operational information

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

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

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

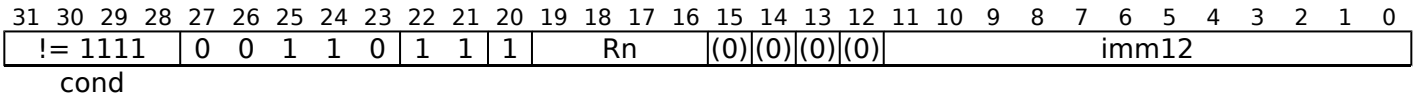
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

CMN (immediate)

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

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

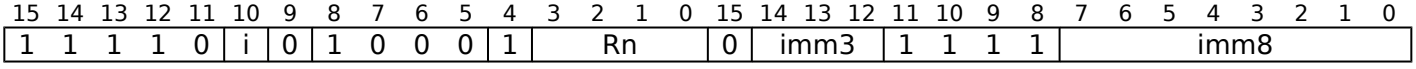
A1



A1

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

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

T1



T1

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

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

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

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rn>	For encoding A1: is the general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated. For encoding T1: is the general-purpose source register, encoded in the "Rn" field.
<const>	For encoding A1: an immediate value. See *Modified immediate constants in A32 instructions* for the range of values. For encoding T1: an immediate value. See *Modified immediate constants in T32 instructions* for the range of values.

Operation

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

Operational information

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

- The execution time of this instruction is independent of:

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

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

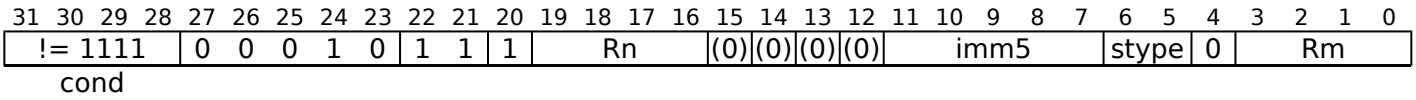
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

CMN (register)

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

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

A1



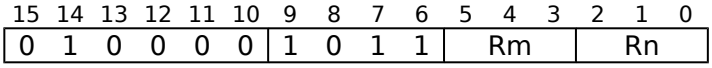
Rotate right with extend (imm5 == 00000 && stype == 11)

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

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

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

n = UInt(Rn);  m = UInt(Rm);
(shift_t, shift_n) = DecodeImmShift(stype, imm5);
```

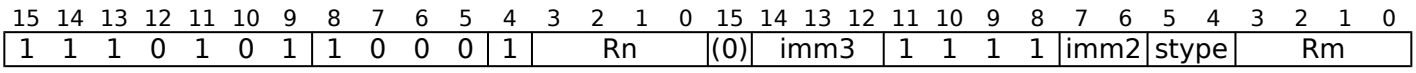
T1



T1

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

n = UInt(Rn);  m = UInt(Rm);
(shift_t, shift_n) = (SRType_LSL, 0);
```

T2



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

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

Shift or rotate by value (!(imm3 == 000 && imm2 == 00 && stype == 11))

```
CMN{<c>}.W <Rn>, <Rm> // (<Rn>, <Rm> can be represented in T1)

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

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

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

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

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

For encoding T1 and T2: is the first general-purpose source register, encoded in the "Rn" field.

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

For encoding T1 and T2: is the second general-purpose source register, encoded in the "Rm" field.

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

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

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

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

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (result, nzcv) = AddWithCarry(R[n], shifted, '0');
    PSTATE.<N,Z,C,V> = nzcv;
```

Operational information

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

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

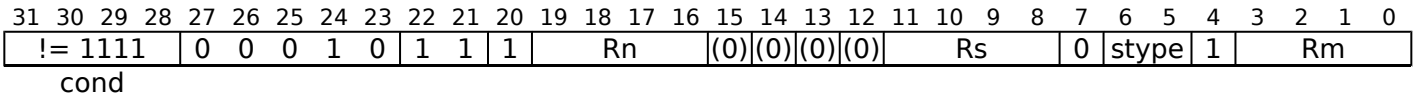
Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

CMN (register-shifted register)

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

A1



A1

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

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

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

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

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

<Rm> Is the second general-purpose source register, encoded in the "Rm" field.

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

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

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

Operation

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

Operational information

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

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

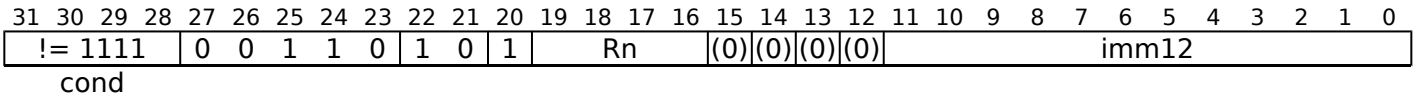
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

CMP (immediate)

Compare (immediate) subtracts an immediate value from a register value. It updates the condition flags based on the result, and discards the result.

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

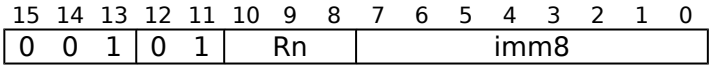
A1



A1

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

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

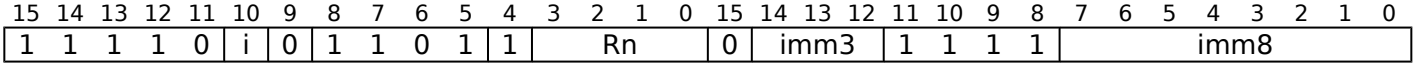
T1



T1

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

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

T2



T2

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

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

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

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

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rn>	For encoding A1: is the general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated. For encoding T1: is a general-purpose source register, encoded in the "Rn" field. For encoding T2: is the general-purpose source register, encoded in the "Rn" field.
<imm8>	Is a 8-bit unsigned immediate, in the range 0 to 255, encoded in the "imm8" field.
<const>	For encoding A1: an immediate value. See *Modified immediate constants in A32 instructions* for the range of values.

For encoding T2: an immediate value. See *Modified immediate constants in T32 instructions* for the range of values.

Operation

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

Operational information

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

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

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

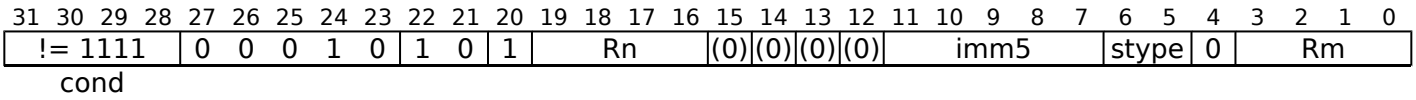
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

CMP (register)

Compare (register) subtracts an optionally-shifted register value from a register value. It updates the condition flags based on the result, and discards the result.

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

A1



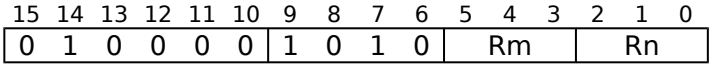
Rotate right with extend (imm5 == 00000 && stype == 11)

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

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

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

n = UInt(Rn);  m = UInt(Rm);
(shift_t, shift_n) = DecodeImmShift(stype, imm5);
```

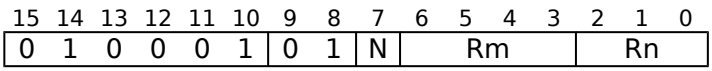
T1



T1

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

n = UInt(Rn);  m = UInt(Rm);
(shift_t, shift_n) = (SRType_LSL, 0);
```

T2



T2

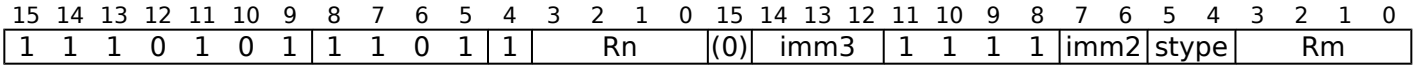
```
CMP{<c>}{<q>} <Rn>, <Rm> // (<Rn> and <Rm> not both from R0-R7)

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

CONSTRAINED UNPREDICTABLE behavior

If `n < 8 && m < 8`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as described, with no change to its behavior and no additional side effects.
- The condition flags become UNKNOWN.

T3



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

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

Shift or rotate by value (!(imm3 == 000 && imm2 == 00 && stype == 11))

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

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

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

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

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

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

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

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

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

For encoding T1, T2 and T3: is the second general-purpose source register, encoded in the "Rm" field.

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

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

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

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

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (result, nzcv) = AddWithCarry(R[n], NOT(shifted), '1');
    PSTATE.<N,Z,C,V> = nzcv;
```

Operational information

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

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

- The values of the data supplied in any of its registers.
- The values of the NZCV flags.

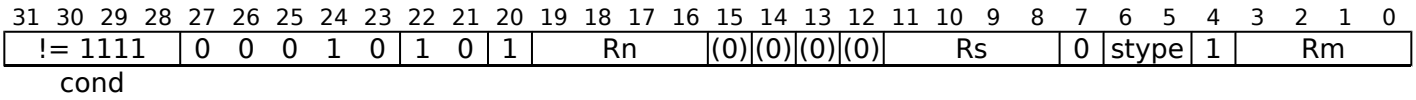
Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

CMP (register-shifted register)

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

A1



A1

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

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

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

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

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

<Rm> Is the second general-purpose source register, encoded in the "Rm" field.

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

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

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

Operation

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

Operational information

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

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

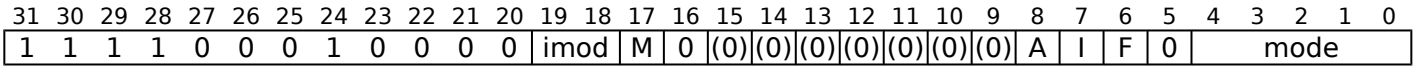
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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



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

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

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

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

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

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

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

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

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

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

```
if mode != '00000' && M == '0' then UNPREDICTABLE;
if (imod<1> == '1' && A:I:F == '000') || (imod<1> == '0' && A:I:F != '000') then UNPREDICTABLE;
enable = (imod == '10');  disable = (imod == '11');  changemode = (M == '1');
affectA = (A == '1');  affectI = (I == '1');  affectF = (F == '1');
if (imod == '00' && M == '0') || imod == '01' then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `imod == '01'`, then one of the following behaviors must occur:

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

If `imod == '00' && M == '0'`, then one of the following behaviors must occur:

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

If `mode != '00000' && M == '0'`, then one of the following behaviors must occur:

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

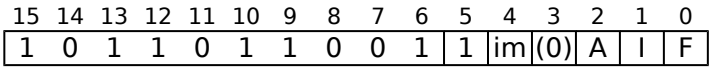
If `imod<1> == '1' && A:I:F == '000'`, then one of the following behaviors must occur:

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

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

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

T1



Interrupt disable (im == 1)

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

Interrupt enable (im == 0)

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

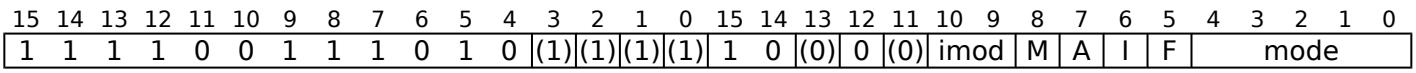
```
if A:I:F == '000' then UNPREDICTABLE;
enable = (im == '0');  disable = (im == '1');  changemode = FALSE;
affectA = (A == '1');  affectI = (I == '1');  affectF = (F == '1');
if InITBlock() then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `A:I:F == '000'`, then one of the following behaviors must occur:

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

T2



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

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

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

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

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

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

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

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

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

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

```
if imod == '00' && M == '0' then SEE "Hint instructions";
if mode != '00000' && M == '0' then UNPREDICTABLE;
if (imod<1> == '1' && A:I:F == '000') || (imod<1> == '0' && A:I:F != '000') then UNPREDICTABLE;
enable = (imod == '10');  disable = (imod == '11');  changemode = (M == '1');
affectA = (A == '1');  affectI = (I == '1');  affectF = (F == '1');
if imod == '01' || InITBlock() then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `imod == '01'`, then one of the following behaviors must occur:

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

If `mode != '00000' && M == '0'`, then one of the following behaviors must occur:

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

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

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

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

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

Hint instructions: In encoding T2, if the imod field is 00 and the M bit is 0, a hint instruction is encoded. To determine which hint instruction, see *Branches and miscellaneous control*.

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

Assembler Symbols

<q>	See *Standard assembler syntax fields*.

<iflags> Is a sequence of one or more of the following, specifying which interrupt mask bits are affected:

a
Sets the A bit in the instruction, causing the specified effect on *PSTATE*.A, the SError interrupt mask bit.

i
Sets the I bit in the instruction, causing the specified effect on *PSTATE*.I, the IRQ interrupt mask bit.

f
Sets the F bit in the instruction, causing the specified effect on *PSTATE*.F, the FIQ interrupt mask bit.

<mode> Is the number of the mode to change to, in the range 0 to 31, encoded in the "mode" field.

Operation

```
if CurrentInstrSet() == InstrSet_A32 then
    EncodingSpecificOperations();
    if PSTATE.EL != EL0 then
        if enable then
            if affectA then PSTATE.A = '0';
            if affectI then PSTATE.I = '0';
            if affectF then PSTATE.F = '0';
        if disable then
            if affectA then PSTATE.A = '1';
            if affectI then PSTATE.I = '1';
            if affectF then PSTATE.F = '1';
        if changemode then
            // AArch32.WriteModeByInstr() sets PSTATE.IL to 1 if this is an illegal mode change.
            AArch32.WriteModeByInstr(mode);
else
    EncodingSpecificOperations();
    if PSTATE.EL != EL0 then
        if enable then
            if affectA then PSTATE.A = '0';
            if affectI then PSTATE.I = '0';
            if affectF then PSTATE.F = '0';
        if disable then
            if affectA then PSTATE.A = '1';
            if affectI then PSTATE.I = '1';
            if affectF then PSTATE.F = '1';
        if changemode then
            // AArch32.WriteModeByInstr() sets PSTATE.IL to 1 if this is an illegal mode change.
            AArch32.WriteModeByInstr(mode);
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

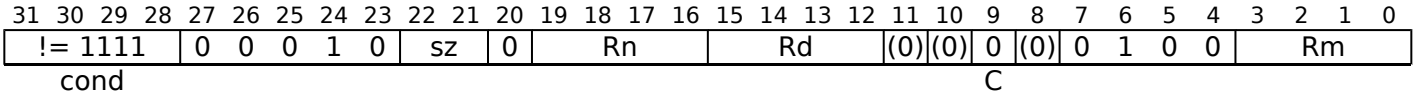
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

CRC32

CRC32 performs a cyclic redundancy check (CRC) calculation on a value held in a general-purpose register. It takes an input CRC value in the first source operand, performs a CRC on the input value in the second source operand, and returns the output CRC value. The second source operand can be 8, 16, or 32 bits. To align with common usage, the bit order of the values is reversed as part of the operation, and the polynomial 0x04C11DB7 is used for the CRC calculation.
In Armv8-A, this is an OPTIONAL instruction, and in Armv8.1 it is mandatory for all implementations to implement it. *ID_ISAR5*.CRC32 indicates whether this instruction is supported in the T32 and A32 instruction sets.

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

A1



CRC32B (sz == 00)

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

CRC32H (sz == 01)

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

CRC32W (sz == 10)

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

```
if ! HaveCRCExt() then UNDEFINED;
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
size = 8 << UInt(sz);
crc32c = (C == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
if size == 64 then UNPREDICTABLE;
if cond != '1110' then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

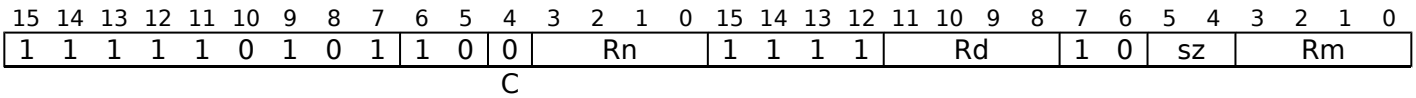
If `size == 64`, then one of the following behaviors must occur:

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

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

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

T1



CRC32B (sz == 00)

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

CRC32H (sz == 01)

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

CRC32W (sz == 10)

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

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

CONSTRAINED UNPREDICTABLE behavior

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

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

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

Assembler Symbols

<q>	See *Standard assembler syntax fields*. An CRC32 instruction must be unconditional.
<Rd>	Is the general-purpose accumulator output register, encoded in the "Rd" field.
<Rn>	Is the general-purpose accumulator input register, encoded in the "Rn" field.
<Rm>	Is the general-purpose data source register, encoded in the "Rm" field.

Operation

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

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

Operational information

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

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

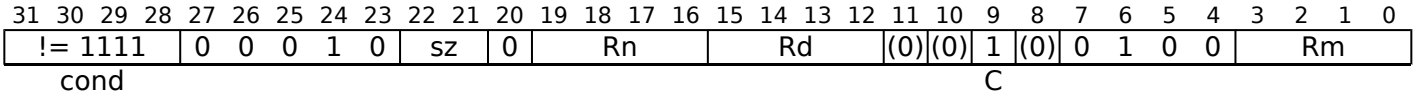
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

CRC32C

CRC32C performs a cyclic redundancy check (CRC) calculation on a value held in a general-purpose register. It takes an input CRC value in the first source operand, performs a CRC on the input value in the second source operand, and returns the output CRC value. The second source operand can be 8, 16, or 32 bits. To align with common usage, the bit order of the values is reversed as part of the operation, and the polynomial 0x1EDC6F41 is used for the CRC calculation.
In Armv8-A, this is an OPTIONAL instruction, and in Armv8.1 it is mandatory for all implementations to implement it. *ID_ISAR5*.CRC32 indicates whether this instruction is supported in the T32 and A32 instruction sets.

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

A1



CRC32CB (sz == 00)

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

CRC32CH (sz == 01)

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

CRC32CW (sz == 10)

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

```
if ! HaveCRCExt() then UNDEFINED;
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
size = 8 << UInt(sz);
crc32c = (C == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
if size == 64 then UNPREDICTABLE;
if cond != '1110' then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

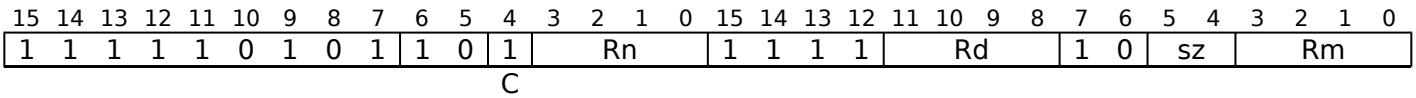
If `size == 64`, then one of the following behaviors must occur:

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

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

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

T1



CRC32CB (sz == 00)

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

CRC32CH (sz == 01)

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

CRC32CW (sz == 10)

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

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

CONSTRAINED UNPREDICTABLE behavior

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

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

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

Assembler Symbols

<q>	See *Standard assembler syntax fields*. An CRC32C instruction must be unconditional.
<Rd>	Is the general-purpose accumulator output register, encoded in the "Rd" field.
<Rn>	Is the general-purpose accumulator input register, encoded in the "Rn" field.
<Rm>	Is the general-purpose data source register, encoded in the "Rm" field.

Operation

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

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

Operational information

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

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

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

CSDB

Consumption of Speculative Data Barrier is a memory barrier that controls speculative execution and data value prediction.

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

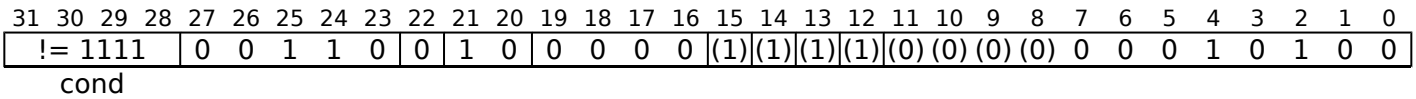
- Data value predictions of any instructions.
- PSTATE.{N,Z,C,V} predictions of any instructions other than conditional branch instructions and conditional instructions that write to the PC appearing in program order before the CSDB that have not been architecturally resolved.

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



A1

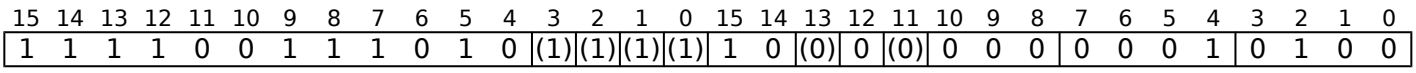
```
CSDB{<c>}{<q>}

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



T1

```
CSDB{<c>}{<q>}

if InITBlock() then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

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

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes unconditionally.
- The instruction executes conditionally.

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

Assembler Symbols

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

Operation

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

    ConsumptionOfSpeculativeDataBarrier();
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

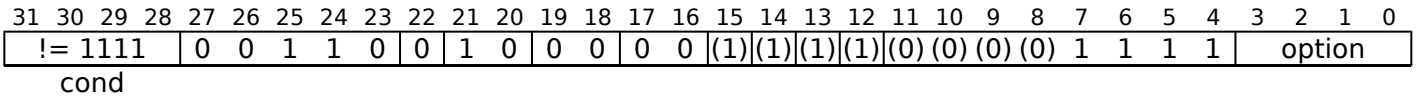
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

DBG

In Armv8, DBG executes as a NOP. Arm deprecates any use of the DBG instruction.

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

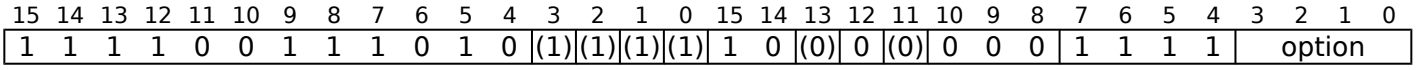
A1



A1

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

// DBG executes as a NOP. The 'option' field is ignored
```

T1



T1

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

// DBG executes as a NOP. The 'option' field is ignored
```

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

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<option>	Is a 4-bit unsigned immediate, in the range 0 to 15, encoded in the "option" field.

Operation

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

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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;

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

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

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

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

DCPS2

Debug Change PE State to EL2 allows the debugger to move the PE into EL2 from a lower Exception level.

DCPS2 is UNDEFINED if any of:

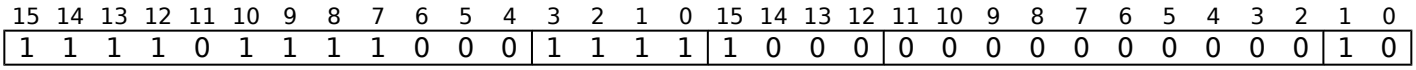
- The PE is in Non-debug state.
- EL2 is not implemented.
- The PE is in Secure state and any of:
  - Secure EL2 is not implemented.
  - Secure EL2 is implemented and Secure EL2 is disabled.

When the PE executes DCPS2:

- If EL2 is using AArch32, the PE enters Hyp mode and ELR_hyp, HSR, SPSR_hyp, DLR and DSPSR become UNKNOWN.
- If EL2 is using AArch64, the PE enters EL2 using AArch64, selects SP_EL2, and ELR_EL2, ESR_EL2, SPSR_EL2, DLR_EL0 and DSPSR_EL0 become UNKNOWN.

For more information on the operation of this instruction, see *DCPS*.

T1



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;

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

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

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

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

DCPS3

Debug Change PE State to EL3 allows the debugger to move the PE into EL3 from a lower Exception level or to a specific mode at the current Exception level.

DCPS3 is UNDEFINED if any of:

- The PE is in Non-debug state.
- EL3 is not implemented.
- EDSCR.SDD is set to 1.

When the PE executes DCPS3:

- If EL3 is using AArch32, the PE enters Monitor mode and LR_mon, SPSR_mon, DLR and DSPSR become UNKNOWN. If DCPS3 is executed in Monitor mode, SCR.NS is cleared to 0.
- If EL3 is using AArch64, the PE enters EL3 using AArch64, selects SP_EL3, and ELR_EL3, ESR_EL3, SPSR_EL3, DLR_EL0 and DSPSR_EL0 become UNKNOWN.

For more information on the operation of 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 SCTLR.SPAN == '0' then
            PSTATE.PAN = '1';
    PSTATE.E = SCTLR.EE;

    LR_mon = bits(32) UNKNOWN;
    SPSR_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;

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

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

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

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

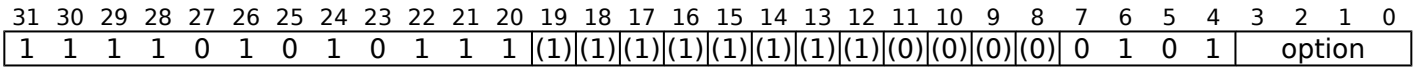
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

DMB

Data Memory Barrier is a memory barrier that ensures the ordering of observations of memory accesses, see *Data Memory Barrier (DMB)*.

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

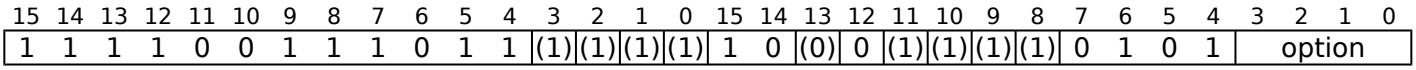
A1



A1

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

// No additional decoding required
```

T1



T1

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

// No additional decoding required
```

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

Assembler Symbols

<c> For encoding A1: see *Standard assembler syntax fields*. Must be AL or omitted.

For encoding T1: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<option> Specifies an optional limitation on the barrier operation. Values are:

SY
Full system is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Can be omitted. This option is referred to as the full system barrier. Encoded as option = 0b1111.

ST
Full system is the required shareability domain, writes are the required access type, both before and after the barrier instruction. SYST is a synonym for ST. Encoded as option = 0b1110.

LD
Full system is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as option = 0b1101.

ISH
Inner Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as option = 0b1011.

ISHST
Inner Shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as option = 0b1010.

ISHLD
Inner Shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as option = 0b1001.

NSH
Non-shareable is the required shareability domain, reads and writes are the required access, both before and after the barrier instruction. Encoded as option = 0b0111.

NSHST
Non-shareable is the required shareability domain, writes are the required access type both before and after the barrier instruction. Encoded as option = 0b0110.

NSHLD
Non-shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as option = 0b0101.

OSH
Outer Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as option = 0b0011.

OSHST
Outer Shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as option = 0b0010.

OSHLD
Outer Shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as option = 0b0001.

For more information on whether an access is before or after a barrier instruction, see *Data Memory Barrier (DMB)*. All other encodings of option are reserved. All unsupported and reserved options must execute as a full system DMB operation, but software must not rely on this behavior.

The instruction supports the following alternative <option> values, but Arm recommends that software does not use these alternative values:

- SH as an alias for ISH.
- SHST as an alias for ISHST.
- UN as an alias for NSH.
- UNST as an alias for NSHST.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    case option of
        when '0001'  domain = MBReqDomain_OuterShareable;  types = MBReqTypes_Reads;
        when '0010'  domain = MBReqDomain_OuterShareable;  types = MBReqTypes_Writes;
        when '0011'  domain = MBReqDomain_OuterShareable;  types = MBReqTypes_All;
        when '0101'  domain = MBReqDomain_Nonshareable;    types = MBReqTypes_Reads;
        when '0110'  domain = MBReqDomain_Nonshareable;    types = MBReqTypes_Writes;
        when '0111'  domain = MBReqDomain_Nonshareable;    types = MBReqTypes_All;
        when '1001'  domain = MBReqDomain_InnerShareable;  types = MBReqTypes_Reads;
        when '1010'  domain = MBReqDomain_InnerShareable;  types = MBReqTypes_Writes;
        when '1011'  domain = MBReqDomain_InnerShareable;  types = MBReqTypes_All;
        when '1101'  domain = MBReqDomain_FullSystem;      types = MBReqTypes_Reads;
        when '1110'  domain = MBReqDomain_FullSystem;      types = MBReqTypes_Writes;
        otherwise    domain = MBReqDomain_FullSystem;      types = MBReqTypes_All;

    if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then
        if HCR.BSU == '11' then
            domain = MBReqDomain_FullSystem;
        if HCR.BSU == '10' && domain != MBReqDomain_FullSystem then
            domain = MBReqDomain_OuterShareable;
        if HCR.BSU == '01' && domain == MBReqDomain_Nonshareable then
            domain = MBReqDomain_InnerShareable;

    DataMemoryBarrier(domain, types);
```

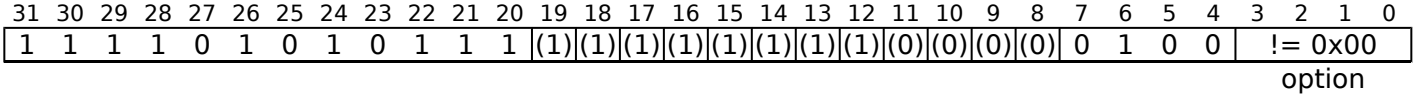
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

DSB

Data Synchronization Barrier is a memory barrier that ensures the completion of memory accesses, see *Data Synchronization Barrier (DSB)*.
An AArch32 DSB instruction does not require the completion of any AArch64 TLB maintenance instructions, regardless of the nXS qualifier, appearing in program order before the AArch32 DSB.

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

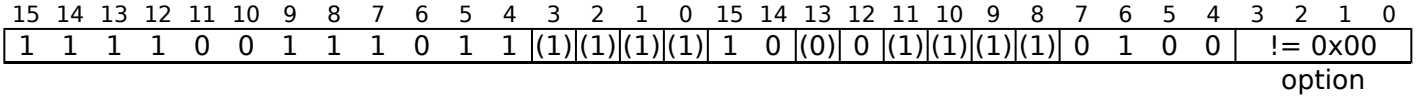
A1



A1

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

// No additional decoding required
```

T1



T1

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

// No additional decoding required
```

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

Assembler Symbols

<c> For encoding A1: see *Standard assembler syntax fields*. Must be AL or omitted.

For encoding T1: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<option> Specifies an optional limitation on the barrier operation. Values are:

SY
Full system is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Can be omitted. This option is referred to as the full system barrier. Encoded as option = 0b1111.

ST
Full system is the required shareability domain, writes are the required access type, both before and after the barrier instruction. SYST is a synonym for ST. Encoded as option = 0b1110.

LD
Full system is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as option = 0b1101.

ISH
Inner Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as option = 0b1011.

ISHST
Inner Shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as option = 0b1010.

ISHLD
Inner Shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as option = 0b1001.

NSH
Non-shareable is the required shareability domain, reads and writes are the required access, both before and after the barrier instruction. Encoded as option = 0b0111.

NSHST
Non-shareable is the required shareability domain, writes are the required access type both before and after the barrier instruction. Encoded as option = 0b0110.

NSHLD
Non-shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as option = 0b0101.

OSH
Outer Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as option = 0b0011.

OSHST
Outer Shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as option = 0b0010.

OSHLD
Outer Shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as option = 0b0001.

For more information on whether an access is before or after a barrier instruction, see *Data Synchronization Barrier (DSB)*. All other encodings of option are reserved, other than the values 0b0000 and 0b0100. All unsupported and reserved options must execute as a full system DSB operation, but software must not rely on this behavior.

The value 0b0000 is used to encode SSBB and the value 0b0100 is used to encode PSSBB.

The instruction supports the following alternative <option> values, but Arm recommends that software does not use these alternative values:

- SH as an alias for ISH.
- SHST as an alias for ISHST.
- UN as an alias for NSH.
- UNST as an alias for NSHST.

Operation

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

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

    case option of
        when '0001'  domain = MBReqDomain_OuterShareable;  types = MBReqTypes_Reads;
        when '0010'  domain = MBReqDomain_OuterShareable;  types = MBReqTypes_Writes;
        when '0011'  domain = MBReqDomain_OuterShareable;  types = MBReqTypes_All;
        when '0101'  domain = MBReqDomain_Nonshareable;    types = MBReqTypes_Reads;
        when '0110'  domain = MBReqDomain_Nonshareable;    types = MBReqTypes_Writes;
        when '0111'  domain = MBReqDomain_Nonshareable;    types = MBReqTypes_All;
        when '1001'  domain = MBReqDomain_InnerShareable;  types = MBReqTypes_Reads;
        when '1010'  domain = MBReqDomain_InnerShareable;  types = MBReqTypes_Writes;
        when '1011'  domain = MBReqDomain_InnerShareable;  types = MBReqTypes_All;
        when '1101'  domain = MBReqDomain_FullSystem;      types = MBReqTypes_Reads;
        when '1110'  domain = MBReqDomain_FullSystem;      types = MBReqTypes_Writes;
        otherwise
            if       option == '0000' then SEE "SSBB";
            elsif    option == '0100' then SEE "PSSBB";
            else     domain = MBReqDomain_FullSystem;      types = MBReqTypes_All;

    if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then
        if HCR.BSU == '11' then
            domain = MBReqDomain_FullSystem;
        if HCR.BSU == '10' && domain != MBReqDomain_FullSystem then
            domain = MBReqDomain_OuterShareable;
        if HCR.BSU == '01' && domain == MBReqDomain_Nonshareable then
            domain = MBReqDomain_InnerShareable;

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

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

EOR, EORS (immediate)

Bitwise Exclusive OR (immediate) performs a bitwise Exclusive OR of a register value and an immediate value, and writes the result to the destination register.

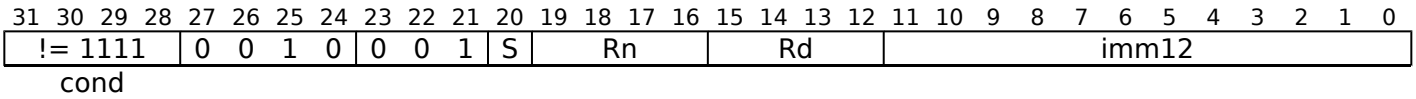
If the destination register is not the PC, the EORS variant of the instruction updates the condition flags based on the result.

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

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

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

A1



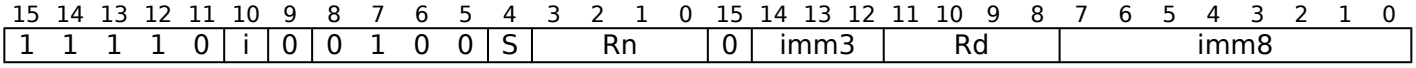
EOR (S == 0)

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

EORS (S == 1)

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

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

T1



EOR (S == 0)

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

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

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

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

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

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<Rd> For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. Arm deprecates using the PC as the destination register, but if the PC is used:

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

For encoding T1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.

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

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

<const> For encoding A1: an immediate value. See *Modified immediate constants in A32 instructions* for the range of values.

For encoding T1: an immediate value. See *Modified immediate constants in T32 instructions* for the range of values.

Operation

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

Operational information

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

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

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

EOR, EORS (register)

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

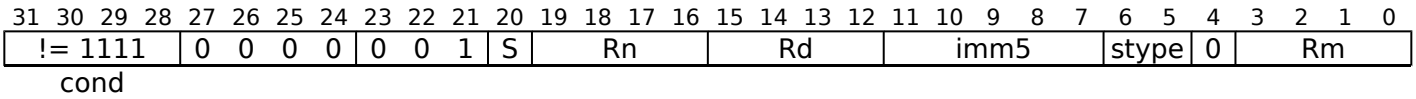
If the destination register is not the PC, the EORS variant of the instruction updates the condition flags based on the result.

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

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

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

A1



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

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

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

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

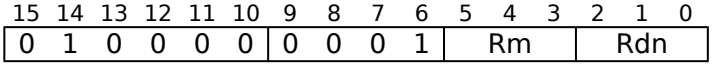
EORS, rotate right with extend (S == 1 && imm5 == 00000 && stype == 11)

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

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

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

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

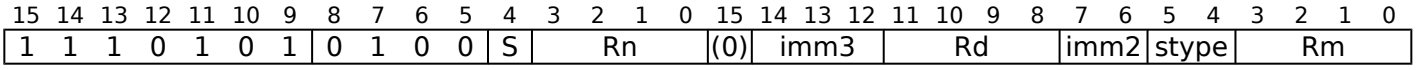
T1



T1

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

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

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

T2



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

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

EOR, shift or rotate by value (S == 0 && !(imm3 == 000 && imm2 == 00 && stype == 11))

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

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

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

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

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

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

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

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

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

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

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

<Rd> For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. Arm deprecates using the PC as the destination register, but if the PC is used:

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

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

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

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

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

For encoding T1 and T2: is the second general-purpose source register, encoded in the "Rm" field.

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

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

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

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

In T32 assembly:

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

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

Operation

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

Operational information

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

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

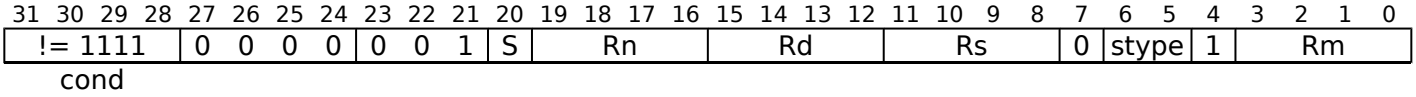
Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

EOR, EORS (register-shifted register)

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

A1



Flag setting (S == 1)

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

Not flag setting (S == 0)

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

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

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

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<Rd> Is the general-purpose destination register, encoded in the "Rd" field.

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

<Rm> Is the second general-purpose source register, encoded in the "Rm" field.

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

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

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

Operation

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

Operational information

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

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

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ERET

Exception Return.

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

The register holding the preferred return address is:

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

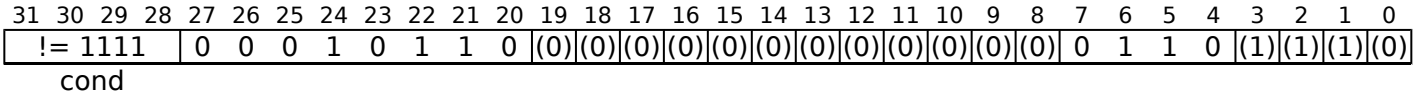
The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state*.

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

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

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

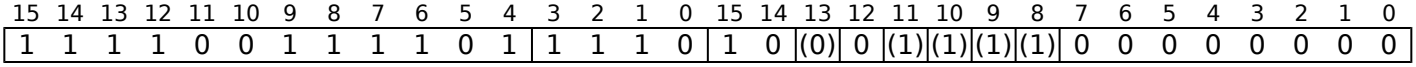
A1



A1

```
ERET{<c>}{<q>}

// No additional decoding required
```

T1



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;
        elsif PSTATE.M == M32_System then
            UNPREDICTABLE;                          // UNDEFINED or NOP
        else
            SynchronizeContext();
            bits(32) spsr = 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_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ESB

Error Synchronization Barrier is an error synchronization event that might also update DISR and VDISR. This instruction can be used at all Exception levels and in Debug state.

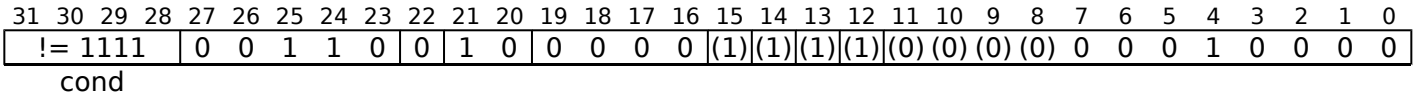
In Debug state, this instruction behaves as if SError interrupts are masked at all Exception levels. See Error Synchronization Barrier in the ARM(R) Reliability, Availability, and Serviceability (RAS) Specification, Armv8, for Armv8-A architecture profile.

If the RAS Extension is not implemented, this instruction executes as a NOP.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

(FEAT_RAS)



A1

```
ESB{<c>}{<q>}

if !HaveRASExt() then EndOfInstruction();  // Instruction executes as NOP
if cond != '1110' then UNPREDICTABLE;      // ESB must be encoded with AL condition
```

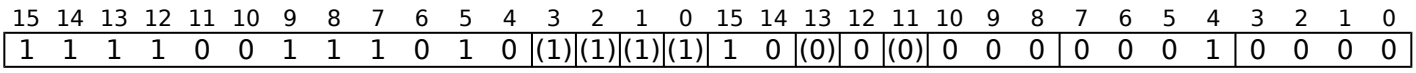
CONSTRAINED UNPREDICTABLE behavior

If `cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes unconditionally.
- The instruction executes conditionally.

T1

(FEAT_RAS)



T1

```
ESB{<c>}{<q>}

if !HaveRASExt() then EndOfInstruction();  // Instruction executes as NOP
if InITBlock() then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes unconditionally.
- The instruction executes conditionally.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();

    SynchronizeErrors();
    AArch32.ESBOperation();
    if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then AArch32.vESBOperation();
    TakeUnmaskedSErrorInterrupts();
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

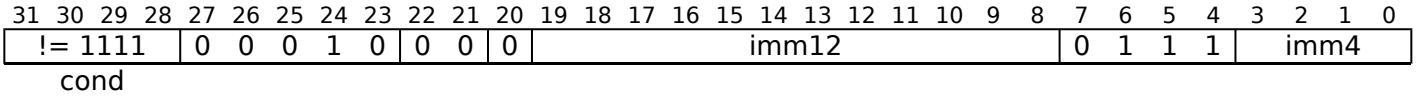
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

HLT

Halting breakpoint causes a software breakpoint to occur.
Halting breakpoint is always unconditional, even inside an IT block.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



A1

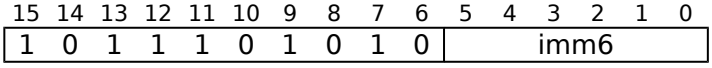
```
HLT{<q>} {#}<imm>

if EDSCR.HDE == '0' || !HaltingAllowed() then UNDEFINED;
if cond != '1110' then UNPREDICTABLE; // HLT must be encoded with AL condition
```

CONSTRAINED UNPREDICTABLE behavior

If `cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes unconditionally.
- The instruction executes conditionally.

T1



T1

```
HLT{<q>} {#}<imm>

if EDSCR.HDE == '0' || !HaltingAllowed() then UNDEFINED;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<q>	See *Standard assembler syntax fields*. An `HLT` instruction must be unconditional.
<imm>	For encoding A1: is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm12:imm4" field. This value is for assembly and disassembly only. It is ignored by the PE, but can be used by a debugger to store more information about the halting breakpoint. For encoding T1: is a 6-bit unsigned immediate, in the range 0 to 63, encoded in the "imm6" field. This value is for assembly and disassembly only. It is ignored by the PE, but can be used by a debugger to store more information about the halting breakpoint.

Operation

```
EncodingSpecificOperations();
Halt(DebugHalt_HaltInstruction);
```

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

HVC

Hypervisor Call causes a Hypervisor Call exception. For more information, see *Hypervisor Call (HVC) exception*. Software executing at EL1 can use this instruction to call the hypervisor to request a service.

The HVC instruction is UNDEFINED:

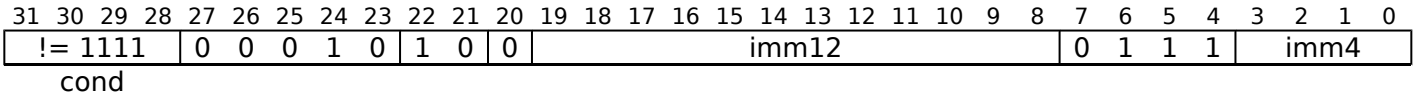
- When EL3 is implemented and using AArch64, and *SCR_EL3*.HCE is set to 0.
- In Non-secure EL1 modes when EL3 is implemented and using AArch32, and *SCR*.HCE is set to 0.
- When EL3 is not implemented and either *HCR_EL2*.HCD is set to 1 or *HCR*.HCD is set to 1.
- When EL2 is not implemented.
- In Secure state, if EL2 is not enabled in the current Security state.
- In User mode.

The HVC instruction is CONSTRAINED UNPREDICTABLE in Hyp mode when EL3 is implemented and using AArch32, and *SCR*.HCE is set to 0.

On executing an HVC instruction, the *HSR, Hyp Syndrome Register* reports the exception as a Hypervisor Call exception, using the EC value 0x12, and captures the value of the immediate argument, see *Use of the HSR*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



A1

```
HVC{<q>} {#}<imm16>
```

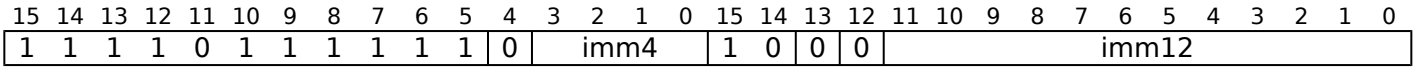
```
if cond != '1110' then UNPREDICTABLE;
imm16 = imm12:imm4;
```

CONSTRAINED UNPREDICTABLE behavior

If `cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes unconditionally.
- The instruction executes conditionally.

T1



T1

```
HVC{<q>} {#}<imm16>
```

```
imm16 = imm4:imm12;
if InITBlock() then UNPREDICTABLE;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<q>	See *Standard assembler syntax fields*. An HVC instruction must be unconditional.
<imm16>	For encoding A1: is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm12:imm4" field. This value is for assembly and disassembly only. It is reported in the HSR but

otherwise is ignored by hardware. An HVC handler might interpret imm16, for example to determine the required service.

For encoding T1: is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm4:imm12" field. This value is for assembly and disassembly only. It is reported in the HSR but otherwise is ignored by hardware. An HVC handler might interpret imm16, for example to determine the required service.

Operation

```
EncodingSpecificOperations();
if !HaveEL(EL2) || PSTATE.EL == EL0 || (IsSecure() && !IsSecureEL2Enabled()) then
    UNDEFINED;

if HaveEL(EL3) then
    if ELUsingAArch32(EL3) && SCR.HCE == '0' && PSTATE.EL == EL2 then
        UNPREDICTABLE;
    else
        hvc_enable = SCR_GEN[].HCE;
else
    hvc_enable = if ELUsingAArch32(EL2) then NOT(HCR.HCD) else NOT(HCR_EL2.HCD);

if hvc_enable == '0' then
    UNDEFINED;
else
    AArch32.CallHypervisor(imm16);
```

CONSTRAINED UNPREDICTABLE behavior

If `ELUsingAArch32(EL3) && SCR.HCE == '0' && PSTATE.EL == EL2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

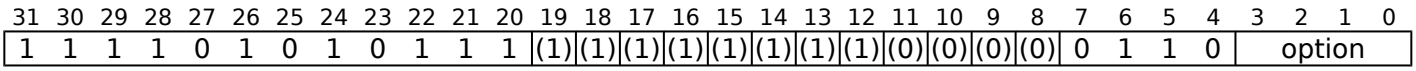
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ISB

Instruction Synchronization Barrier flushes the pipeline in the PE and is a context synchronization event. For more information, see *Instruction Synchronization Barrier (ISB)*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

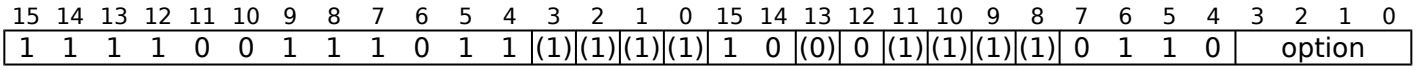
A1



A1

```
ISB{<c>}{<q>} {<option>}

// No additional decoding required
```

T1



T1

```
ISB{<c>}{<q>} {<option>}

// No additional decoding required
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c> For encoding A1: see *Standard assembler syntax fields*. Must be AL or omitted.

For encoding T1: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<option> Specifies an optional limitation on the barrier operation. Values are:

SY
Full system barrier operation, encoded as option = 0b1111. Can be omitted.

All other encodings of option are reserved. The corresponding instructions execute as full system barrier operations, but must not be relied upon by software.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    InstructionSynchronizationBarrier();
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

IT

If-Then makes up to four following instructions (the IT block) conditional. The conditions for the instructions in the IT block are the same as, or the inverse of, the condition the `IT` instruction specifies for the first instruction in the block.

The `IT` instruction itself does not affect the condition flags, but the execution of the instructions in the IT block can change the condition flags.

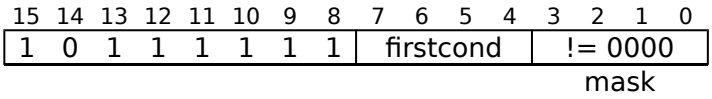
16-bit instructions in the IT block, other than `CMP`, `CMN` and `TST`, do not set the condition flags. An `IT` instruction with the AL condition can change the behavior without conditional execution.

The architecture permits exception return to an instruction in the IT block only if the restoration of the *CPSR* restores *PSTATE*.IT to a state consistent with the conditions specified by the `IT` instruction. Any other exception return to an instruction in an IT block is UNPREDICTABLE. Any branch to a target instruction in an IT block is not permitted, and if such a branch is made it is UNPREDICTABLE what condition is used when executing that target instruction and any subsequent instruction in the IT block.

Many uses of the IT instruction are deprecated for performance reasons, and an implementation might include ITD controls that can disable those uses of IT, making them UNDEFINED.

For more information see *Conditional execution* and *Conditional instructions*. The first of these sections includes more information about the ITD controls.

T1



T1

```
IT{<x>{<y>{<z>}}}{<q>} <cond>
```

```
if mask == '0000' then SEE "Related encodings";
if firstcond == '1111' || (firstcond == '1110' && BitCount(mask) != 1) then UNPREDICTABLE;
if InITBlock() then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `firstcond == '1111' || (firstcond == '1110' && BitCount(mask) != 1)`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The '1111' condition is treated as being the same as the '1110' condition, meaning always, and the ITSTATE state machine is progressed in the same way as for any other cond_base value.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Related encodings: *Miscellaneous 16-bit instructions*.

Assembler Symbols

<x> The condition for the second instruction in the IT block. If omitted, the "mask" field is set to 0b1000. If present it is encoded in the "mask[3]" field:

T
firstcond[0]

E
NOT firstcond[0]

<y> The condition for the third instruction in the IT block. If omitted and <x> is present, the "mask[2:0]" field is set to 0b100. If <y> is present it is encoded in the "mask[2]" field:

T
firstcond[0]

E
NOT firstcond[0]

<z> The condition for the fourth instruction in the IT block. If omitted and <y> is present, the "mask[1:0]" field is set to 0b10. If <z> is present, the "mask[0]" field is set to 1, and it is encoded in the "mask[1]" field:

T
firstcond[0]

E
NOT firstcond[0]

<q> See *Standard assembler syntax fields*.

<cond> The condition for the first instruction in the IT block, encoded in the "firstcond" field. See *Condition codes* for the range of conditions available, and the encodings.

The conditions specified in an `IT` instruction must match those specified in the syntax of the instructions in its IT block. When assembling to A32 code, assemblers check `IT` instruction syntax for validity but do not generate assembled instructions for them. See *Conditional instructions*.

Operation

```
EncodingSpecificOperations();
AArch32.CheckITEnabled(mask);
PSTATE.IT<7:0> = firstcond:mask;
ShouldAdvanceIT = FALSE;
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

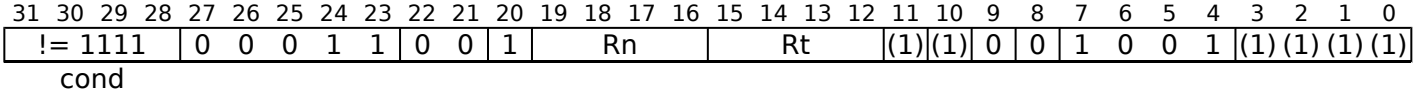
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDA

Load-Acquire Word loads a word from memory and writes it to a register. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release*
For more information about support for shared memory see *Synchronization and semaphores*. For information about memory accesses see *Memory accesses*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

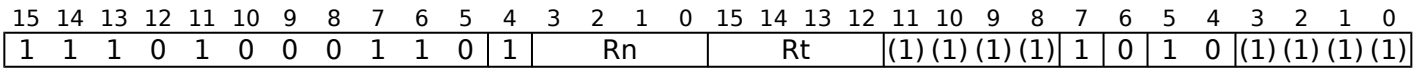
A1



A1

```
LDA{<c>}{<q>} <Rt>, [<Rn>]

t = UInt(Rt); n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

T1



T1

```
LDA{<c>}{<q>} <Rt>, [<Rn>]

t = UInt(Rt); n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rt>	Is the general-purpose register to be transferred, encoded in the "Rt" field.
<Rn>	Is the general-purpose base register, encoded in the "Rn" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    R[t] = MemO[address, 4];
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

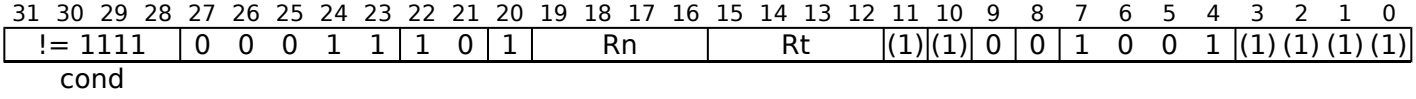
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDAB

Load-Acquire Byte loads a byte from memory, zero-extends it to form a 32-bit word and writes it to a register. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release*.
For more information about support for shared memory see *Synchronization and semaphores*. For information about memory accesses see *Memory accesses*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

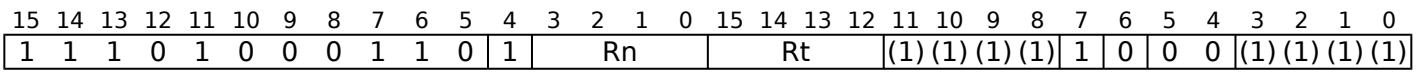
A1



A1

```
LDAB{<c>}{<q>} <Rt>, [<Rn>]

t = UInt(Rt); n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

T1



T1

```
LDAB{<c>}{<q>} <Rt>, [<Rn>]

t = UInt(Rt); n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rt>	Is the general-purpose register to be transferred, encoded in the "Rt" field.
<Rn>	Is the general-purpose base register, encoded in the "Rn" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    R[t] = ZeroExtend(MemO[address, 1], 32);
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDAEX

Load-Acquire Exclusive Word loads a word from memory, writes it to a register and:

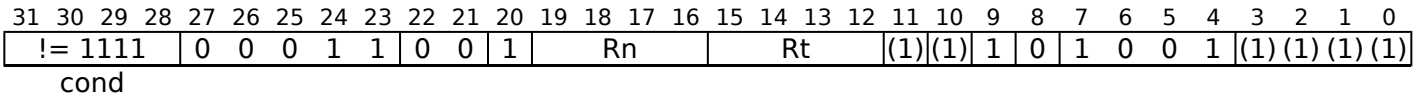
- If the address has the Shared Memory attribute, marks the physical address as exclusive access for the executing PE in a global monitor.
- Causes the executing PE to indicate an active exclusive access in the local monitor.

The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release*.

For more information about support for shared memory see *Synchronization and semaphores*. For information about memory accesses see *Memory accesses*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

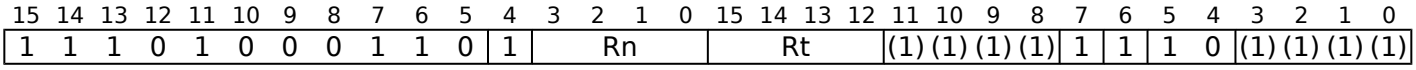
A1



A1

```
LDAEX{<c>}{<q>} <Rt>, [<Rn>]

t = UInt(Rt);  n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

T1



T1

```
LDAEX{<c>}{<q>} <Rt>, [<Rn>]

t = UInt(Rt);  n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rt>	Is the general-purpose register to be transferred, encoded in the "Rt" field.
<Rn>	Is the general-purpose base register, encoded in the "Rn" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    AArch32.SetExclusiveMonitors(address, 4);
    R[t] = MemO[address, 4];
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDAEXB

Load-Acquire Exclusive Byte loads a byte from memory, zero-extends it to form a 32-bit word, writes it to a register and:

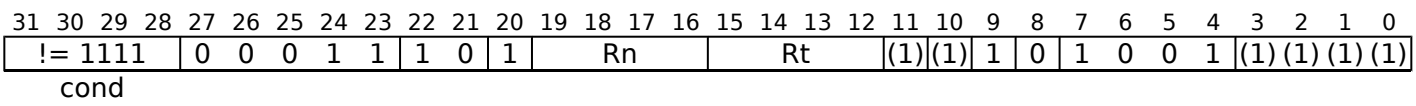
- If the address has the Shared Memory attribute, marks the physical address as exclusive access for the executing PE in a global monitor.
- Causes the executing PE to indicate an active exclusive access in the local monitor.

The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release*.

For more information about support for shared memory see *Synchronization and semaphores*. For information about memory accesses see *Memory accesses*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

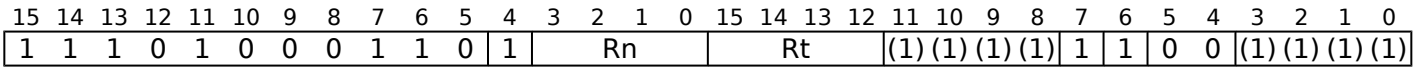
A1



A1

```
LDAEXB{<c>}{<q>} <Rt>, [<Rn>]

t = UInt(Rt);  n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

T1



T1

```
LDAEXB{<c>}{<q>} <Rt>, [<Rn>]

t = UInt(Rt);  n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rt>	Is the general-purpose register to be transferred, encoded in the "Rt" field.
<Rn>	Is the general-purpose base register, encoded in the "Rn" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    AArch32.SetExclusiveMonitors(address, 1);
    R[t] = ZeroExtend(MemO[address, 1], 32);
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDAEXD

Load-Acquire Exclusive Doubleword loads a doubleword from memory, writes it to two registers and:

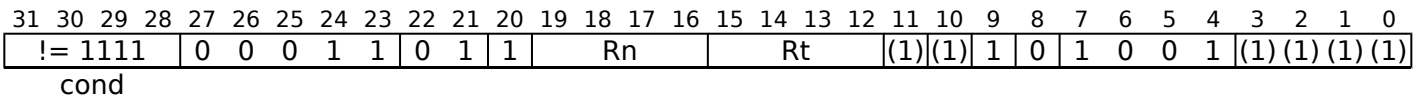
- If the address has the Shared Memory attribute, marks the physical address as exclusive access for the executing PE in a global monitor
- Causes the executing PE to indicate an active exclusive access in the local monitor.

The instruction also acts as a barrier instruction with the ordering requirements described in *Load-Acquire, Store-Release*.

For more information about support for shared memory see *Synchronization and semaphores*. For information about memory accesses see *Memory accesses*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



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

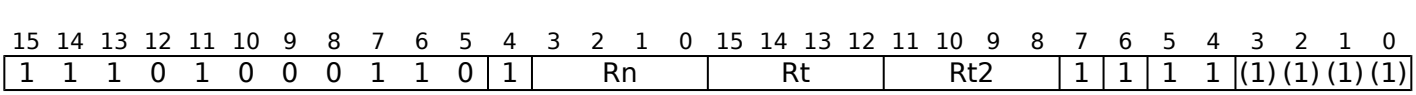
If `Rt<0> == '1'`, then one of the following behaviors must occur:

- 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



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 = MemO[address, 8];
    // Extract words from 64-bit loaded value such that R[t] is
    // loaded from address and R[t2] from address+4.
    R[t]  = if BigEndian(AccType_ORDERED) then value<63:32> else value<31:0>;
    R[t2] = if BigEndian(AccType_ORDERED) then value<31:0>  else value<63:32>;
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDAEXH

Load-Acquire Exclusive Halfword loads a halfword from memory, zero-extends it to form a 32-bit word, writes it to a register and:

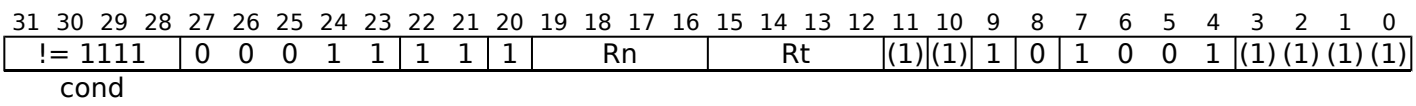
- If the address has the Shared Memory attribute, marks the physical address as exclusive access for the executing PE in a global monitor.
- Causes the executing PE to indicate an active exclusive access in the local monitor.

The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release*.

For more information about support for shared memory see *Synchronization and semaphores*. For information about memory accesses see *Memory accesses*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

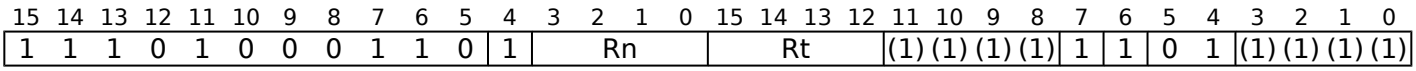
A1



A1

```
LDAEXH{<c>}{<q>} <Rt>, [<Rn>]

t = UInt(Rt);  n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

T1



T1

```
LDAEXH{<c>}{<q>} <Rt>, [<Rn>]

t = UInt(Rt);  n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rt>	Is the general-purpose register to be transferred, encoded in the "Rt" field.
<Rn>	Is the general-purpose base register, encoded in the "Rn" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    AArch32.SetExclusiveMonitors(address, 2);
    R[t] = ZeroExtend(MemO[address, 2], 32);
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

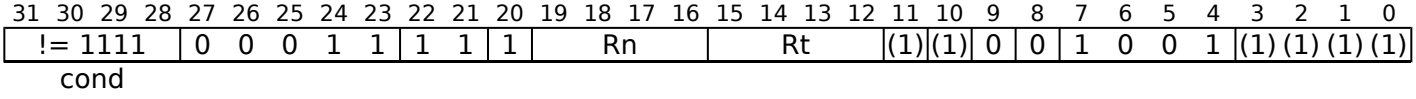
LDAH

Load-Acquire Halfword loads a halfword from memory, zero-extends it to form a 32-bit word and writes it to a register. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release*.

For more information about support for shared memory see *Synchronization and semaphores*. For information about memory accesses see *Memory accesses*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

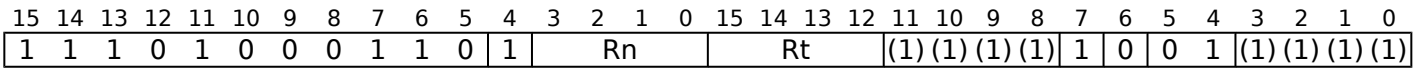
A1



A1

```
LDAH{<c>}{<q>} <Rt>, [<Rn>]

t = UInt(Rt);  n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

T1



T1

```
LDAH{<c>}{<q>} <Rt>, [<Rn>]

t = UInt(Rt);  n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rt>	Is the general-purpose register to be transferred, encoded in the "Rt" field.
<Rn>	Is the general-purpose base register, encoded in the "Rn" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    R[t] = ZeroExtend(MemO[address, 2], 32);
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDC (immediate)

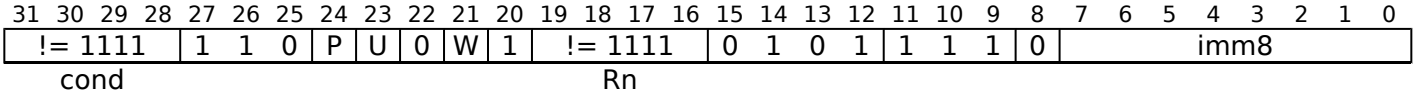
Load data to System register (immediate) calculates an address from a base register value and an immediate offset, loads a word from memory, and writes it to the *DBGDTRTXint* System register. It can use offset, post-indexed, pre-indexed, or unindexed addressing. For information about memory accesses see *Memory accesses*.

In an implementation that includes EL2, the permitted LDC access to *DBGDTRTXint* can be trapped to Hyp mode, meaning that an attempt to execute an LDC instruction in a Non-secure mode other than Hyp mode, that would be permitted in the absence of the Hyp trap controls, generates a Hyp Trap exception. For more information, see *Trapping general Non-secure System register accesses to debug registers*.

For simplicity, the LDC pseudocode does not show this possible trap to Hyp mode.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



Offset (P == 1 && W == 0)

```
LDC{<c>}{<q>} p14, c5, [<Rn>{, #{+/-}<imm>}]
```

Post-indexed (P == 0 && W == 1)

```
LDC{<c>}{<q>} p14, c5, [<Rn>], #{+/-}<imm>
```

Pre-indexed (P == 1 && W == 1)

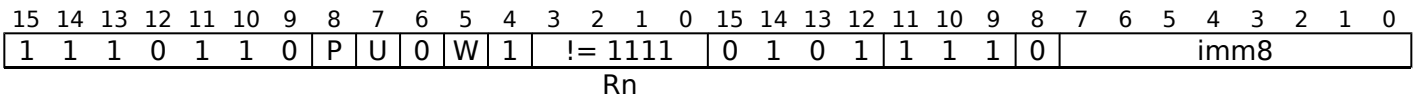
```
LDC{<c>}{<q>} p14, c5, [<Rn>, #{+/-}<imm>]!
```

Unindexed (P == 0 && U == 1 && W == 0)

```
LDC{<c>}{<q>} p14, c5, [<Rn>], <option>
```

```
if Rn == '1111' then SEE "LDC (literal)";
if P == '0' && U == '0' && W == '0' then UNDEFINED;
n = UInt(Rn);  cp = 14;
imm32 = ZeroExtend(imm8:'00', 32);  index = (P == '1');  add = (U == '1');  wback = (W == '1');
```

T1



Offset (P == 1 && W == 0)

```
LDC{<c>}{<q>} p14, c5, [<Rn>{, #{+/-}<imm>}]
```

Post-indexed (P == 0 && W == 1)

```
LDC{<c>}{<q>} p14, c5, [<Rn>], #{+/-}<imm>
```

Pre-indexed (P == 1 && W == 1)

```
LDC{<c>}{<q>} p14, c5, [<Rn>, #{+/-}<imm>]!
```

Unindexed (P == 0 && U == 1 && W == 0)

```
LDC{<c>}{<q>} p14, c5, [<Rn>], <option>
```

```
if Rn == '1111' then SEE "LDC (literal)";
if P == '0' && U == '0' && W == '0' then UNDEFINED;
n = UInt(Rn);  cp = 14;
imm32 = ZeroExtend(imm8:'00', 32);  index = (P == '1');  add = (U == '1');  wback = (W == '1');
```

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rn>	Is the general-purpose base register, encoded in the "Rn" field. If the PC is used, see *LDC (literal)*.
<option>	Is an 8-bit immediate, in the range 0 to 255 enclosed in { }, encoded in the "imm8" field. The value of this field is ignored when executing this instruction.
+/-	Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<imm>	Is the immediate offset used for forming the address, a multiple of 4 in the range 0-1020, defaulting to 0 and encoded in the "imm8" field, as <imm>/4.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];

    // System register write to DBGDTRTXint.
    AArch32.SysRegWriteM(cp, ThisInstr(), address);

    if wback then R[n] = offset_addr;
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDC (literal)

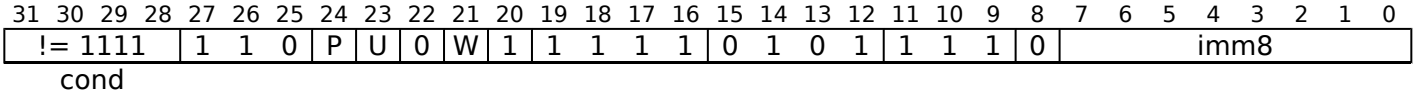
Load data to System register (literal) calculates an address from the PC value and an immediate offset, loads a word from memory, and writes it to the *DBGDTRTXint* System register. For information about memory accesses see *Memory accesses*.

In an implementation that includes EL2, the permitted LDC access to *DBGDTRTXint* can be trapped to Hyp mode, meaning that an attempt to execute an LDC instruction in a Non-secure mode other than Hyp mode, that would be permitted in the absence of the Hyp trap controls, generates a Hyp Trap exception. For more information, see *Trapping general Non-secure System register accesses to debug registers*.

For simplicity, the LDC pseudocode does not show this possible trap to Hyp mode.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



A1 (!(P == 0 && U == 0 && W == 0))

```
LDC{<c>}{<q>} p14, c5, <label>

LDC{<c>}{<q>} p14, c5, [PC, #{+/-}<imm>]

LDC{<c>}{<q>} p14, c5, [PC], <option>
```

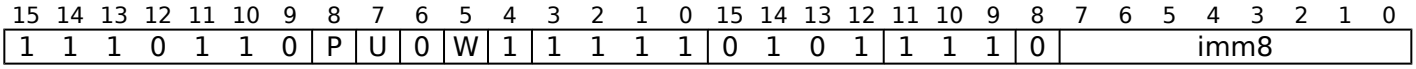
```
if P == '0' && U == '0' && W == '0' then UNDEFINED;
index = (P == '1');  add = (U == '1');  cp = 14;  imm32 = ZeroExtend(imm8:'00', 32);
if W == '1' || (P == '0' && CurrentInstrSet() != InstrSet_A32) then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `W == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes without writeback of the base address.
- The instruction uses the addressing mode described in the equivalent immediate offset instruction.

T1



T1 (!(P == 0 && U == 0 && W == 0))

```
LDC{<c>}{<q>} p14, c5, <label>

LDC{<c>}{<q>} p14, c5, [PC, #{+/-}<imm>]
```

```
if P == '0' && U == '0' && W == '0' then UNDEFINED;
index = (P == '1');  add = (U == '1');  cp = 14;  imm32 = ZeroExtend(imm8:'00', 32);
if W == '1' || (P == '0' && CurrentInstrSet() != InstrSet_A32) then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `W == '1' || P == '0'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes without writeback of the base address.

- The instruction executes as LDC (immediate) with writeback to the PC. The instruction is handled as described in *Using R15*.

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<option> Is an 8-bit immediate, in the range 0 to 255 enclosed in { }, encoded in the "imm8" field. The value of this field is ignored when executing this instruction.

<label> The label of the literal data item that is to be loaded into <Rt>. The assembler calculates the required value of the offset from the Align(PC, 4) value of the instruction to this label. Permitted values of the offset are multiples of 4 in the range -1020 to 1020.

If the offset is zero or positive, imm32 is equal to the offset and add == TRUE (encoded as U == 1).

If the offset is negative, imm32 is equal to minus the offset and add == FALSE (encoded as U == 0).

+/- Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<imm> Is the immediate offset used for forming the address, a multiple of 4 in the range 0-1020, defaulting to 0 and encoded in the "imm8" field, as <imm>/4.

The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see *Use of labels in UAL instruction syntax*.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (Align(PC,4) + imm32) else (Align(PC,4) - imm32);
    address = if index then offset_addr else Align(PC,4);

    // System register write to DBGDTRTXint.
    AArch32.SysRegWriteM(cp, ThisInstr(), address);
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

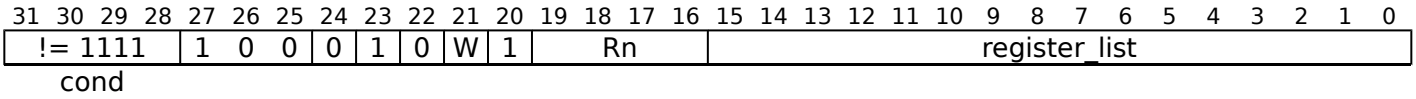
LDM, LDMIA, LDMFD

Load Multiple (Increment After, Full Descending) loads multiple registers from consecutive memory locations using an address from a base register. The consecutive memory locations start at this address, and the address just above the highest of those locations can optionally be written back to the base register.
The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. See also *Encoding of lists of general-purpose registers and the PC*.
Armv8.2 permits the deprecation of some Load Multiple ordering behaviors in AArch32 state, for more information see *FEAT_LSMAOC*. The registers loaded can include the PC, causing a branch to a loaded address. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*. Related system instructions are *LDM (User registers)* and *LDM (exception return)*.

This instruction is used by the alias POP (multiple registers).

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2) .

A1



A1

```
LDM{IA}{<c>}{<q>} <Rn>{!}, <registers> // (Preferred syntax)

LDMFD{<c>}{<q>} <Rn>{!}, <registers> // (Alternate syntax, Full Descending stack)
```

```
n = UInt(Rn);  registers = register_list;  wback = (W == '1');
if n == 15 || BitCount(registers) < 1 then UNPREDICTABLE;
if wback && registers<n> == '1' then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

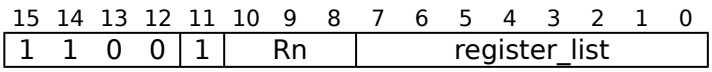
If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes as `LDM` with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers loaded.

If `wback && registers<n> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1



T1

```
LDM{IA}{<c>}{<q>} <Rn>{!}, <registers> // (Preferred syntax)

LDMFD{<c>}{<q>} <Rn>{!}, <registers> // (Alternate syntax, Full Descending stack)
```

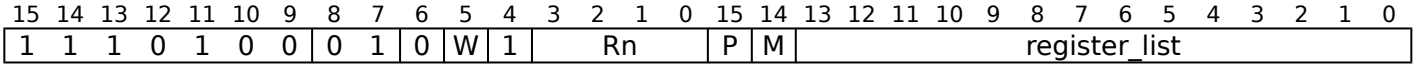
```
n = UInt(Rn);  registers = '00000000':register_list;  wback = (registers<n> == '0');
if BitCount(registers) < 1 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes as `LDM` with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers loaded.

T2



T2

```
LDM{IA}{<c>}.W <Rn>{!}, <registers> // (Preferred syntax, if <Rn>, '!' and <registers> can be represented

LDMFD{<c>}.W <Rn>{!}, <registers> // (Alternate syntax, Full Descending stack, if <Rn>, '!' and <register

LDM{IA}{<c>}{<q>} <Rn>{!}, <registers> // (Preferred syntax)

LDMFD{<c>}{<q>} <Rn>{!}, <registers> // (Alternate syntax, Full Descending stack)
```

```
n = UInt(Rn);  registers = P:M:register_list;  wback = (W == '1');
if n == 15 || BitCount(registers) < 2 || (P == '1' && M == '1') then UNPREDICTABLE;
if wback && registers<n> == '1' then UNPREDICTABLE;
if registers<13> == '1' then UNPREDICTABLE;
if registers<15> == '1' && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes as `LDM` with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers loaded.

If `wback && registers<n> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

If `BitCount(registers) == 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction loads a single register using the specified addressing modes.
- The instruction executes as `LDM` with the same addressing mode but targeting an unspecified set of registers. These registers might include R15.

If `registers<13> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs all of the loads using the specified addressing mode, but R13 is UNKNOWN.

If `P == '1' && M == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction loads the register list and either R14 or R15, both R14 and R15, or neither of these registers.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

IA	Is an optional suffix for the Increment After form.
<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rn>	Is the general-purpose base register, encoded in the "Rn" field.
!	For encoding A1 and T2: the address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the "W" field as 1, otherwise this field defaults to 0. For encoding T1: the address adjusted by the size of the data loaded is written back to the base register. It is omitted if <Rn> is included in <registers>, otherwise it must be present.
<registers>	For encoding A1: is a list of one or more registers to be loaded, separated by commas and surrounded by { and }. The PC can be in the list. Arm deprecates using these instructions with both the LR and the PC in the list. For encoding T1: is a list of one or more registers to be loaded, separated by commas and surrounded by { and }. The registers in the list must be in the range R0-R7, encoded in the "register_list" field. For encoding T2: is a list of one or more registers to be loaded, separated by commas and surrounded by { and }. The registers in the list must be in the range R0-R12, encoded in the "register_list" field, and can optionally contain one of the LR or the PC. If the LR is in the list, the "M" field is set to 1, otherwise it defaults to 0. If the PC is in the list, the "P" field is set to 1, otherwise it defaults to 0. If the PC is in the list: • The LR must not be in the list. • The instruction must be either outside any IT block, or the last instruction in an IT block.

Alias Conditions

Alias	Of variant	Is preferred when
POP (multiple registers)	T2	`W == '1' && Rn == '1101' && BitCount(P:M:register_list) > 1`
POP (multiple registers)	A1	`W == '1' && Rn == '1101' && BitCount(register_list) > 1`

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    for i = 0 to 14
        if registers<i> == '1' then
            R[i] = MemS[address,4];  address = address + 4;
    if registers<15> == '1' then
        LoadWritePC(MemS[address,4]);
    if wback && registers<n> == '0' then R[n] = R[n] + 4*BitCount(registers);
    if wback && registers<n> == '1' then R[n] = bits(32) UNKNOWN;
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDM (exception return)

Load Multiple (exception return) loads multiple registers from consecutive memory locations using an address from a base register. The *SPSR* of the current mode is copied to the *CPSR*. An address adjusted by the size of the data loaded can optionally be written back to the base register.

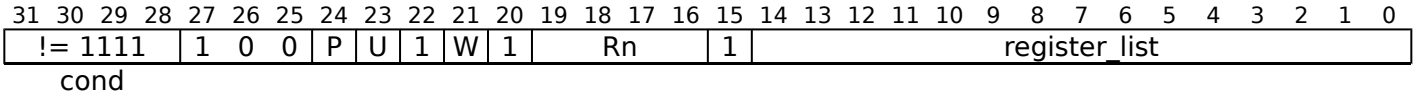
The registers loaded include the PC. The word loaded for the PC is treated as an address and a branch occurs to that address.

The PE checks the encoding that is copied to the *CPSR* for an illegal return event. See *Illegal return events from AArch32 state*.

Load Multiple (exception return) is:

- UNDEFINED in Hyp mode.
- UNPREDICTABLE in debug state, and in User mode and System mode.

A1



A1

```
LDM{<amode>}{<c>}{<q>} <Rn>{!}, <registers_with_pc>^
```

```
n = UInt(Rn);  registers = register_list;
wback = (W == '1');  increment = (U == '1');  wordhigher = (P == U);
if n == 15 then UNPREDICTABLE;
if wback && registers<n> == '1' then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `wback && registers<n> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs all the loads using the specified addressing mode and the content of the register being written back is UNKNOWN. In addition, if an exception occurs during the execution of this instruction, the base address might be corrupted so that the instruction cannot be repeated.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<amode> is one of:

DA
Decrement After. The consecutive memory addresses end at the address in the base register. Encoded as P = 0, U = 0.

FA
Full Ascending. For this instruction, a synonym for DA.

DB
Decrement Before. The consecutive memory addresses end one word below the address in the base register. Encoded as P = 1, U = 0.

EA
Empty Ascending. For this instruction, a synonym for DB.

IA
Increment After. The consecutive memory addresses start at the address in the base register. This is the default. Encoded as P = 0, U = 1.

FD
Full Descending. For this instruction, a synonym for IA.

IB
Increment Before. The consecutive memory addresses start one word above the address in the base register. Encoded as P = 1, U = 1.

ED
Empty Descending. For this instruction, a synonym for IB.

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

! The address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the "W" field as 1, otherwise this field defaults to 0.

<registers_with_pc> Is a list of one or more registers, separated by commas and surrounded by { and }. It specifies the set of registers to be loaded. The registers are loaded with the lowest-numbered register from the lowest memory address, through to the highest-numbered register from the highest memory address. The PC must be specified in the register list, and the instruction causes a branch to the address (data) loaded into the PC. See also *Encoding of lists of general-purpose registers and the PC*.

Instructions with similar syntax but without the PC included in the registers list are described in *LDM (User registers)*.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if PSTATE.EL == EL2 then
        UNDEFINED;
    elsif PSTATE.M IN {M32_User,M32_System} then
        UNPREDICTABLE;                        // UNDEFINED or NOP
    else
        length = 4*BitCount(registers) + 4;
        address = if increment then R[n] else R[n]-length;
        if wordhigher then address = address+4;

        for i = 0 to 14
            if registers<i> == '1' then
                R[i] = MemS[address,4];  address = address + 4;
        new_pc_value = MemS[address,4];

        if wback && registers<n> == '0' then R[n] = if increment then R[n]+length else R[n]-length;
        if wback && registers<n> == '1' then R[n] = bits(32) UNKNOWN;

        AArch32.ExceptionReturn(new_pc_value, SPSR[]);
```

CONSTRAINED UNPREDICTABLE behavior

If `PSTATE.M IN {M32_User,M32_System}`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

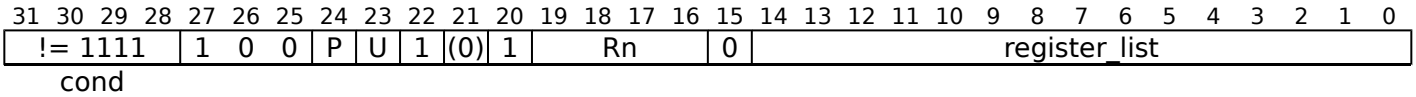
LDM (User registers)

In an EL1 mode other than System mode, Load Multiple (User registers) loads multiple User mode registers from consecutive memory locations using an address from a base register. The registers loaded cannot include the PC. The PE reads the base register value normally, using the current mode to determine the correct Banked version of the register. This instruction cannot writeback to the base register.

Load Multiple (User registers) is UNDEFINED in Hyp mode, and UNPREDICTABLE in User and System modes.

Armv8.2 permits the deprecation of some Load Multiple ordering behaviors in AArch32 state, for more information see *FEAT_LSMAOC*.

A1



A1

```
LDM{<amode>}{<c>}{<q>} <Rn>, <registers_without_pc>^
```

```
n = UInt(Rn);  registers = register_list;  increment = (U == '1');  wordhigher = (P == U);
if n == 15 || BitCount(registers) < 1 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction operates as an `LDM` with the same addressing mode but targeting an unspecified set of registers. These registers might include R15.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<amode> is one of:

DA
Decrement After. The consecutive memory addresses end at the address in the base register. Encoded as P = 0, U = 0.

FA
Full Ascending. For this instruction, a synonym for DA.

DB
Decrement Before. The consecutive memory addresses end one word below the address in the base register. Encoded as P = 1, U = 0.

EA
Empty Ascending. For this instruction, a synonym for DB.

IA
Increment After. The consecutive memory addresses start at the address in the base register. This is the default. Encoded as P = 0, U = 1.

FD
Full Descending. For this instruction, a synonym for IA.

IB
Increment Before. The consecutive memory addresses start one word above the address in the base register. Encoded as P = 1, U = 1.

ED
Empty Descending. For this instruction, a synonym for IB.

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

<registers_without_pc> Is a list of one or more registers, separated by commas and surrounded by { and }. It specifies the set of registers to be loaded by the LDM instruction. The registers are loaded with the lowest-numbered register from the lowest memory address, through to the highest-numbered register from the highest memory address. The PC must not be in the register list. See also *Encoding of lists of general-purpose registers and the PC*.

Instructions with similar syntax but with the PC included in <registers_without_pc> are described in *LDM (exception return)*.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if PSTATE.EL == EL2 then UNDEFINED;
    elsif PSTATE.M IN {M32_User,M32_System} then UNPREDICTABLE;
    else
        length = 4*BitCount(registers);
        address = if increment then R[n] else R[n]-length;
        if wordhigher then address = address+4;
        for i = 0 to 14
            if registers<i> == '1' then  // Load User mode register
                Rmode[i, M32_User] = MemS[address,4];  address = address + 4;
```

CONSTRAINED UNPREDICTABLE behavior

If PSTATE.M IN {M32_User,M32_System}, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as an LDM with the same addressing mode but targeting an unspecified set of registers. These registers might include R15.

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDMDA, LDMFA

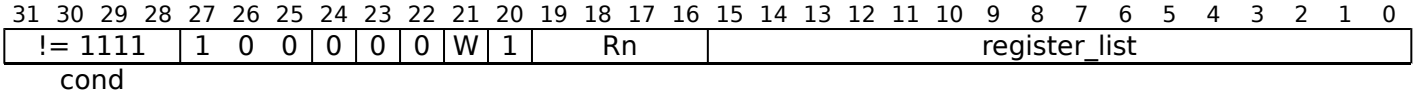
Load Multiple Decrement After (Full Ascending) loads multiple registers from consecutive memory locations using an address from a base register. The consecutive memory locations end at this address, and the address just below the lowest of those locations can optionally be written back to the base register.

The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. See also *Encoding of lists of general-purpose registers and the PC*.

Armv8.2 permits the deprecation of some Load Multiple ordering behaviors in AArch32 state, for more information see *FEAT_LSMAOC*. The registers loaded can include the PC, causing a branch to a loaded address. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*.

Related system instructions are *LDM (User registers)* and *LDM (exception return)*.

A1



A1

```
LDMDA{<c>}{<q>} <Rn>{!}, <registers> // (Preferred syntax)

LDMFA{<c>}{<q>} <Rn>{!}, <registers> // (Alternate syntax, Full Ascending stack)
```

```
n = UInt(Rn);  registers = register_list;  wback = (W == '1');
if n == 15 || BitCount(registers) < 1 then UNPREDICTABLE;
if wback && registers<n> == '1' then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction operates as an `LDM` with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers loaded.

If `wback && registers<n> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rn>	Is the general-purpose base register, encoded in the "Rn" field.
!	The address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the "W" field as 1, otherwise this field defaults to 0.
<registers>	Is a list of one or more registers to be loaded, separated by commas and surrounded by { and }. The PC can be in the list. Arm deprecates using these instructions with both the LR and the PC in the list.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n] - 4*BitCount(registers) + 4;
    for i = 0 to 14
        if registers<i> == '1' then
            R[i] = MemS[address,4];  address = address + 4;
    if registers<15> == '1' then
        LoadWritePC(MemS[address,4]);
    if wback && registers<n> == '0' then R[n] = R[n] - 4*BitCount(registers);
    if wback && registers<n> == '1' then R[n] = bits(32) UNKNOWN;
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDMDB, LDMEA

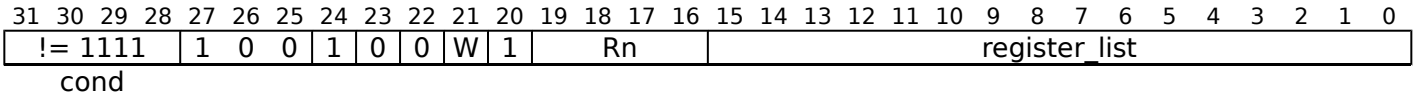
Load Multiple Decrement Before (Empty Ascending) loads multiple registers from consecutive memory locations using an address from a base register. The consecutive memory locations end just below this address, and the address of the lowest of those locations can optionally be written back to the base register.

The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. See also *Encoding of lists of general-purpose registers and the PC*.

Armv8.2 permits the deprecation of some Load Multiple ordering behaviors in AArch32 state, for more information see *FEAT_LSMAOC*. The registers loaded can include the PC, causing a branch to a loaded address. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*. Related system instructions are *LDM (User registers)* and *LDM (exception return)*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



A1

```
LDMDB{<c>}{<q>} <Rn>{!}, <registers> // (Preferred syntax)

LDMEA{<c>}{<q>} <Rn>{!}, <registers> // (Alternate syntax, Empty Ascending stack)

n = UInt(Rn);  registers = register_list;  wback = (W == '1');
if n == 15 || BitCount(registers) < 1 then UNPREDICTABLE;
if wback && registers<n> == '1' then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

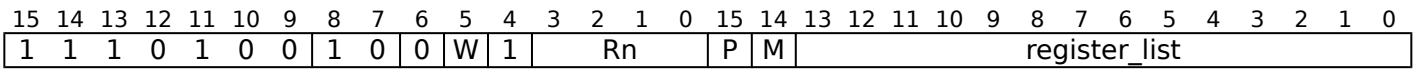
If `wback && registers<n> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes as `LDM` with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers loaded.

T1



T1

```
LDMDB{<c>}{<q>} <Rn>{!}, <registers> // (Preferred syntax)

LDMEA{<c>}{<q>} <Rn>{!}, <registers> // (Alternate syntax, Empty Ascending stack)
```

```
n = UInt(Rn);  registers = P:M:register_list;  wback = (W == '1');
if n == 15 || BitCount(registers) < 2 || (P == '1' && M == '1') then UNPREDICTABLE;
if wback && registers<n> == '1' then UNPREDICTABLE;
if registers<13> == '1' then UNPREDICTABLE;
if registers<15> == '1' && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `wback && registers<n> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes as `LDM` with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers loaded.

If `BitCount(registers) == 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction loads a single register using the specified addressing modes.
- The instruction executes as `LDM` with the same addressing mode but targeting an unspecified set of registers. These registers might include R15.

If `registers<13> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs all of the loads using the specified addressing mode, but R13 is UNKNOWN.

If `P == '1' && M == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction loads the register list and either R14 or R15, both R14 and R15, or neither of these registers.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rn>	Is the general-purpose base register, encoded in the "Rn" field.
!	The address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the "W" field as 1, otherwise this field defaults to 0.
<registers>	For encoding A1: is a list of one or more registers to be loaded, separated by commas and surrounded by { and }. The PC can be in the list. Arm deprecates using these instructions with both the LR and the PC in the list. For encoding T1: is a list of one or more registers to be loaded, separated by commas and surrounded by { and }. The registers in the list must be in the range R0-R12, encoded in the "register_list" field,

and can optionally contain one of the LR or the PC. If the LR is in the list, the "M" field is set to 1, otherwise it defaults to 0. If the PC is in the list, the "P" field is set to 1, otherwise it defaults to 0.

If the PC is in the list:

- The LR must not be in the list.
- The instruction must be either outside any IT block, or the last instruction in an IT block.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n] - 4*BitCount(registers);
    for i = 0 to 14
        if registers<i> == '1' then
            R[i] = MemS[address,4];  address = address + 4;
    if registers<15> == '1' then
        LoadWritePC(MemS[address,4]);
    if wback && registers<n> == '0' then R[n] = R[n] - 4*BitCount(registers);
    if wback && registers<n> == '1' then R[n] = bits(32) UNKNOWN;
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDMIB, LDMED

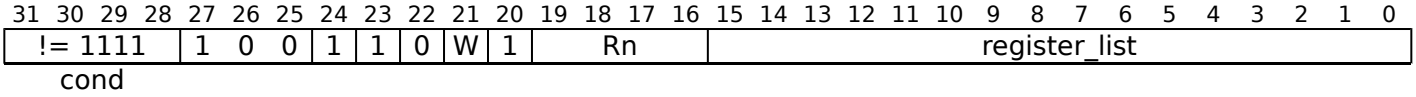
Load Multiple Increment Before (Empty Descending) loads multiple registers from consecutive memory locations using an address from a base register. The consecutive memory locations start just above this address, and the address of the last of those locations can optionally be written back to the base register.

The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. See also *Encoding of lists of general-purpose registers and the PC*.

Armv8.2 permits the deprecation of some Load Multiple ordering behaviors in AArch32 state, for more information see *FEAT_LSMAOC*. The registers loaded can include the PC, causing a branch to a loaded address. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*.

Related system instructions are *LDM (User registers)* and *LDM (exception return)*.

A1



A1

```
LDMIB{<c>}{<q>} <Rn>{!}, <registers> // (Preferred syntax)

LDMED{<c>}{<q>} <Rn>{!}, <registers> // (Alternate syntax, Empty Descending stack)
```

```
n = UInt(Rn);  registers = register_list;  wback = (W == '1');
if n == 15 || BitCount(registers) < 1 then UNPREDICTABLE;
if wback && registers<n> == '1' then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction operates as an `LDM` with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers loaded.

If `wback && registers<n> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rn>	Is the general-purpose base register, encoded in the "Rn" field.
!	The address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the "W" field as 1, otherwise this field defaults to 0.
<registers>	Is a list of one or more registers to be loaded, separated by commas and surrounded by { and }. The PC can be in the list. Arm deprecates using these instructions with both the LR and the PC in the list.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n] + 4;
    for i = 0 to 14
        if registers<i> == '1' then
            R[i] = MemS[address,4];  address = address + 4;
    if registers<15> == '1' then
        LoadWritePC(MemS[address,4]);
    if wback && registers<n> == '0' then R[n] = R[n] + 4*BitCount(registers);
    if wback && registers<n> == '1' then R[n] = bits(32) UNKNOWN;
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

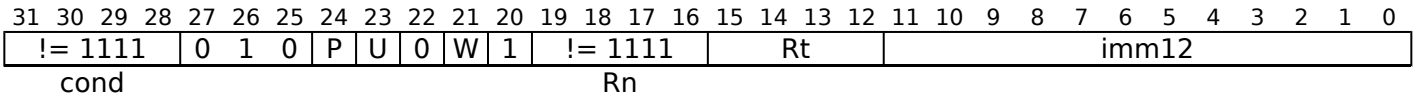
LDR (immediate)

Load Register (immediate) calculates an address from a base register value and an immediate offset, loads a word from memory, and writes it to a register. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see *Memory accesses*.

This instruction is used by the alias POP (single register).

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 , T2 , T3 and T4) .

A1



Offset (P == 1 && W == 0)

```
LDR{<c>}{<q>} <Rt>, [<Rn> {, #{+/-}<imm>}]
```

Post-indexed (P == 0 && W == 0)

```
LDR{<c>}{<q>} <Rt>, [<Rn>], #{+/-}<imm>
```

Pre-indexed (P == 1 && W == 1)

```
LDR{<c>}{<q>} <Rt>, [<Rn>, #{+/-}<imm>]!
```

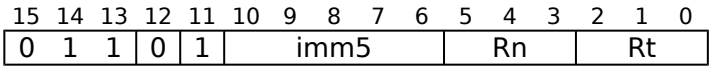
```
if Rn == '1111' then SEE "LDR (literal)";
if P == '0' && W == '1' then SEE "LDRT";
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
if wback && n == t then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1

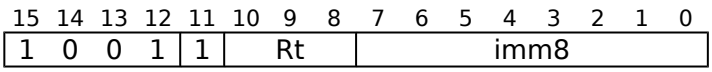


T1

```
LDR{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```

```
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm5:'00', 32);
index = TRUE;  add = TRUE;  wback = FALSE;
```

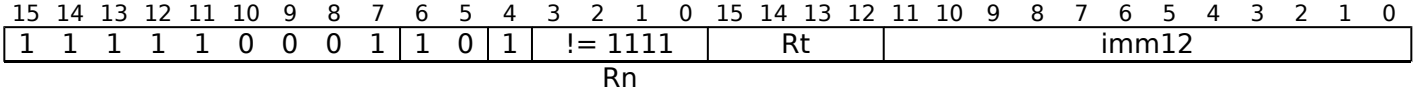
T2



T2

```
LDR{<c>}{<q>} <Rt>, [SP{, #{+}<imm>}]

t = UInt(Rt);  n = 13;  imm32 = ZeroExtend(imm8:'00', 32);
index = TRUE;  add = TRUE;  wback = FALSE;
```

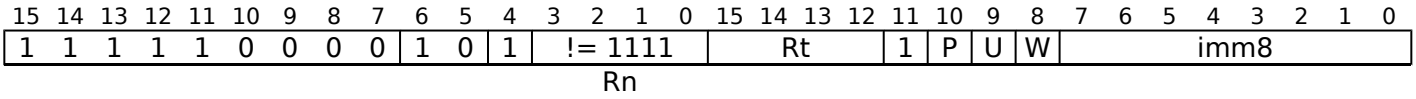
T3



T3

```
LDR{<c>}.W <Rt>, [<Rn> {, #{+}<imm>}] // (<Rt>, <Rn>, <imm> can be represented in T1 or T2)

LDR{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]

if Rn == '1111' then SEE "LDR (literal)";
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32); index = TRUE;  add = TRUE;
wback = FALSE; if t == 15 && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

T4



Offset (P == 1 && U == 0 && W == 0)

```
LDR{<c>}{<q>} <Rt>, [<Rn> {, #-<imm>}]
```

Post-indexed (P == 0 && W == 1)

```
LDR{<c>}{<q>} <Rt>, [<Rn>], #{+/-}<imm>
```

Pre-indexed (P == 1 && W == 1)

```
LDR{<c>}{<q>} <Rt>, [<Rn>, #{+/-}<imm>]!

if Rn == '1111' then SEE "LDR (literal)";
if P == '1' && U == '1' && W == '0' then SEE "LDRT";
if P == '0' && W == '0' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);
imm32 = ZeroExtend(imm8, 32); index = (P == '1');  add = (U == '1');  wback = (W == '1');
if (wback && n == t) || (t == 15 && InITBlock() && !LastInITBlock()) then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<Rt> For encoding A1: is the general-purpose register to be transferred, encoded in the "Rt" field. The PC can be used. If the PC is used, the instruction branches to the address (data) loaded to the PC. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*.

For encoding T1 and T2: is the general-purpose register to be transferred, encoded in the "Rt" field.

For encoding T3 and T4: is the general-purpose register to be transferred, encoded in the "Rt" field. The PC can be used, provided the instruction is either outside an IT block or the last instruction of an IT block. If the PC is used, the instruction branches to the address (data) loaded to the PC. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*.

<Rn> For encoding A1, T3 and T4: is the general-purpose base register, encoded in the "Rn" field. For PC use see *LDR (literal)*.

For encoding T1: is the general-purpose base register, encoded in the "Rn" field.

+/- Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

\+ Specifies the offset is added to the base register.

<imm> For encoding A1: is the 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 if omitted, and encoded in the "imm12" field.

For encoding T1: is the optional positive unsigned immediate byte offset, a multiple of 4, in the range 0 to 124, defaulting to 0 and encoded in the "imm5" field as <imm>/4.

For encoding T2: is the optional positive unsigned immediate byte offset, a multiple of 4, in the range 0 to 1020, defaulting to 0 and encoded in the "imm8" field as <imm>/4.

For encoding T3: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

For encoding T4: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm8" field.

Alias Conditions

Alias	Of variant	Is preferred when
POP (single register)	A1 (post-indexed)	`P == '0' && U == '1' && W == '0' && Rn == '1101' && imm12 == '000000000100'`
POP (single register)	T4 (post-indexed)	`Rn == '1101' && P == '0' && U == '1' && W == '1' && imm8 == '00000100'`

Operation

```
if CurrentInstrSet() == InstrSet_A32 then
    if ConditionPassed() then
        EncodingSpecificOperations();
        offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
        address = if index then offset_addr else R[n];
        data = MemU[address,4];
        if wback then R[n] = offset_addr;
        if t == 15 then
            if address<1:0> == '00' then
                LoadWritePC(data);
            else
                UNPREDICTABLE;
        else
            R[t] = data;
else
    if ConditionPassed() then
        EncodingSpecificOperations();
        offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
        address = if index then offset_addr else R[n];
        data = MemU[address,4];
        if wback then R[n] = offset_addr;
        if t == 15 then
            if address<1:0> == '00' then
                LoadWritePC(data);
            else
                UNPREDICTABLE;
        else
            R[t] = data;
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

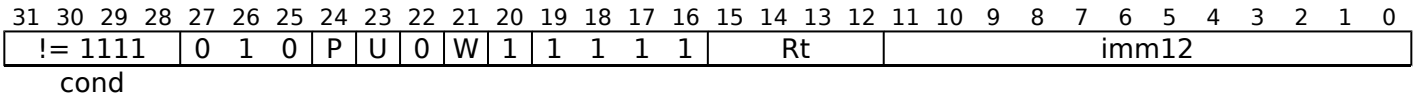
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDR (literal)

Load Register (literal) calculates an address from the PC value and an immediate offset, loads a word from memory, and writes it to a register. For information about memory accesses see *Memory accesses*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2) .

A1



A1 (!(P == 0 && W == 1))

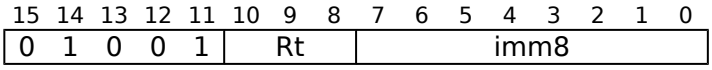
```
LDR{<c>}{<q>} <Rt>, <label> // (Normal form)

LDR{<c>}{<q>} <Rt>, [PC, #{+/-}<imm>] // (Alternative form)

if P == '0' && W == '1' then SEE "LDRT";
t = UInt(Rt);  imm32 = ZeroExtend(imm12, 32);
add = (U == '1');  wback = (P == '0') || (W == '1');
if wback then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `wback`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: wback = FALSE;.
- The instruction treats bit[24] as the P bit, and bit[21] as the writeback (W) bit, and uses the same addressing mode as described in *LDR (immediate)*. The instruction uses post-indexed addressing when P == '0' and uses pre-indexed addressing otherwise. The instruction is handled as described in *Using R15*.

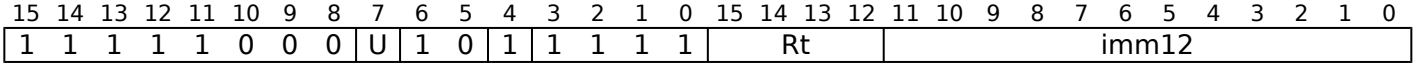
T1



T1

```
LDR{<c>}{<q>} <Rt>, <label> // (Normal form)

t = UInt(Rt);  imm32 = ZeroExtend(imm8:'00', 32);  add = TRUE;
```

T2



T2

```
LDR{<c>}.W <Rt>, <label> // (Preferred syntax, and <Rt>, <label> can be represented in T1)

LDR{<c>}{<q>} <Rt>, <label> // (Preferred syntax)

LDR{<c>}{<q>} <Rt>, [PC, #{+/-}<imm>] // (Alternative syntax)

t = UInt(Rt);  imm32 = ZeroExtend(imm12, 32);  add = (U == '1');
if t == 15 && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<Rt> For encoding A1: is the general-purpose register to be transferred, encoded in the "Rt" field. The PC can be used. If the PC is used, the instruction branches to the address (data) loaded to the PC. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*.

For encoding T1: is the general-purpose register to be transferred, encoded in the "Rt" field.

For encoding T2: is the general-purpose register to be transferred, encoded in the "Rt" field. The SP can be used. The PC can be used, provided the instruction is either outside an IT block or the last instruction of an IT block. If the PC is used, the instruction branches to the address (data) loaded to the PC. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*.

<label> For encoding A1 and T2: the label of the literal data item that is to be loaded into <Rt>. The assembler calculates the required value of the offset from the Align(PC, 4) value of the instruction to this label. Permitted values of the offset are -4095 to 4095.

If the offset is zero or positive, imm32 is equal to the offset and add == TRUE, encoded as U == 1.

If the offset is negative, imm32 is equal to minus the offset and add == FALSE, encoded as U == 0.

For encoding T1: the label of the literal data item that is to be loaded into <Rt>. The assembler calculates the required value of the offset from the Align(PC, 4) value of the instruction to this label. Permitted values of the offset are Multiples of four in the range 0 to 1020.

+/- Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<imm> For encoding A1: is the 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 if omitted, and encoded in the "imm12" field.

For encoding T2: is a 12-bit unsigned immediate byte offset, in the range 0 to 4095, encoded in the "imm12" field.

The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see *Use of labels in UAL instruction syntax*.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    base = Align(PC,4);
    address = if add then (base + imm32) else (base - imm32);
    data = MemU[address,4];
    if t == 15 then
        if address<1:0> == '00' then
            LoadWritePC(data);
        else
            UNPREDICTABLE;
    else
        R[t] = data;
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

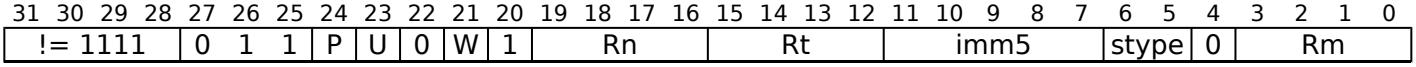
LDR (register)

Load Register (register) calculates an address from a base register value and an offset register value, loads a word from memory, and writes it to a register. The offset register value can optionally be shifted. For information about memory accesses, see *Memory accesses*.

The T32 form of LDR (register) does not support register writeback.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2) .

A1



Offset (P == 1 && W == 0)

```
LDR{<c>}{<q>} <Rt>, [<Rn>, {+/-}<Rm>{, <shift>}]
```

Post-indexed (P == 0 && W == 0)

```
LDR{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>{, <shift>}
```

Pre-indexed (P == 1 && W == 1)

```
LDR{<c>}{<q>} <Rt>, [<Rn>, {+/-}<Rm>{, <shift>}]!
```

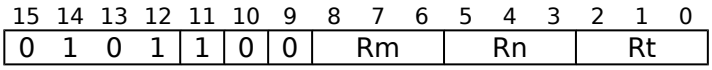
```
if P == '0' && W == '1' then SEE "LDRT";
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm5);
if m == 15 then UNPREDICTABLE;
if wback && (n == 15 || n == t) then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1

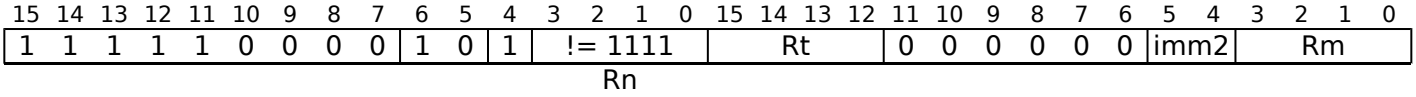


T1

```
LDR{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>]
```

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
(shift_t, shift_n) = (SRType_LSL, 0);
```

T2



T2

```
LDR{<c>}.W <Rt>, [<Rn>, {+}<Rm>] // (<Rt>, <Rn>, <Rm> can be represented in T1)

LDR{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>{, LSL #<imm>}]
```

```
if Rn == '1111' then SEE "LDR (literal)";
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
(shift_t, shift_n) = (SRType_LSL, UInt(imm2));
if m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
if t == 15 && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<Rt> For encoding A1: is the general-purpose register to be transferred, encoded in the "Rt" field. The PC can be used. If the PC is used, the instruction branches to the address (data) loaded to the PC. This branch is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*.

For encoding T1: is the general-purpose register to be transferred, encoded in the "Rt" field.

For encoding T2: is the general-purpose register to be transferred, encoded in the "Rt" field. The PC can be used, provided the instruction is either outside an IT block or the last instruction of an IT block. If the PC is used, the instruction branches to the address (data) loaded to the PC. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*.

<Rn> For encoding A1: is the general-purpose base register, encoded in the "Rn" field. The PC can be used in the offset variant.

For encoding T1 and T2: is the general-purpose base register, encoded in the "Rn" field.

+/- Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

\+ Specifies the index register is added to the base register.

<Rm> Is the general-purpose index register, encoded in the "Rm" field.

<shift> The shift to apply to the value read from <Rm>. If absent, no shift is applied. Otherwise, see *Shifts applied to a register*.

<imm> If present, the size of the left shift to apply to the value from <Rm>, in the range 1-3. <imm> is encoded in imm2. If absent, no shift is specified and imm2 is encoded as 0b00.

Operation

```
if CurrentInstrSet() == InstrSet_A32 then
    if ConditionPassed() then
        EncodingSpecificOperations();
        offset = Shift(R[m], shift_t, shift_n, PSTATE.C);
        offset_addr = if add then (R[n] + offset) else (R[n] - offset);
        address = if index then offset_addr else R[n];
        data = MemU[address,4];
        if wback then R[n] = offset_addr;
        if t == 15 then
            if address<1:0> == '00' then
                LoadWritePC(data);
            else
                UNPREDICTABLE;
        else
            R[t] = data;
else
    if ConditionPassed() then
        EncodingSpecificOperations();
        offset = Shift(R[m], shift_t, shift_n, PSTATE.C);
        offset_addr = (R[n] + offset);
        address = offset_addr;
        data = MemU[address,4];
        if t == 15 then
            if address<1:0> == '00' then
                LoadWritePC(data);
            else
                UNPREDICTABLE;
        else
            R[t] = data;
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

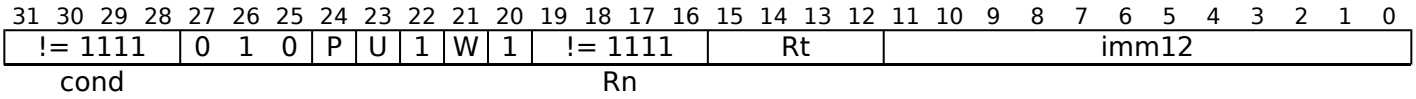
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDRB (immediate)

Load Register Byte (immediate) calculates an address from a base register value and an immediate offset, loads a byte from memory, zero-extends it to form a 32-bit word, and writes it to a register. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see *Memory accesses*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 , T2 and T3) .

A1



Offset (P == 1 && W == 0)

```
LDRB{<c>}{<q>} <Rt>, [<Rn> {, #{+/-}<imm>}]
```

Post-indexed (P == 0 && W == 0)

```
LDRB{<c>}{<q>} <Rt>, [<Rn>], #{+/-}<imm>
```

Pre-indexed (P == 1 && W == 1)

```
LDRB{<c>}{<q>} <Rt>, [<Rn>, #{+/-}<imm>]!
```

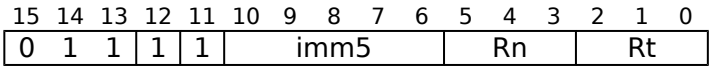
```
if Rn == '1111' then SEE "LDRB (literal)";
if P == '0' && W == '1' then SEE "LDRBT";
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
if t == 15 || (wback && n == t) then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1

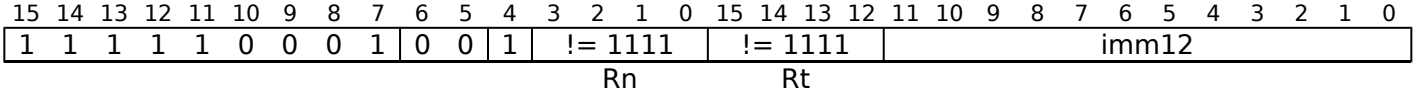


T1

```
LDRB{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```

```
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm5, 32);
index = TRUE;  add = TRUE;  wback = FALSE;
```

T2

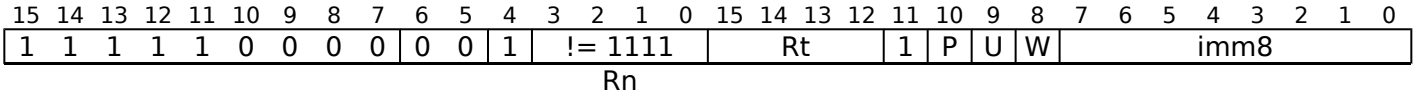


T2

```
LDRB{<c>}.W <Rt>, [<Rn> {, #{+}<imm>}] // (<Rt>, <Rn>, <imm> can be represented in T1)

LDRB{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```

```
if Rt == '1111' then SEE "PLD";
if Rn == '1111' then SEE "LDRB (literal)";
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
index = TRUE;  add = TRUE;  wback = FALSE;
// Armv8-A removes UNPREDICTABLE for R13
```

T3



Offset (Rt != 1111 && P == 1 && U == 0 && W == 0)

```
LDRB{<c>}{<q>} <Rt>, [<Rn> {, #-<imm>}]
```

Post-indexed (P == 0 && W == 1)

```
LDRB{<c>}{<q>} <Rt>, [<Rn>], #{+/-}<imm>
```

Pre-indexed (P == 1 && W == 1)

```
LDRB{<c>}{<q>} <Rt>, [<Rn>, #{+/-}<imm>]!
```

```
if Rt == '1111' && P == '1' && U == '0' && W == '0' then SEE "PLD, PLDW (immediate)";
if Rn == '1111' then SEE "LDRB (literal)";
if P == '1' && U == '1' && W == '0' then SEE "LDRBT";
if P == '0' && W == '0' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
if  (t == 15 &&  W == '1') || (wback && n == t) then UNPREDICTABLE;
// Armv8-A removes UNPREDICTABLE for R13
```

CONSTRAINED UNPREDICTABLE behavior

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rt>	Is the general-purpose register to be transferred, encoded in the "Rt" field.
<Rn>	For encoding A1, T2 and T3: is the general-purpose base register, encoded in the "Rn" field. For PC use see *LDRB (literal)*.
	For encoding T1: is the general-purpose base register, encoded in the "Rn" field.

+/- Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

\+ Specifies the offset is added to the base register.

<imm> For encoding A1: is the 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 if omitted, and encoded in the "imm12" field.

For encoding T1: is an optional 5-bit unsigned immediate byte offset, in the range 0 to 31, defaulting to 0 and encoded in the "imm5" field.

For encoding T2: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

For encoding T3: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm8" field.

Operation

```
if CurrentInstrSet() == InstrSet_A32 then
    if ConditionPassed() then
        EncodingSpecificOperations();
        offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
        address = if index then offset_addr else R[n];
        R[t] = ZeroExtend(MemU[address,1], 32);
        if wback then R[n] = offset_addr;
else
    if ConditionPassed() then
        EncodingSpecificOperations();
        offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
        address = if index then offset_addr else R[n];
        R[t] = ZeroExtend(MemU[address,1], 32);
        if wback then R[n] = offset_addr;
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

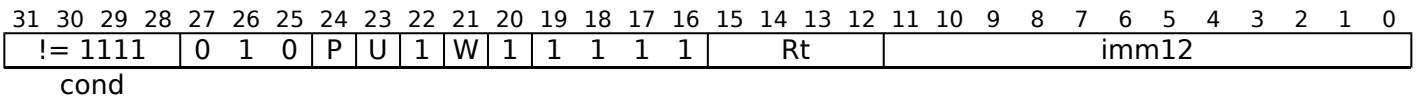
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDRB (literal)

Load Register Byte (literal) calculates an address from the PC value and an immediate offset, loads a byte from memory, zero-extends it to form a 32-bit word, and writes it to a register. For information about memory accesses see *Memory accesses*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



A1 (!(P == 0 && W == 1))

```
LDRB{<c>}{<q>} <Rt>, <label> // (Normal form)

LDRB{<c>}{<q>} <Rt>, [PC, #{+/-}<imm>] // (Alternative form)
```

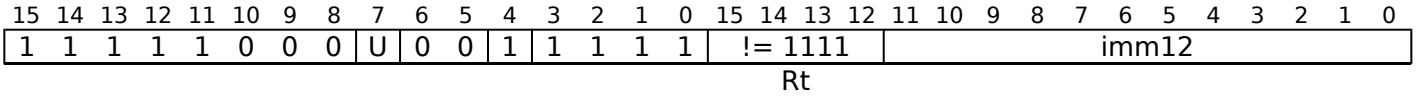
```
if P == '0' && W == '1' then SEE "LDRBT";
t = UInt(Rt);  imm32 = ZeroExtend(imm12, 32);
add = (U == '1');  wback = (P == '0') || (W == '1');
if t == 15 || wback then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `wback`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes with the additional decode: wback = FALSE;.
- The instruction treats bit[24] as the P bit, and bit[21] as the writeback (W) bit, and uses the same addressing mode as described in *LDRB (immediate)*. The instruction uses post-indexed addressing when P == '0' and uses pre-indexed addressing otherwise. The instruction is handled as described in *Using R15*.

T1



T1

```
LDRB{<c>}{<q>} <Rt>, <label> // (Preferred syntax)

LDRB{<c>}{<q>} <Rt>, [PC, #{+/-}<imm>] // (Alternative syntax)
```

```
if Rt == '1111' then SEE "PLD";
t = UInt(Rt);  imm32 = ZeroExtend(imm12, 32);  add = (U == '1');
// Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rt>	Is the general-purpose register to be transferred, encoded in the "Rt" field.

<label> The label of the literal data item that is to be loaded into <Rt>. The assembler calculates the required value of the offset from the Align(PC, 4) value of the instruction to this label. Permitted values of the offset are -4095 to 4095.

If the offset is zero or positive, imm32 is equal to the offset and add == TRUE, encoded as U == 1.

If the offset is negative, imm32 is equal to minus the offset and add == FALSE, encoded as U == 0.

+/- Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<imm> For encoding A1: is the 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 if omitted, and encoded in the "imm12" field.

For encoding T1: is a 12-bit unsigned immediate byte offset, in the range 0 to 4095, encoded in the "imm12" field.

The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see *Use of labels in UAL instruction syntax*.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    base = Align(PC,4);
    address = if add then (base + imm32) else (base - imm32);
    R[t] = ZeroExtend(MemU[address,1], 32);
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

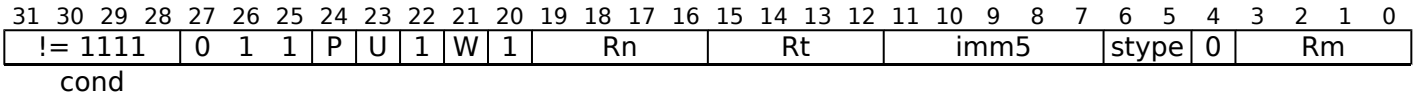
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDRB (register)

Load Register Byte (register) calculates an address from a base register value and an offset register value, loads a byte from memory, zero-extends it to form a 32-bit word, and writes it to a register. The offset register value can optionally be shifted. For information about memory accesses see *Memory accesses*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2) .

A1



Offset (P == 1 && W == 0)

```
LDRB{<c>}{<q>} <Rt>, [<Rn>, {+/-}<Rm>{, <shift>}]
```

Post-indexed (P == 0 && W == 0)

```
LDRB{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>{, <shift>}
```

Pre-indexed (P == 1 && W == 1)

```
LDRB{<c>}{<q>} <Rt>, [<Rn>, {+/-}<Rm>{, <shift>}]!
```

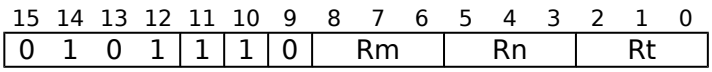
```
if P == '0' && W == '1' then SEE "LDRBT";
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm5);
if t == 15 || m == 15 then UNPREDICTABLE;
if wback && (n == 15 || n == t) then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1

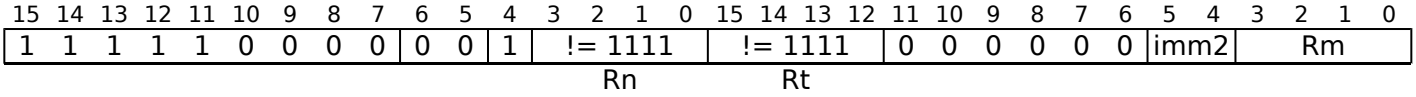


T1

```
LDRB{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>]
```

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
```

T2



T2

```
LDRB{<c>}.W <Rt>, [<Rn>, {+}<Rm>] // (<Rt>, <Rn>, <Rm> can be represented in T1)

LDRB{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>{, LSL #<imm>}]
```

```
if Rt == '1111' then SEE "PLD";
if Rn == '1111' then SEE "LDRB (literal)";
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, UInt(imm2));
if m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<Rt> Is the general-purpose register to be transferred, encoded in the "Rt" field.

<Rn> For encoding A1: is the general-purpose base register, encoded in the "Rn" field. The PC can be used in the offset variant.

For encoding T1 and T2: is the general-purpose base register, encoded in the "Rn" field.

+/- Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

\+ Specifies the index register is added to the base register.

<Rm> Is the general-purpose index register, encoded in the "Rm" field.

<shift> The shift to apply to the value read from <Rm>. If absent, no shift is applied. Otherwise, see *Shifts applied to a register*.

<imm> If present, the size of the left shift to apply to the value from <Rm>, in the range 1-3. <imm> is encoded in imm2. If absent, no shift is specified and imm2 is encoded as 0b00.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset = Shift(R[m], shift_t, shift_n, PSTATE.C);
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if index then offset_addr else R[n];
    R[t] = ZeroExtend(MemU[address,1],32);
    if wback then R[n] = offset_addr;
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDRBT

Load Register Byte Unprivileged loads a byte from memory, zero-extends it to form a 32-bit word, and writes it to a register. For information about memory accesses see *Memory accesses*.

The memory access is restricted as if the PE were running in User mode. This makes no difference if the PE is actually running in User mode.

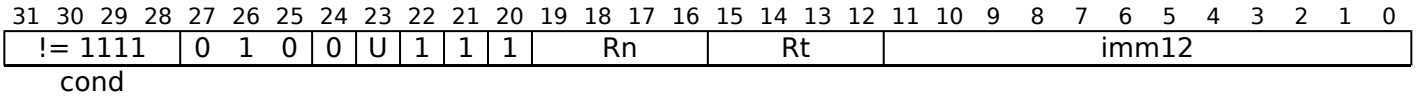
LDRBT is UNPREDICTABLE in Hyp mode.

The T32 instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an immediate offset, and leaves the base register unchanged.

The A32 instruction uses a post-indexed addressing mode, that uses a base register value as the address for the memory access, and calculates a new address from a base register value and an offset and writes it back to the base register. The offset can be an immediate value or an optionally-shifted register value.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1) .

A1



A1

```
LDRBT{<c>}{<q>} <Rt>, [<Rn>] {, #{+/-}<imm>}

t = UInt(Rt);  n = UInt(Rn);  postindex = TRUE;  add = (U == '1');
register_form = FALSE;  imm32 = ZeroExtend(imm12, 32);
if t == 15 || n == 15 || n == t then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

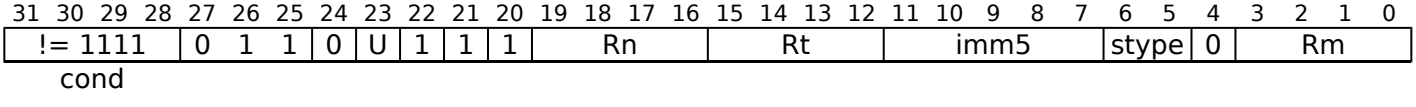
If `n == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction uses post-indexed addressing with the base register as PC. This is handled as described in *Using R15*.
- The instruction uses immediate offset addressing with the base register as PC, without writeback.

If `n == t && n != 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

A2



A2

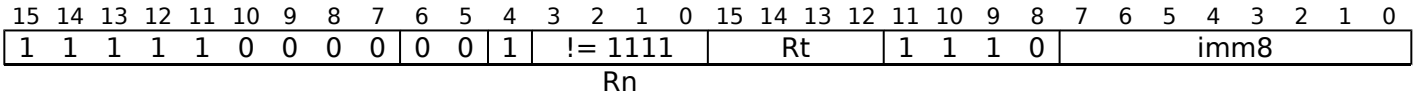
```
LDRBT{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>{, <shift>}

t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);  postindex = TRUE;  add = (U == '1');
register_form = TRUE;  (shift_t, shift_n) = DecodeImmShift(stype, imm5);
if t == 15 || n == 15 || n == t || m == 15 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `n == t && n != 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1



T1

```
LDRBT{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]

if Rn == '1111' then SEE "LDRB (literal)";
t = UInt(Rt);  n = UInt(Rn);  postindex = FALSE;  add = TRUE;
register_form = FALSE;  imm32 = ZeroExtend(imm8, 32);
if t == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<Rt> For encoding A1: is the general-purpose register to be transferred, encoded in the "Rt" field. The PC can be used, but this is deprecated.

For encoding A2 and T1: is the general-purpose register to be transferred, encoded in the "Rt" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

+/- For encoding A1: specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

For encoding A2: specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<Rm> Is the general-purpose index register, encoded in the "Rm" field.

<shift> The shift to apply to the value read from <Rm>. If absent, no shift is applied. Otherwise, see *Shifts applied to a register*.

+ Specifies the offset is added to the base register.

<imm> For encoding A1: is the 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 if omitted, and encoded in the "imm12" field.

For encoding T1: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field.

Operation

```
if ConditionPassed() then
    if PSTATE.EL == EL2 then UNPREDICTABLE;               // Hyp mode
    EncodingSpecificOperations();
    offset = if register_form then Shift(R[m], shift_t, shift_n, PSTATE.C) else imm32;
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if postindex then R[n] else offset_addr;
    R[t] = ZeroExtend(MemU_unpriv[address,1],32);
    if postindex then R[n] = offset_addr;
```

CONSTRAINED UNPREDICTABLE behavior

If `PSTATE.EL == EL2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes as `LDRB` (immediate).

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

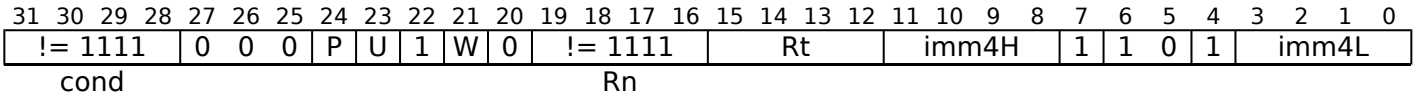
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDRD (immediate)

Load Register Dual (immediate) calculates an address from a base register value and an immediate offset, loads two words from memory, and writes them to two registers. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see *Memory accesses*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



Offset (P == 1 && W == 0)

```
LDRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn> {, #{+/-}<imm>}]
```

Post-indexed (P == 0 && W == 0)

```
LDRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>], #{+/-}<imm>
```

Pre-indexed (P == 1 && W == 1)

```
LDRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>, #{+/-}<imm>]!
```

```
if Rn == '1111' then SEE "LDRD (literal)";
if Rt<0> == '1' then UNPREDICTABLE;
t = UInt(Rt);  t2 = t+1;  n = UInt(Rn);  imm32 = ZeroExtend(imm4H:imm4L, 32);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
if P == '0' && W == '1' then UNPREDICTABLE;
if wback && (n == t || n == t2) then UNPREDICTABLE;
if t2 == 15 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `wback && (n == t || n == t2)`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

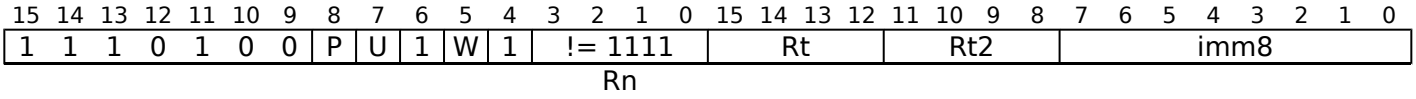
If `P == '0' && W == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as an LDRD using one of offset, post-indexed, or pre-indexed addressing.

If `Rt<0> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: t<0> = '0'.
- The instruction executes with the additional decode: t2 = t.
- The instruction executes as described, with no change to its behavior and no additional side-effects. This does not apply when Rt == '1111'.

T1



Offset (P == 1 && W == 0)

```
LDRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn> {, #{+/-}<imm>}]
```

Post-indexed (P == 0 && W == 1)

```
LDRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>], #{+/-}<imm>
```

Pre-indexed (P == 1 && W == 1)

```
LDRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>, #{+/-}<imm>]!
```

```
if P == '0' && W == '0' then SEE "Related encodings";
if Rn == '1111' then SEE "LDRD (literal)";
t = UInt(Rt);  t2 = UInt(Rt2);  n = UInt(Rn);  imm32 = ZeroExtend(imm8:'00', 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
if wback && (n == t || n == t2) then UNPREDICTABLE;
if t == 15 || t2 == 15 || t == t2 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

CONSTRAINED UNPREDICTABLE behavior

If `wback && (n == t || n == t2)`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

If `t == t2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The load instruction executes but the destination register takes an UNKNOWN value.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Related encodings: *Load/store dual, load/store exclusive, table branch*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rt>	For encoding A1: is the first general-purpose register to be transferred, encoded in the "Rt" field. This register must be even-numbered and not R14. For encoding T1: is the first general-purpose register to be transferred, encoded in the "Rt" field.
<Rt2>	For encoding A1: is the second general-purpose register to be transferred. This register must be <R(t+1)>. For encoding T1: is the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Rn>	Is the general-purpose base register, encoded in the "Rn" field. For PC use see *LDRD (literal)*.
+/-	Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<imm> For encoding A1: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm4H:imm4L" field.

For encoding T1: is the unsigned immediate byte offset, a multiple of 4, in the range 0 to 1020, defaulting to 0 if omitted, and encoded in the "imm8" field as <imm>/4.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    if address == Align(address, 8) then
        data = MemA[address,8];
        if BigEndian(AccType_ATOMIC)  then
            R[t] = data<63:32>;
            R[t2] = data<31:0>;
        else
            R[t] = data<31:0>;
            R[t2] = data<63:32>;
    else
        R[t] = MemA[address,4];
        R[t2] = MemA[address+4,4];
    if wback then R[n] = offset_addr;
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

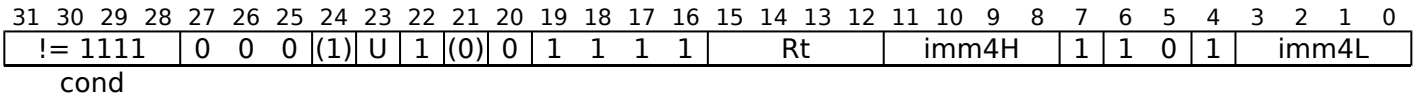
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDRD (literal)

Load Register Dual (literal) calculates an address from the PC value and an immediate offset, loads two words from memory, and writes them to two registers. For information about memory accesses see *Memory accesses*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



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

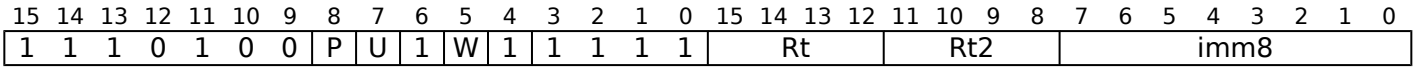
If `Rt<0> == '1'`, then one of the following behaviors must occur:

- 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



T1 (!(P == 0 && W == 0))

```
LDRD{<c>}{<q>} <Rt>, <Rt2>, <label> // (Normal form)

LDRD{<c>}{<q>} <Rt>, <Rt2>, [PC, #{+/-}<imm>] // (Alternative form)
```

```
if P == '0' && W == '0' then SEE "Related encodings";
t = UInt(Rt);  t2 = UInt(Rt2);
imm32 = ZeroExtend(imm8:'00', 32);  add = (U == '1');
if t == 15 || t2 == 15 || t == t2 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
if W == '1' then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `t == t2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

- The load instruction executes but the destination register takes an UNKNOWN value.

If `W == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes without writeback of the base address.
- The instruction uses post-indexed addressing when P == '0' and uses pre-indexed addressing otherwise. The instruction is handled as described in *Using R15*.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Related encodings: *Load/Store dual, Load/Store-Exclusive, Load-Acquire/Store-Release, table branch*.

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<Rt> For encoding A1: is the first general-purpose register to be transferred, encoded in the "Rt" field. This register must be even-numbered and not R14.

For encoding T1: is the first general-purpose register to be transferred, encoded in the "Rt" field.

<Rt2> For encoding A1: is the second general-purpose register to be transferred. This register must be <R(t+1)>.

For encoding T1: is the second general-purpose register to be transferred, encoded in the "Rt2" field.

<label> For encoding A1: the label of the literal data item that is to be loaded into <Rt>. The assembler calculates the required value of the offset from the Align(PC, 4) value of the instruction to this label. Any value in the range -255 to 255 is permitted.

If the offset is zero or positive, imm32 is equal to the offset and add == TRUE, encoded as U == 1. If the offset is negative, imm32 is equal to minus the offset and add == FALSE, encoded as U == 0.

For encoding T1: the label of the literal data item that is to be loaded into <Rt>. The assembler calculates the required value of the offset from the Align(PC, 4) value of the instruction to this label. Permitted values of the offset are multiples of 4 in the range -1020 to 1020.

If the offset is zero or positive, imm32 is equal to the offset and add == TRUE, encoded as U == 1.

If the offset is negative, imm32 is equal to minus the offset and add == FALSE, encoded as U == 0.

+/- Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<imm> For encoding A1: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm4H:imm4L" field.

For encoding T1: is the optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field.

The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see *Use of labels in UAL instruction syntax*.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = if add then (Align(PC,4) + imm32) else (Align(PC,4) - imm32);
    if address == Align(address, 8) then
        data = MemA[address,8];
        if BigEndian(AccType_ATOMIC) then
            R[t] = data<63:32>;
            R[t2] = data<31:0>;
        else
            R[t] = data<31:0>;
            R[t2] = data<63:32>;
    else
        R[t] = MemA[address,4];
        R[t2] = MemA[address+4,4];
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

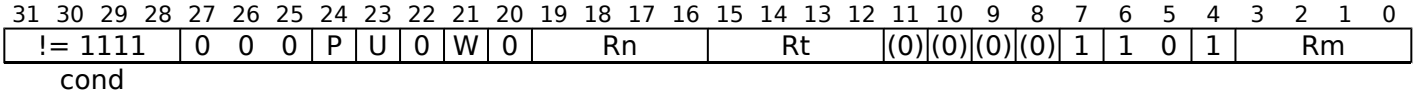
Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDRD (register)

Load Register Dual (register) calculates an address from a base register value and a register offset, loads two words from memory, and writes them to two registers. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see *Memory accesses*.

A1



Offset (P == 1 && W == 0)

```
LDRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>, {+/-}<Rm>]
```

Post-indexed (P == 0 && W == 0)

```
LDRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>], {+/-}<Rm>
```

Pre-indexed (P == 1 && W == 1)

```
LDRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>, {+/-}<Rm>]!
```

```
if Rt<0> == '1' then UNPREDICTABLE;
t = UInt(Rt);  t2 = t+1;  n = UInt(Rn);  m = UInt(Rm);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
if P == '0' && W == '1' then UNPREDICTABLE;
if t2 == 15 || m == 15 || m == t || m == t2 then UNPREDICTABLE;
if wback && (n == 15 || n == t || n == t2) then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `wback && (n == t || n == t2)`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

If `P == '0' && W == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as an LDRD using one of offset, post-indexed, or pre-indexed addressing.

If `m == t || m == t2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction loads register Rm with an UNKNOWN value.

If `Rt<0> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: t<0> = '0'.
- The instruction executes with the additional decode: t2 = t.
- The instruction executes as described, with no change to its behavior and no additional side-effects. This does not apply when Rt == '1111'.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<Rt> Is the first general-purpose register to be transferred, encoded in the "Rt" field. This register must be even-numbered and not R14.

<Rt2> Is the second general-purpose register to be transferred. This register must be <R(t+1)>.

<Rn> Is the general-purpose base register, encoded in the "Rn" field. The PC can be used in the offset variant.

+/- Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<Rm> Is the general-purpose index register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + R[m]) else (R[n] - R[m]);
    address = if index then offset_addr else R[n];
    if address == Align(address, 8) then
        data = MemA[address,8];
        if BigEndian(AccType_ATOMIC) then
            R[t] = data<63:32>;
            R[t2] = data<31:0>;
        else
            R[t] = data<31:0>;
            R[t2] = data<63:32>;
    else
        R[t] = MemA[address,4];
        R[t2] = MemA[address+4,4];

    if wback then R[n] = offset_addr;
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDREX

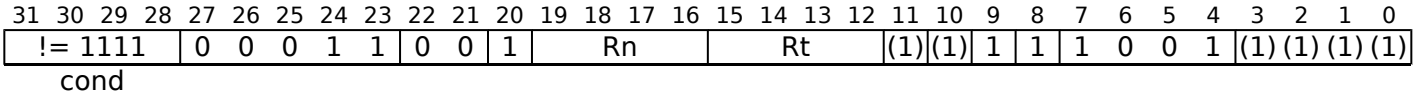
Load Register Exclusive calculates an address from a base register value and an immediate offset, loads a word from memory, writes it to a register and:

- If the address has the Shared Memory attribute, marks the physical address as exclusive access for the executing PE in a global monitor.
- Causes the executing PE to indicate an active exclusive access in the local monitor.

For more information about support for shared memory see *Synchronization and semaphores*. For information about memory accesses see *Memory accesses*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

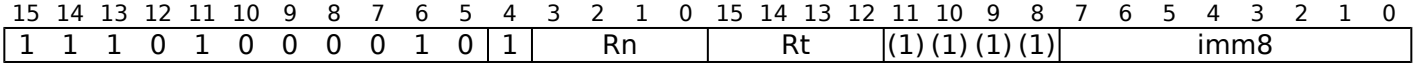
A1



A1

```
LDREX{<c>}{<q>} <Rt>, [<Rn> {, {#}<imm>}]

t = UInt(Rt);  n = UInt(Rn);  imm32 = Zeros(32); // Zero offset
if t == 15 || n == 15 then UNPREDICTABLE;
```

T1



T1

```
LDREX{<c>}{<q>} <Rt>, [<Rn> {, #<imm>}]

t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm8:'00', 32);
if t == 15 || n == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rt>	Is the general-purpose register to be transferred, encoded in the "Rt" field.
<Rn>	Is the general-purpose base register, encoded in the "Rn" field.
<imm>	For encoding A1: the immediate offset added to the value of <Rn> to calculate the address. <imm> can only be 0 or omitted. For encoding T1: the immediate offset added to the value of <Rn> to calculate the address. <imm> can be omitted, meaning an offset of 0. Values are multiples of 4 in the range 0-1020.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n] + imm32;
    AArch32.SetExclusiveMonitors(address,4);
    R[t] = MemA[address,4];
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDREXB

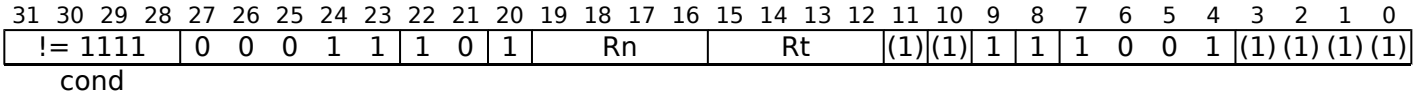
Load Register Exclusive Byte derives an address from a base register value, loads a byte from memory, zero-extends it to form a 32-bit word, writes it to a register and:

- If the address has the Shared Memory attribute, marks the physical address as exclusive access for the executing PE in a global monitor.
- Causes the executing PE to indicate an active exclusive access in the local monitor.

For more information about support for shared memory see *Synchronization and semaphores*. For information about memory accesses see *Memory accesses*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

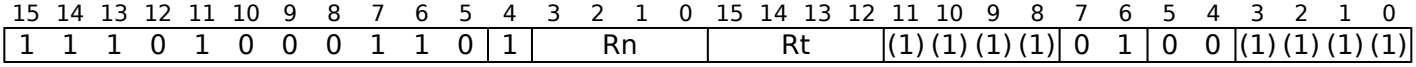
A1



A1

```
LDREXB{<c>}{<q>} <Rt>, [<Rn>]

t = UInt(Rt);  n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

T1



T1

```
LDREXB{<c>}{<q>} <Rt>, [<Rn>]

t = UInt(Rt);  n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rt>	Is the general-purpose register to be transferred, encoded in the "Rt" field.
<Rn>	Is the general-purpose base register, encoded in the "Rn" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    AArch32.SetExclusiveMonitors(address,1);
    R[t] = ZeroExtend(MemA[address,1], 32);
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDREXD

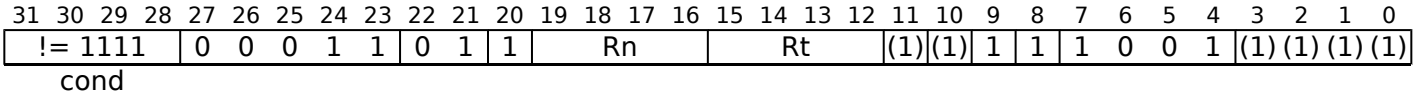
Load Register Exclusive Doubleword derives an address from a base register value, loads a 64-bit doubleword from memory, writes it to two registers and:

- If the address has the Shared Memory attribute, marks the physical address as exclusive access for the executing PE in a global monitor.
- Causes the executing PE to indicate an active exclusive access in the local monitor.

For more information about support for shared memory see *Synchronization and semaphores*. For information about memory accesses see *Memory accesses*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



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

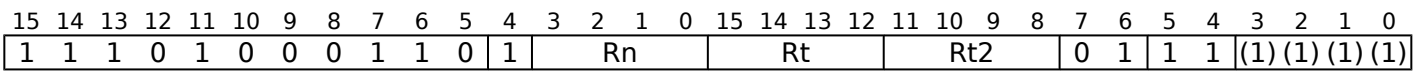
If `Rt<0> == '1'`, then one of the following behaviors must occur:

- 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



T1

```
LDREXD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>]

t = UInt(Rt);  t2 = UInt(Rt2);  n = UInt(Rn);
if t == 15 || t2 == 15 || t == t2 || n == 15 then UNPREDICTABLE;
// Armv8-A removes UNPREDICTABLE for R13
```

CONSTRAINED UNPREDICTABLE behavior

If `t == t2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The load instruction executes but the destination register takes an UNKNOWN value.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rt>	For encoding A1: is the first general-purpose register to be transferred, encoded in the "Rt" field. <Rt> must be even-numbered and not R14.
	For encoding T1: is the first general-purpose register to be transferred, encoded in the "Rt" field.
<Rt2>	For encoding A1: is the second general-purpose register to be transferred. <Rt2> must be <R(t+1)>.
	For encoding T1: is the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Rn>	Is the general-purpose base register, encoded in the "Rn" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    AArch32.SetExclusiveMonitors(address,8);
    value = MemA[address,8];
    // Extract words from 64-bit loaded value such that R[t] is
    // loaded from address and R[t2] from address+4.
    R[t]  = if BigEndian(AccType_ATOMIC) then value<63:32> else value<31:0>;
    R[t2] = if BigEndian(AccType_ATOMIC) then value<31:0> else value<63:32>;
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDREXH

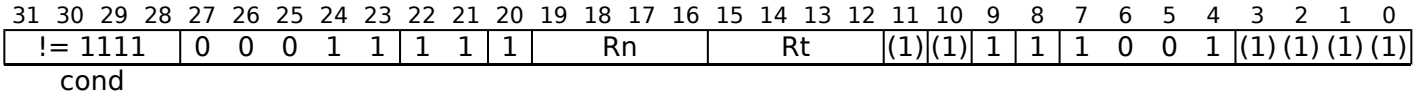
Load Register Exclusive Halfword derives an address from a base register value, loads a halfword from memory, zero-extends it to form a 32-bit word, writes it to a register and:

- If the address has the Shared Memory attribute, marks the physical address as exclusive access for the executing PE in a global monitor.
- Causes the executing PE to indicate an active exclusive access in the local monitor.

For more information about support for shared memory see *Synchronization and semaphores*. For information about memory accesses see *Memory accesses*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

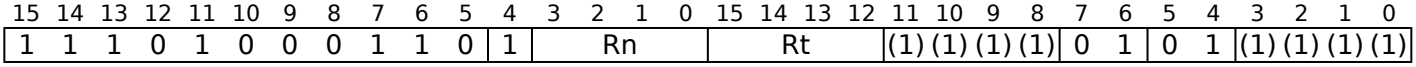
A1



A1

```
LDREXH{<c>}{<q>} <Rt>, [<Rn>]

t = UInt(Rt);  n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

T1



T1

```
LDREXH{<c>}{<q>} <Rt>, [<Rn>]

t = UInt(Rt);  n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rt>	Is the general-purpose register to be transferred, encoded in the "Rt" field.
<Rn>	Is the general-purpose base register, encoded in the "Rn" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    AArch32.SetExclusiveMonitors(address,2);
    R[t] = ZeroExtend(MemA[address,2], 32);
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

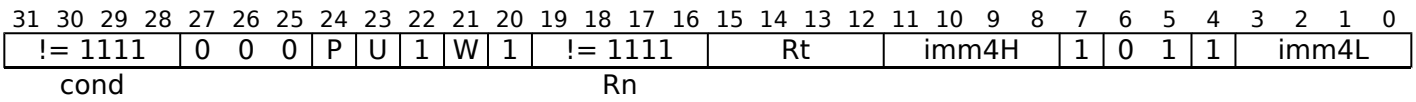
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDRH (immediate)

Load Register Halfword (immediate) calculates an address from a base register value and an immediate offset, loads a halfword from memory, zero-extends it to form a 32-bit word, and writes it to a register. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see *Memory accesses*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 , T2 and T3) .

A1



Offset (P == 1 && W == 0)

```
LDRH{<c>}{<q>} <Rt>, [<Rn> {, #{+/-}<imm>}]
```

Post-indexed (P == 0 && W == 0)

```
LDRH{<c>}{<q>} <Rt>, [<Rn>], #{+/-}<imm>
```

Pre-indexed (P == 1 && W == 1)

```
LDRH{<c>}{<q>} <Rt>, [<Rn>, #{+/-}<imm>]!
```

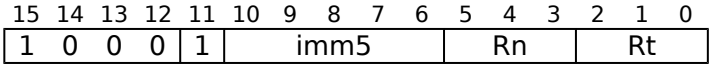
```
if Rn == '1111' then SEE "LDRH (literal)";
if P == '0' && W == '1' then SEE "LDRHT";
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm4H:imm4L, 32);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
if t == 15 || (wback && n == t) then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1

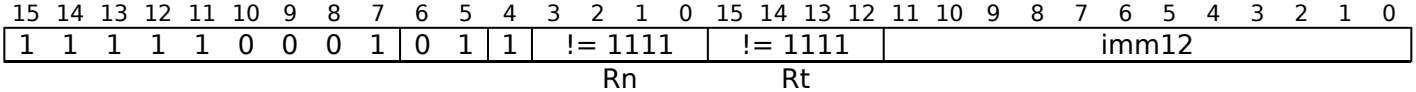


T1

```
LDRH{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```

```
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm5:'0', 32);
index = TRUE;  add = TRUE;  wback = FALSE;
```

T2

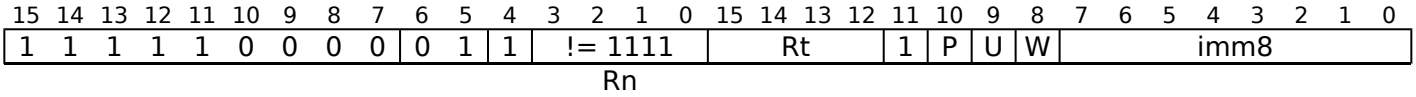


T2

```
LDRH{<c>}.W <Rt>, [<Rn> {, #{+}<imm>}] // (<Rt>, <Rn>, <imm> can be represented in T1)

LDRH{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```

```
if Rt == '1111' then SEE "PLD (immediate)";
if Rn == '1111' then SEE "LDRH (literal)";
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
index = TRUE;  add = TRUE;  wback = FALSE;
// Armv8-A removes UNPREDICTABLE for R13
```

T3



Offset (Rt != 1111 && P == 1 && U == 0 && W == 0)

```
LDRH{<c>}{<q>} <Rt>, [<Rn> {, #-<imm>}]
```

Post-indexed (P == 0 && W == 1)

```
LDRH{<c>}{<q>} <Rt>, [<Rn>], #{+/-}<imm>
```

Pre-indexed (P == 1 && W == 1)

```
LDRH{<c>}{<q>} <Rt>, [<Rn>, #{+/-}<imm>]!
```

```
if Rn == '1111' then SEE "LDRH (literal)";
if Rt == '1111' && P == '1' && U == '0' && W == '0' then SEE "PLDW (immediate)";
if P == '1' && U == '1' && W == '0' then SEE "LDRHT";
if P == '0' && W == '0' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
if (t == 15 && W == '1') || (wback && n == t) then UNPREDICTABLE;
// Armv8-A removes UNPREDICTABLE for R13
```

CONSTRAINED UNPREDICTABLE behavior

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rt>	Is the general-purpose register to be transferred, encoded in the "Rt" field.
<Rn>	For encoding A1, T2 and T3: is the general-purpose base register, encoded in the "Rn" field. For PC use see *LDRH (literal)*.
	For encoding T1: is the general-purpose base register, encoded in the "Rn" field.

+/- Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

+ Specifies the offset is added to the base register.

<imm> For encoding A1: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm4H:imm4L" field.

For encoding T1: is the optional positive unsigned immediate byte offset, a multiple of 2, in the range 0 to 62, defaulting to 0 and encoded in the "imm5" field as <imm>/2.

For encoding T2: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

For encoding T3: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm8" field.

Operation

```
if CurrentInstrSet() == InstrSet_A32 then
    if ConditionPassed() then
        EncodingSpecificOperations();
        offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
        address = if index then offset_addr else R[n];
        data = MemU[address,2];
        if wback then R[n] = offset_addr;
        R[t] = ZeroExtend(data, 32);
else
    if ConditionPassed() then
        EncodingSpecificOperations();
        offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
        address = if index then offset_addr else R[n];
        data = MemU[address,2];
        if wback then R[n] = offset_addr;
        R[t] = ZeroExtend(data, 32);
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

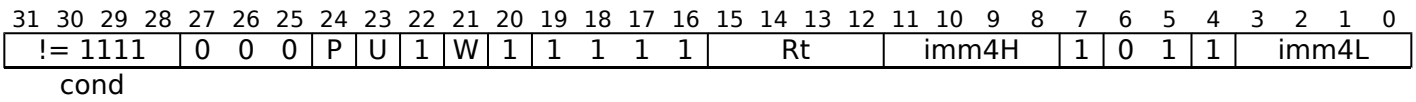
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDRH (literal)

Load Register Halfword (literal) calculates an address from the PC value and an immediate offset, loads a halfword from memory, zero-extends it to form a 32-bit word, and writes it to a register. For information about memory accesses see *Memory accesses*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



A1 (!(P == 0 && W == 1))

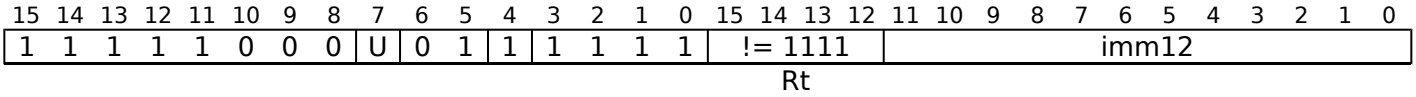
```
LDRH{<c>}{<q>} <Rt>, <label> // (Normal form)

LDRH{<c>}{<q>} <Rt>, [PC, #{+/-}<imm>] // (Alternative form)

if P == '0' && W == '1' then SEE "LDRHT";
t = UInt(Rt);  imm32 = ZeroExtend(imm4H:imm4L, 32);
add = (U == '1');  wback = (P == '0') || (W == '1');
if t == 15 || wback then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `wback`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes with the additional decode: wback = FALSE;.
- The instruction treats bit[24] as the P bit, and bit[21] as the writeback (W) bit, and uses the same addressing mode as described in *LDRH (immediate)*. The instruction uses post-indexed addressing when P == '0' and uses pre-indexed addressing otherwise. The instruction is handled as described in *Using R15*.

T1



T1

```
LDRH{<c>}{<q>} <Rt>, <label> // (Preferred syntax)

LDRH{<c>}{<q>} <Rt>, [PC, #{+/-}<imm>] // (Alternative syntax)

if Rt == '1111' then SEE "PLD (literal)";
t = UInt(Rt);  imm32 = ZeroExtend(imm12, 32);  add = (U == '1');
// Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rt>	Is the general-purpose register to be transferred, encoded in the "Rt" field.

<label> For encoding A1: the label of the literal data item that is to be loaded into <Rt>. The assembler calculates the required value of the offset from the Align(PC, 4) value of the instruction to this label. Any value in the range -255 to 255 is permitted.

If the offset is zero or positive, imm32 is equal to the offset and add == TRUE, encoded as U == 1. If the offset is negative, imm32 is equal to minus the offset and add == FALSE, encoded as U == 0.

For encoding T1: the label of the literal data item that is to be loaded into <Rt>. The assembler calculates the required value of the offset from the Align(PC, 4) value of the instruction to this label. Permitted values of the offset are -4095 to 4095.

If the offset is zero or positive, imm32 is equal to the offset and add == TRUE, encoded as U == 1.

If the offset is negative, imm32 is equal to minus the offset and add == FALSE, encoded as U == 0.

+/- Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<imm> For encoding A1: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm4H:imm4L" field.

For encoding T1: is a 12-bit unsigned immediate byte offset, in the range 0 to 4095, encoded in the "imm12" field.

The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see *Use of labels in UAL instruction syntax*.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    base = Align(PC,4);
    address = if add then (base + imm32) else (base - imm32);
    data = MemU[address,2];
    R[t] = ZeroExtend(data, 32);
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

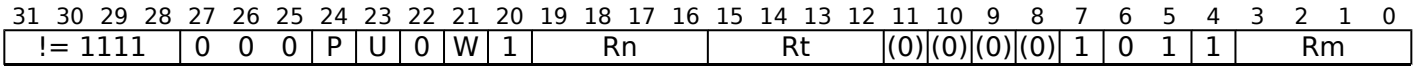
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDRH (register)

Load Register Halfword (register) calculates an address from a base register value and an offset register value, loads a halfword from memory, zero-extends it to form a 32-bit word, and writes it to a register. The offset register value can be shifted left by 0, 1, 2, or 3 bits. For information about memory accesses see *Memory accesses*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2) .

A1



Offset (P == 1 && W == 0)

```
LDRH{<c>}{<q>} <Rt>, [<Rn>, {+/-}<Rm>]
```

Post-indexed (P == 0 && W == 0)

```
LDRH{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>
```

Pre-indexed (P == 1 && W == 1)

```
LDRH{<c>}{<q>} <Rt>, [<Rn>, {+/-}<Rm>]!
```

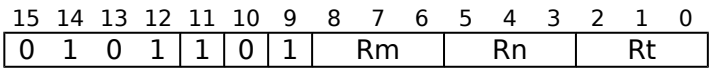
```
if P == '0' && W == '1' then SEE "LDRHT";
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
(shift_t, shift_n) = (SRType_LSL, 0);
if t == 15 || m == 15 then UNPREDICTABLE;
if wback && (n == 15 || n == t) then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1

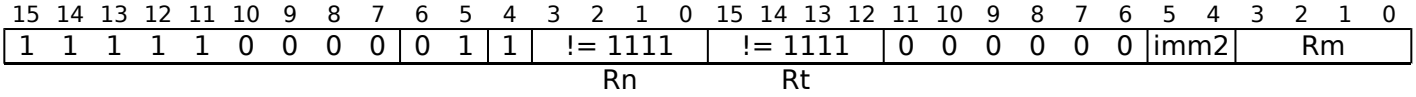


T1

```
LDRH{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>]
```

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
```

T2



T2

```
LDRH{<c>}.W <Rt>, [<Rn>, {+}<Rm>] // (<Rt>, <Rn>, <Rm> can be represented in T1)

LDRH{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>{, LSL #<imm>}]
```

```
if Rn == '1111' then SEE "LDRH (literal)";
if Rt == '1111' then SEE "PLDW (register)";
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, UInt(imm2));
if m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<Rt> Is the general-purpose register to be transferred, encoded in the "Rt" field.

<Rn> For encoding A1: is the general-purpose base register, encoded in the "Rn" field. The PC can be used in the offset variant.

For encoding T1 and T2: is the general-purpose base register, encoded in the "Rn" field.

+/- Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

\+ Specifies the index register is added to the base register.

<Rm> Is the general-purpose index register, encoded in the "Rm" field.

<imm> If present, the size of the left shift to apply to the value from <Rm>, in the range 1-3. <imm> is encoded in imm2. If absent, no shift is specified and imm2 is encoded as 0b00.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset = Shift(R[m], shift_t, shift_n, PSTATE.C);
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if index then offset_addr else R[n];
    data = MemU[address,2];
    if wback then R[n] = offset_addr;
    R[t] = ZeroExtend(data, 32);
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDRHT

Load Register Halfword Unprivileged loads a halfword from memory, zero-extends it to form a 32-bit word, and writes it to a register. For information about memory accesses see *Memory accesses*.

The memory access is restricted as if the PE were running in User mode. This makes no difference if the PE is actually running in User mode.

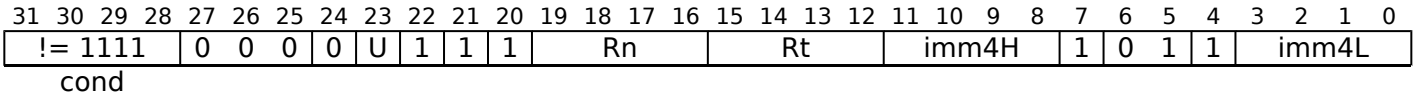
LDRHT is UNPREDICTABLE in Hyp mode.

The T32 instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an immediate offset, and leaves the base register unchanged.

The A32 instruction uses a post-indexed addressing mode, that uses a base register value as the address for the memory access, and calculates a new address from a base register value and an offset and writes it back to the base register. The offset can be an immediate value or a register value.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1) .

A1



A1

```
LDRHT{<c>}{<q>} <Rt>, [<Rn>] {, #{+/-}<imm>}

t = UInt(Rt);  n = UInt(Rn);  postindex = TRUE;  add = (U == '1');
register_form = FALSE;  imm32 = ZeroExtend(imm4H:imm4L, 32);
if t == 15 || n == 15 || n == t then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

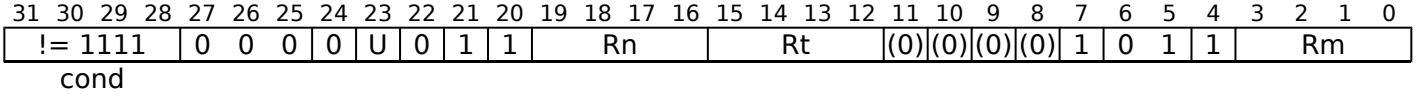
If `n == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction uses post-indexed addressing with the base register as PC. This is handled as described in *Using R15*.
- The instruction is treated as if bit[24] == '1' and bit[21] == '0'. The instruction uses immediate offset addressing with the base register as PC, without writeback.

If `n == t && n != 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

A2



A2

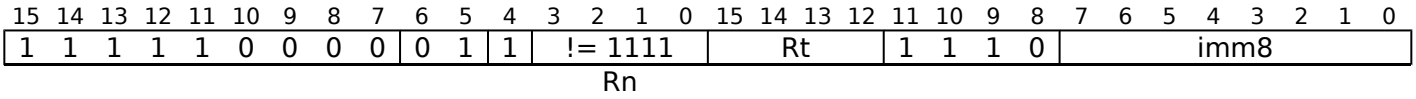
```
LDRHT{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>

t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);  postindex = TRUE;  add = (U == '1');
register_form = TRUE;
if t == 15 || n == 15 || n == t || m == 15 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `n == t && n != 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1



T1

```
LDRHT{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]

if Rn == '1111' then SEE "LDRH (literal)";
t = UInt(Rt);  n = UInt(Rn);  postindex = FALSE;  add = TRUE;
register_form = FALSE;  imm32 = ZeroExtend(imm8, 32);
if t == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<Rt> Is the general-purpose register to be transferred, encoded in the "Rt" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

+/- For encoding A1: specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

For encoding A2: specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<Rm> Is the general-purpose index register, encoded in the "Rm" field.

\+ Specifies the offset is added to the base register.

<imm> For encoding A1: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm4H:imm4L" field.

For encoding T1: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field.

Operation

```
if ConditionPassed() then
    if PSTATE.EL == EL2 then UNPREDICTABLE;               // Hyp mode
    EncodingSpecificOperations();
    offset = if register_form then R[m] else imm32;
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if postindex then R[n] else offset_addr;
    data = MemU_unpriv[address,2];
    if postindex then R[n] = offset_addr;
    R[t] = ZeroExtend(data, 32);
```

CONSTRAINED UNPREDICTABLE behavior

If `PSTATE.EL == EL2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes as `LDRH` (immediate).

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

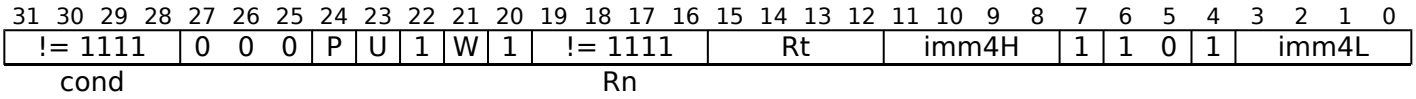
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDRSB (immediate)

Load Register Signed Byte (immediate) calculates an address from a base register value and an immediate offset, loads a byte from memory, sign-extends it to form a 32-bit word, and writes it to a register. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see *Memory accesses*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2) .

A1



Offset (P == 1 && W == 0)

```
LDRSB{<c>}{<q>} <Rt>, [<Rn> {, #{+/-}<imm>}]
```

Post-indexed (P == 0 && W == 0)

```
LDRSB{<c>}{<q>} <Rt>, [<Rn>], #{+/-}<imm>
```

Pre-indexed (P == 1 && W == 1)

```
LDRSB{<c>}{<q>} <Rt>, [<Rn>, #{+/-}<imm>]!
```

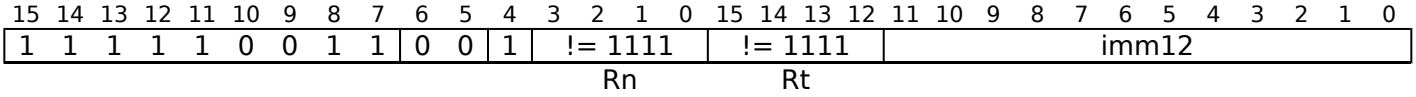
```
if Rn == '1111' then SEE "LDRSB (literal)";
if P == '0' && W == '1' then SEE "LDRSBT";
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm4H:imm4L, 32);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
if t == 15 || (wback && n == t) then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1

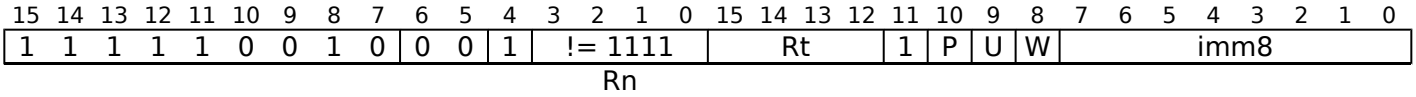


T1

```
LDRSB{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```

```
if Rt == '1111' then SEE "PLI";
if Rn == '1111' then SEE "LDRSB (literal)";
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
index = TRUE;  add = TRUE;  wback = FALSE;
// Armv8-A removes UNPREDICTABLE for R13
```

T2



Offset (Rt != 1111 && P == 1 && U == 0 && W == 0)

```
LDRSB{<c>}{<q>} <Rt>, [<Rn> {, #-<imm>}]
```

Post-indexed (P == 0 && W == 1)

```
LDRSB{<c>}{<q>} <Rt>, [<Rn>], #{+/-}<imm>
```

Pre-indexed (P == 1 && W == 1)

```
LDRSB{<c>}{<q>} <Rt>, [<Rn>, #{+/-}<imm>]!
```

```
if Rt == '1111' && P == '1' && U == '0' && W == '0' then SEE "PLI";
if Rn == '1111' then SEE "LDRSB (literal)";
if P == '1' && U == '1' && W == '0' then SEE "LDRSBT";
if P == '0' && W == '0' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
if (t == 15 && W == '1') || (wback && n == t) then UNPREDICTABLE;
// Armv8-A removes UNPREDICTABLE for R13
```

CONSTRAINED UNPREDICTABLE behavior

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<Rt> Is the general-purpose register to be transferred, encoded in the "Rt" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field. For PC use see *LDRSB (literal)*.

+/- Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

\+ Specifies the offset is added to the base register.

<imm> For encoding A1: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm4H:imm4L" field.

For encoding T1: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

For encoding T2: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm8" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    R[t] = SignExtend(MemU[address,1], 32);
    if wback then R[n] = offset_addr;
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

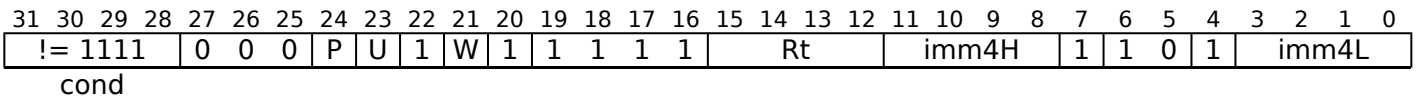
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDRSB (literal)

Load Register Signed Byte (literal) calculates an address from the PC value and an immediate offset, loads a byte from memory, sign-extends it to form a 32-bit word, and writes it to a register. For information about memory accesses see *Memory accesses*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



A1 (!(P == 0 && W == 1))

```
LDRSB{<c>}{<q>} <Rt>, <label> // (Normal form)

LDRSB{<c>}{<q>} <Rt>, [PC, #{+/-}<imm>] // (Alternative form)
```

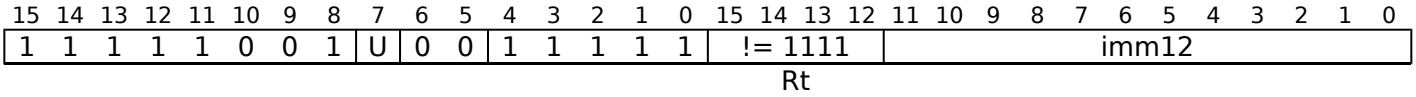
```
if P == '0' && W == '1' then SEE "LDRSBT";
t = UInt(Rt);  imm32 = ZeroExtend(imm4H:imm4L, 32);
add = (U == '1');  wback = (P == '0') || (W == '1');
if t == 15 || wback then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `wback`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes with the additional decode: wback = FALSE;.
- The instruction treats bit[24] as the P bit, and bit[21] as the writeback (W) bit, and uses the same addressing mode as described in *LDRSB (immediate)*. The instruction uses post-indexed addressing when P == '0' and uses pre-indexed addressing otherwise. The instruction is handled as described in *Using R15*.

T1



T1

```
LDRSB{<c>}{<q>} <Rt>, <label> // (Preferred syntax)

LDRSB{<c>}{<q>} <Rt>, [PC, #{+/-}<imm>] // (Alternative syntax)
```

```
if Rt == '1111' then SEE "PLI";
t = UInt(Rt);  imm32 = ZeroExtend(imm12, 32);  add = (U == '1');
// Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rt>	Is the general-purpose register to be transferred, encoded in the "Rt" field.

<label> For encoding A1: the label of the literal data item that is to be loaded into <Rt>. The assembler calculates the required value of the offset from the Align(PC, 4) value of the instruction to this label. Any value in the range -255 to 255 is permitted.

If the offset is zero or positive, imm32 is equal to the offset and add == TRUE, encoded as U == 1. If the offset is negative, imm32 is equal to minus the offset and add == FALSE, encoded as U == 0.

For encoding T1: the label of the literal data item that is to be loaded into <Rt>. The assembler calculates the required value of the offset from the Align(PC, 4) value of the instruction to this label. Permitted values of the offset are -4095 to 4095.

If the offset is zero or positive, imm32 is equal to the offset and add == TRUE, encoded as U == 1.

If the offset is negative, imm32 is equal to minus the offset and add == FALSE, encoded as U == 0.

+/- Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<imm> For encoding A1: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm4H:imm4L" field.

For encoding T1: is a 12-bit unsigned immediate byte offset, in the range 0 to 4095, encoded in the "imm12" field.

The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see *Use of labels in UAL instruction syntax*.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    base = Align(PC,4);
    address = if add then (base + imm32) else (base - imm32);
    R[t] = SignExtend(MemU[address,1], 32);
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

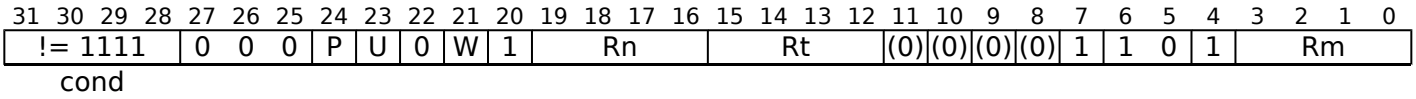
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDRSB (register)

Load Register Signed Byte (register) calculates an address from a base register value and an offset register value, loads a byte from memory, sign-extends it to form a 32-bit word, and writes it to a register. The offset register value can be shifted left by 0, 1, 2, or 3 bits. For information about memory accesses see *Memory accesses*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2) .

A1



Offset (P == 1 && W == 0)

```
LDRSB{<c>}{<q>} <Rt>, [<Rn>, {+/-}<Rm>]
```

Post-indexed (P == 0 && W == 0)

```
LDRSB{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>
```

Pre-indexed (P == 1 && W == 1)

```
LDRSB{<c>}{<q>} <Rt>, [<Rn>, {+/-}<Rm>]!
```

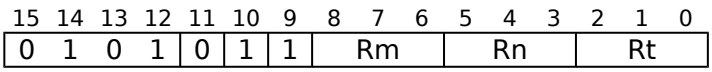
```
if P == '0' && W == '1' then SEE "LDRSBT";
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
(shift_t, shift_n) = (SRType_LSL, 0);
if t == 15 || m == 15 then UNPREDICTABLE;
if wback && (n == 15 || n == t) then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1

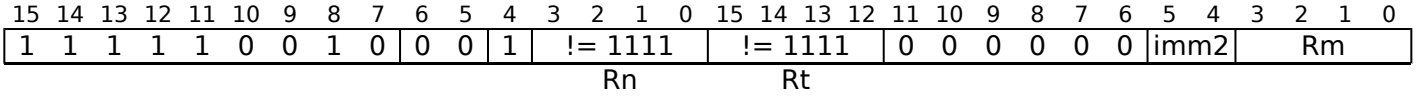


T1

```
LDRSB{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>]
```

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
```

T2



T2

```
LDRSB{<c>}.W <Rt>, [<Rn>, {+}<Rm>] // (<Rt>, <Rn>, <Rm> can be represented in T1)

LDRSB{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>{, LSL #<imm>}]
```

```
if Rt == '1111' then SEE "PLI";
if Rn == '1111' then SEE "LDRSB (literal)";
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, UInt(imm2));
if m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rt>	Is the general-purpose register to be transferred, encoded in the "Rt" field.
<Rn>	For encoding A1: is the general-purpose base register, encoded in the "Rn" field. The PC can be used in the offset variant. For encoding T1 and T2: is the general-purpose base register, encoded in the "Rn" field.
+/-	Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

+	Specifies the index register is added to the base register.
<Rm>	Is the general-purpose index register, encoded in the "Rm" field.
<imm>	If present, the size of the left shift to apply to the value from <Rm>, in the range 1-3. <imm> is encoded in imm2. If absent, no shift is specified and imm2 is encoded as 0b00.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset = Shift(R[m], shift_t, shift_n, PSTATE.C);
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if index then offset_addr else R[n];
    R[t] = SignExtend(MemU[address,1], 32);
    if wback then R[n] = offset_addr;
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDRSBT

Load Register Signed Byte Unprivileged loads a byte from memory, sign-extends it to form a 32-bit word, and writes it to a register. For information about memory accesses see *Memory accesses*.

The memory access is restricted as if the PE were running in User mode. This makes no difference if the PE is actually running in User mode.

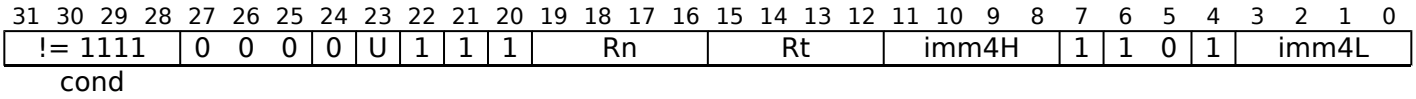
LDRSBT is UNPREDICTABLE in Hyp mode.

The T32 instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an immediate offset, and leaves the base register unchanged.

The A32 instruction uses a post-indexed addressing mode, that uses a base register value as the address for the memory access, and calculates a new address from a base register value and an offset and writes it back to the base register. The offset can be an immediate value or a register value.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1) .

A1



A1

```
LDRSBT{<c>}{<q>} <Rt>, [<Rn>] {, #{+/-}<imm>}

t = UInt(Rt);  n = UInt(Rn);  postindex = TRUE;  add = (U == '1');
register_form = FALSE;  imm32 = ZeroExtend(imm4H:imm4L, 32);
if t == 15 || n == 15 || n == t then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

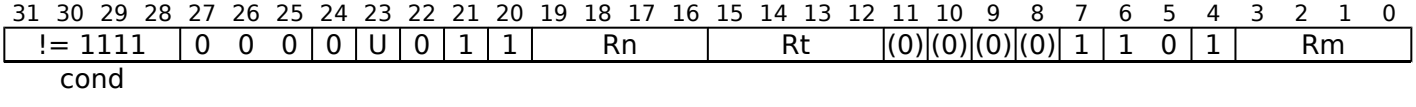
If `n == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction uses post-indexed addressing with the base register as PC. This is handled as described in *Using R15*.
- The instruction is treated as if bit[24] == '1' and bit[21] == '0'. The instruction uses immediate offset addressing with the base register as PC, without writeback.

If `n == t && n != 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

A2



A2

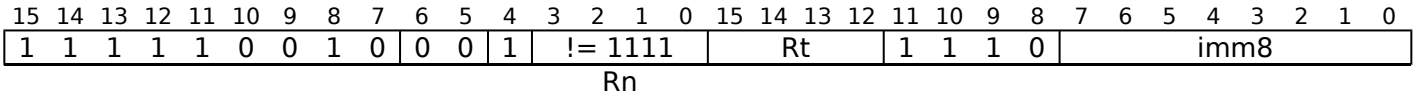
```
LDRSBT{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>

t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);  postindex = TRUE;  add = (U == '1');
register_form = TRUE;
if t == 15 || n == 15 || n == t || m == 15 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `n == t && n != 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1



T1

```
LDRSBT{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]

if Rn == '1111' then SEE "LDRSB (literal)";
t = UInt(Rt);  n = UInt(Rn);  postindex = FALSE;  add = TRUE;
register_form = FALSE;  imm32 = ZeroExtend(imm8, 32);
if t == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<Rt> Is the general-purpose register to be transferred, encoded in the "Rt" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

+/- For encoding A1: specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

For encoding A2: specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<Rm> Is the general-purpose index register, encoded in the "Rm" field.

\+ Specifies the offset is added to the base register.

<imm> For encoding A1: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm4H:imm4L" field.

For encoding T1: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field.

Operation

```
if ConditionPassed() then
    if PSTATE.EL == EL2 then UNPREDICTABLE;               // Hyp mode
    EncodingSpecificOperations();
    offset = if register_form then R[m] else imm32;
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if postindex then R[n] else offset_addr;
    R[t] = SignExtend(MemU_unpriv[address,1], 32);
    if postindex then R[n] = offset_addr;
```

CONSTRAINED UNPREDICTABLE behavior

If `PSTATE.EL == EL2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes as `LDRSB` (immediate).

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

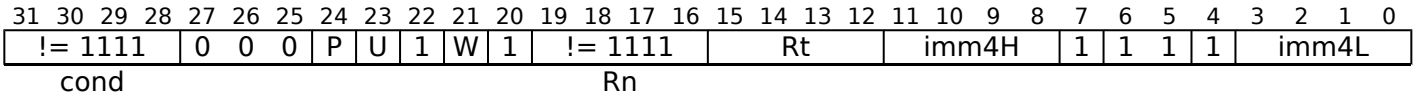
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDRSH (immediate)

Load Register Signed Halfword (immediate) calculates an address from a base register value and an immediate offset, loads a halfword from memory, sign-extends it to form a 32-bit word, and writes it to a register. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see *Memory accesses*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2) .

A1



Offset (P == 1 && W == 0)

```
LDRSH{<c>}{<q>} <Rt>, [<Rn> {, #{+/-}<imm>}]
```

Post-indexed (P == 0 && W == 0)

```
LDRSH{<c>}{<q>} <Rt>, [<Rn>], #{+/-}<imm>
```

Pre-indexed (P == 1 && W == 1)

```
LDRSH{<c>}{<q>} <Rt>, [<Rn>, #{+/-}<imm>]!
```

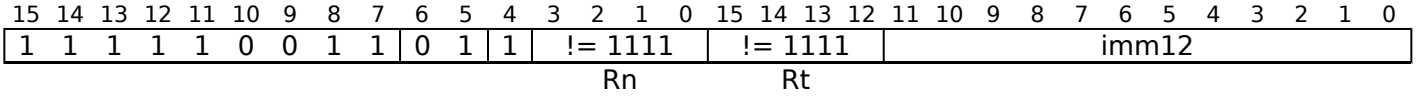
```
if Rn == '1111' then SEE "LDRSH (literal)";
if P == '0' && W == '1' then SEE "LDRSHT";
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm4H:imm4L, 32);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
if t == 15 || (wback && n == t) then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1

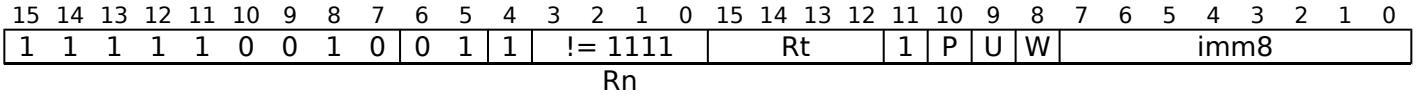


T1

```
LDRSH{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```

```
if Rn == '1111' then SEE "LDRSH (literal)";
if Rt == '1111' then SEE "Related instructions";
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
index = TRUE;  add = TRUE;  wback = FALSE;
// Armv8-A removes UNPREDICTABLE for R13
```

T2



Offset (Rt != 1111 && P == 1 && U == 0 && W == 0)

```
LDRSH{<c>}{<q>} <Rt>, [<Rn> {, #-<imm>}]
```

Post-indexed (P == 0 && W == 1)

```
LDRSH{<c>}{<q>} <Rt>, [<Rn>], #{+/-}<imm>
```

Pre-indexed (P == 1 && W == 1)

```
LDRSH{<c>}{<q>} <Rt>, [<Rn>, #{+/-}<imm>]!
```

```
if Rn == '1111' then SEE "LDRSH (literal)";
if Rt == '1111' && P == '1' && U == '0' && W == '0' then SEE "Related instructions";
if P == '1' && U == '1' && W == '0' then SEE "LDRSHT";
if P == '0' && W == '0' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
if (t == 15 && W == '1') || (wback && n == t) then UNPREDICTABLE;
// Armv8-A removes UNPREDICTABLE for R13
```

CONSTRAINED UNPREDICTABLE behavior

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Related instructions: *Load/store single*.

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<Rt> Is the general-purpose register to be transferred, encoded in the "Rt" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field. For PC use see *LDRSH (literal)*.

+/- Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

\+ Specifies the offset is added to the base register.

<imm> For encoding A1: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm4H:imm4L" field.

For encoding T1: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

For encoding T2: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm8" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    data = MemU[address,2];
    if wback then R[n] = offset_addr;
    R[t] = SignExtend(data, 32);
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

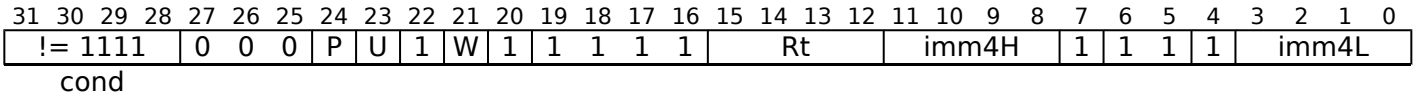
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDRSH (literal)

Load Register Signed Halfword (literal) calculates an address from the PC value and an immediate offset, loads a halfword from memory, sign-extends it to form a 32-bit word, and writes it to a register. For information about memory accesses see *Memory accesses*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



A1 (!(P == 0 && W == 1))

```
LDRSH{<c>}{<q>} <Rt>, <label> // (Normal form)

LDRSH{<c>}{<q>} <Rt>, [PC, #{+/-}<imm>] // (Alternative form)
```

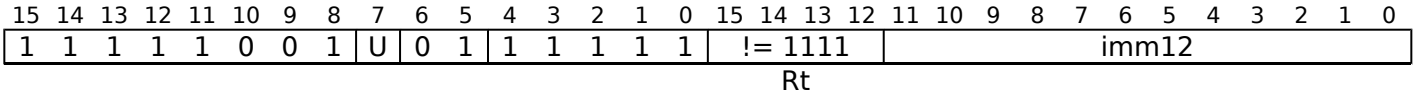
```
if P == '0' && W == '1' then SEE "LDRSHT";
t = UInt(Rt);  imm32 = ZeroExtend(imm4H:imm4L, 32);
add = (U == '1');  wback = (P == '0') || (W == '1');
if t == 15 || wback then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `wback`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes with the additional decode: wback = FALSE;.
- The instruction treats bit[24] as the P bit, and bit[21] as the writeback (W) bit, and uses the same addressing mode as described in *LDRSH (immediate)*. The instruction uses post-indexed addressing when P == '0' and uses pre-indexed addressing otherwise. The instruction is handled as described in *Using R15*.

T1



T1

```
LDRSH{<c>}{<q>} <Rt>, <label> // (Preferred syntax)

LDRSH{<c>}{<q>} <Rt>, [PC, #{+/-}<imm>] // (Alternative syntax)
```

```
if Rt == '1111' then SEE "Related instructions";
t = UInt(Rt);  imm32 = ZeroExtend(imm12, 32);  add = (U == '1');
// Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.
Related instructions: *Load, signed (literal)*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rt>	Is the general-purpose register to be transferred, encoded in the "Rt" field.

<label> For encoding A1: the label of the literal data item that is to be loaded into <Rt>. The assembler calculates the required value of the offset from the Align(PC, 4) value of the instruction to this label. Any value in the range -255 to 255 is permitted.

If the offset is zero or positive, imm32 is equal to the offset and add == TRUE, encoded as U == 1. If the offset is negative, imm32 is equal to minus the offset and add == FALSE, encoded as U == 0.

For encoding T1: the label of the literal data item that is to be loaded into <Rt>. The assembler calculates the required value of the offset from the Align(PC, 4) value of the instruction to this label. Permitted values of the offset are -4095 to 4095.

If the offset is zero or positive, imm32 is equal to the offset and add == TRUE, encoded as U == 1.

If the offset is negative, imm32 is equal to minus the offset and add == FALSE, encoded as U == 0.

+/- Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<imm> For encoding A1: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm4H:imm4L" field.

For encoding T1: is a 12-bit unsigned immediate byte offset, in the range 0 to 4095, encoded in the "imm12" field.

The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see *Use of labels in UAL instruction syntax*.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    base = Align(PC,4);
    address = if add then (base + imm32) else (base - imm32);
    data = MemU[address,2];
    R[t] = SignExtend(data, 32);
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

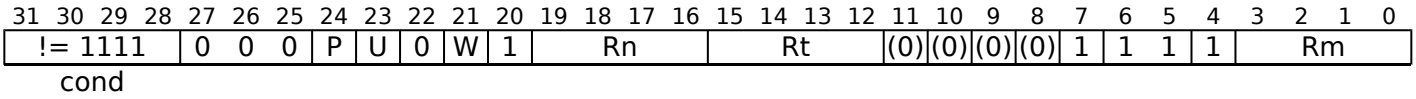
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDRSH (register)

Load Register Signed Halfword (register) calculates an address from a base register value and an offset register value, loads a halfword from memory, sign-extends it to form a 32-bit word, and writes it to a register. The offset register value can be shifted left by 0, 1, 2, or 3 bits. For information about memory accesses see *Memory accesses*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2) .

A1



Offset (P == 1 && W == 0)

```
LDRSH{<c>}{<q>} <Rt>, [<Rn>, {+/-}<Rm>]
```

Post-indexed (P == 0 && W == 0)

```
LDRSH{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>
```

Pre-indexed (P == 1 && W == 1)

```
LDRSH{<c>}{<q>} <Rt>, [<Rn>, {+/-}<Rm>]!
```

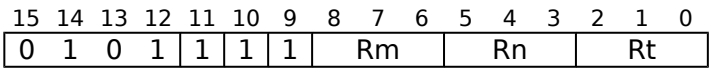
```
if P == '0' && W == '1' then SEE "LDRSHT";
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
(shift_t, shift_n) = (SRType_LSL, 0);
if t == 15 || m == 15 then UNPREDICTABLE;
if wback && (n == 15 || n == t) then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is <arm-defined-word>unknown</arm-defined-word>. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1

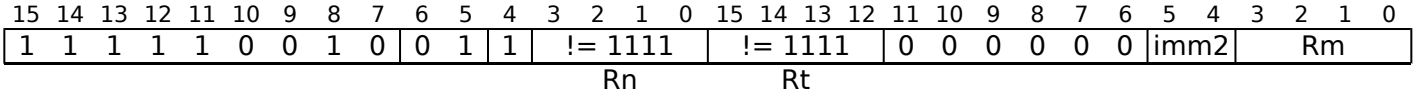


T1

```
LDRSH{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>]
```

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
```

T2



T2

```
LDRSH{<c>}.W <Rt>, [<Rn>, {+}<Rm>] // (<Rt>, <Rn>, <Rm> can be represented in T1)

LDRSH{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>{, LSL #<imm>}]
```

```
if Rn == '1111' then SEE "LDRSH (literal)";
if Rt == '1111' then SEE "Related instructions";
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, UInt(imm2));
if m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.
Related instructions: *Load/store, signed (register offset)*.

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<Rt> Is the general-purpose register to be transferred, encoded in the "Rt" field.

<Rn> For encoding A1: is the general-purpose base register, encoded in the "Rn" field. The PC can be used in the offset variant.

For encoding T1 and T2: is the general-purpose base register, encoded in the "Rn" field.

+/- Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

\+ Specifies the index register is added to the base register.

<Rm> Is the general-purpose index register, encoded in the "Rm" field.

<imm> If present, the size of the left shift to apply to the value from <Rm>, in the range 1-3. <imm> is encoded in imm2. If absent, no shift is specified and imm2 is encoded as 0b00.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset = Shift(R[m], shift_t, shift_n, PSTATE.C);
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if index then offset_addr else R[n];
    data = MemU[address,2];
    if wback then R[n] = offset_addr;
    R[t] = SignExtend(data, 32);
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDRSHT

Load Register Signed Halfword Unprivileged loads a halfword from memory, sign-extends it to form a 32-bit word, and writes it to a register. For information about memory accesses see *Memory accesses*.

The memory access is restricted as if the PE were running in User mode. This makes no difference if the PE is actually running in User mode.

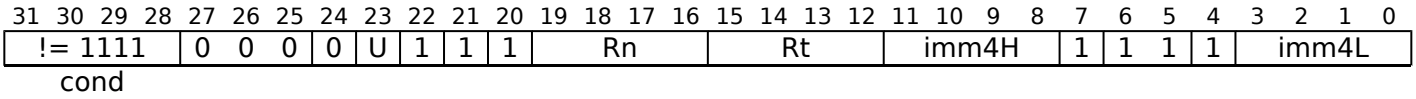
LDRSHT is UNPREDICTABLE in Hyp mode.

The T32 instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an immediate offset, and leaves the base register unchanged.

The A32 instruction uses a post-indexed addressing mode, that uses a base register value as the address for the memory access, and calculates a new address from a base register value and an offset and writes it back to the base register. The offset can be an immediate value or a register value.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1) .

A1



A1

```
LDRSHT{<c>}{<q>} <Rt>, [<Rn>] {, #{+/-}<imm>}

t = UInt(Rt);  n = UInt(Rn);  postindex = TRUE;  add = (U == '1');
register_form = FALSE;  imm32 = ZeroExtend(imm4H:imm4L, 32);
if t == 15 || n == 15 || n == t then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

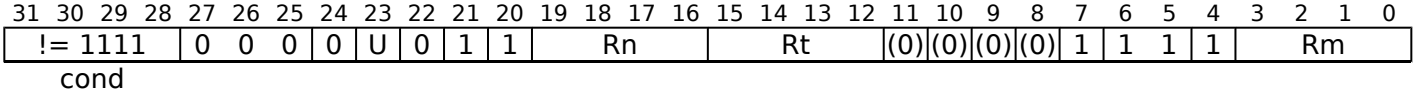
If `n == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction uses post-indexed addressing with the base register as PC. This is handled as described in *Using R15*.
- The instruction is treated as if bit[24] == '1' and bit[21] == '0'. The instruction uses immediate offset addressing with the base register as PC, without writeback.

If `n == t && n != 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

A2



A2

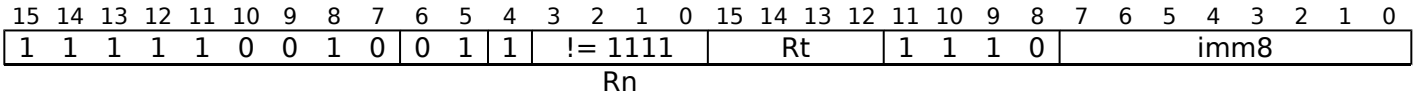
```
LDRSHT{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>

t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);  postindex = TRUE;  add = (U == '1');
register_form = TRUE;
if t == 15 || n == 15 || n == t || m == 15 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `n == t && n != 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1



T1

```
LDRSHT{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]

if Rn == '1111' then SEE "LDRSH (literal)";
t = UInt(Rt);  n = UInt(Rn);  postindex = FALSE;  add = TRUE;
register_form = FALSE;  imm32 = ZeroExtend(imm8, 32);
if t == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<Rt> Is the general-purpose register to be transferred, encoded in the "Rt" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

+/- For encoding A1: specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

For encoding A2: specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<Rm> Is the general-purpose index register, encoded in the "Rm" field.

\+ Specifies the offset is added to the base register.

<imm> For encoding A1: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm4H:imm4L" field.

For encoding T1: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field.

Operation

```
if ConditionPassed() then
    if PSTATE.EL == EL2 then UNPREDICTABLE;               // Hyp mode
    EncodingSpecificOperations();
    offset = if register_form then R[m] else imm32;
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if postindex then R[n] else offset_addr;
    data = MemU_unpriv[address,2];
    if postindex then R[n] = offset_addr;
    R[t] = SignExtend(data, 32);
```

CONSTRAINED UNPREDICTABLE behavior

If `PSTATE.EL == EL2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes as `LDRSH` (immediate).

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDRT

Load Register Unprivileged loads a word from memory, and writes it to a register. For information about memory accesses see *Memory accesses*.

The memory access is restricted as if the PE were running in User mode. This makes no difference if the PE is actually running in User mode.

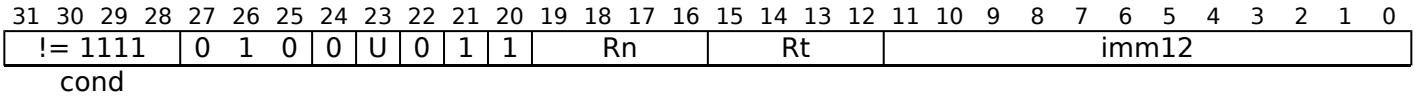
LDRT is UNPREDICTABLE in Hyp mode.

The T32 instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an immediate offset, and leaves the base register unchanged.

The A32 instruction uses a post-indexed addressing mode, that uses a base register value as the address for the memory access, and calculates a new address from a base register value and an offset and writes it back to the base register. The offset can be an immediate value or an optionally-shifted register value.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1) .

A1



A1

```
LDRT{<c>}{<q>} <Rt>, [<Rn>] {, #{+/-}<imm>}

t = UInt(Rt);  n = UInt(Rn);  postindex = TRUE;  add = (U == '1');
register_form = FALSE;  imm32 = ZeroExtend(imm12, 32);
if t == 15 || n == 15 || n == t then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

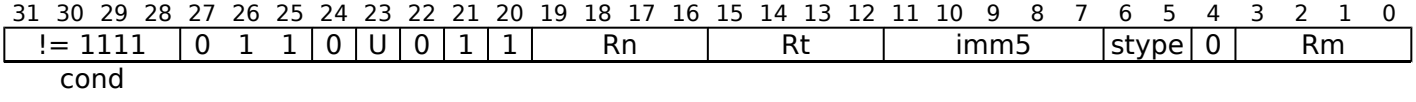
If `n == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction uses post-indexed addressing with the base register as PC. This is handled as described in *Using R15*.
- The instruction is treated as if bit[24] == '1' and bit[21] == '0'. The instruction uses immediate offset addressing with the base register as PC, without writeback.

If `n == t && n != 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

A2



A2

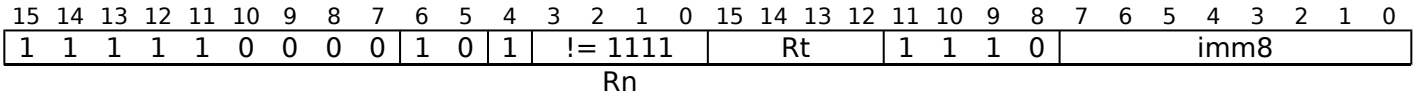
```
LDRT{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>{, <shift>}

t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);  postindex = TRUE;  add = (U == '1');
register_form = TRUE;  (shift_t, shift_n) = DecodeImmShift(stype, imm5);
if t == 15 || n == 15 || n == t || m == 15 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `n == t && n != 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such as instruction, the base address might be corrupted so that the instruction cannot be repeated.

T1



T1

```
LDRT{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]

if Rn == '1111' then SEE "LDR (literal)";
t = UInt(Rt);  n = UInt(Rn);  postindex = FALSE;  add = TRUE;
register_form = FALSE;  imm32 = ZeroExtend(imm8, 32);
if t == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<Rt> For encoding A1: is the general-purpose register to be transferred, encoded in the "Rt" field. The PC can be used, but this is deprecated.

For encoding A2 and T1: is the general-purpose register to be transferred, encoded in the "Rt" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

+/- For encoding A1: specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

For encoding A2: specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<Rm> Is the general-purpose index register, encoded in the "Rm" field.

<shift> The shift to apply to the value read from <Rm>. If absent, no shift is applied. Otherwise, see *Shifts applied to a register*.

+ Specifies the offset is added to the base register.

<imm> For encoding A1: is the 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 if omitted, and encoded in the "imm12" field.

For encoding T1: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field.

Operation

```
if ConditionPassed() then
    if PSTATE.EL == EL2 then UNPREDICTABLE;               // Hyp mode
    EncodingSpecificOperations();
    offset = if register_form then Shift(R[m], shift_t, shift_n, PSTATE.C) else imm32;
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if postindex then R[n] else offset_addr;
    data = MemU_unpriv[address,4];
    if postindex then R[n] = offset_addr;
    R[t] = data;
```

CONSTRAINED UNPREDICTABLE behavior

If `PSTATE.EL == EL2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes as `LDR` (immediate).

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LSL (immediate)

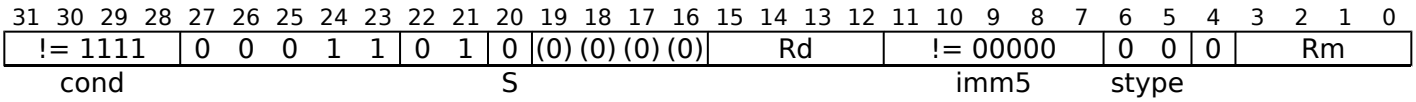
Logical Shift Left (immediate) shifts a register value left by an immediate number of bits, shifting in zeros, and writes the result to the destination register.

This is an alias of MOV, MOVS (register). This means:

- The encodings in this description are named to match the encodings of MOV, MOVS (register).
- The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T2 and T3) .

A1



MOV, shift or rotate by value

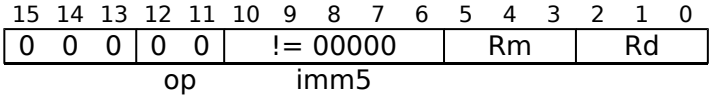
```
LSL{<c>}{<q>} {<Rd>,} <Rm>, #<imm>
```

is equivalent to

```
MOV{<c>}{<q>} <Rd>, <Rm>, LSL #<imm>
```

and is always the preferred disassembly.

T2



T2

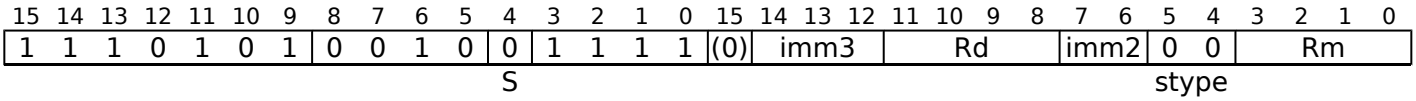
```
LSL<c>{<q>} {<Rd>,} <Rm>, #<imm> // (Inside IT block)
```

is equivalent to

```
MOV<c>{<q>} <Rd>, <Rm>, LSL #<imm>
```

and is the preferred disassembly when `InITBlock()`.

T3



MOV, shift or rotate by value

```
LSL<c>.W {<Rd>,} <Rm>, #<imm> // (Inside IT block, and <Rd>, <Rm>, <imm> can be represented in T2)
LSL{<c>}{<q>} {<Rd>,} <Rm>, #<imm>
```

is equivalent to

```
MOV{<c>}{<q>} <Rd>, <Rm>, LSL #<imm>
```

and is always the preferred disassembly.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rd>	For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. Arm deprecates using the PC as the destination register, but if the PC is used, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*.
	For encoding T2 and T3: is the general-purpose destination register, encoded in the "Rd" field.
<Rm>	For encoding A1: is the general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.
	For encoding T2 and T3: is the general-purpose source register, encoded in the "Rm" field.
<imm>	For encoding A1: is the shift amount, in the range 0 to 31, encoded in the "imm5" field as <imm> modulo 32.
	For encoding T2: is the shift amount, in the range 1 to 31, encoded in the "imm5" field as <amount> modulo 32.
	For encoding T3: is the shift amount, in the range 0 to 31, encoded in the "imm3:imm2" field as <imm> modulo 32.

Operation

The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LSL (register)

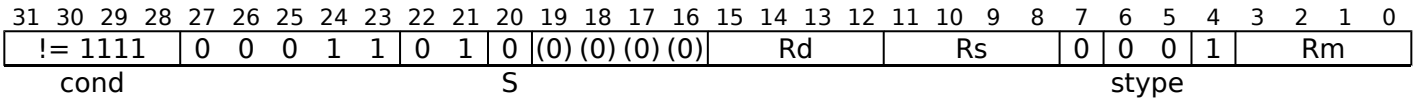
Logical Shift Left (register) shifts a register value left by a variable number of bits, shifting in zeros, and writes the result to the destination register. The variable number of bits is read from the bottom byte of a register

This is an alias of MOV, MOVS (register-shifted register). This means:

- The encodings in this description are named to match the encodings of MOV, MOVS (register-shifted register).
- The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2) .

A1



Not flag setting

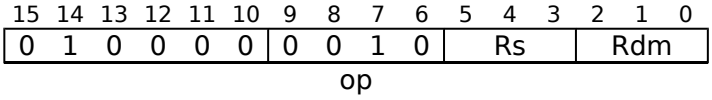
```
LSL{<c>}{<q>} {<Rd>,} <Rm>, <Rs>
```

is equivalent to

```
MOV{<c>}{<q>} <Rd>, <Rm>, LSL <Rs>
```

and is always the preferred disassembly.

T1



Logical shift left

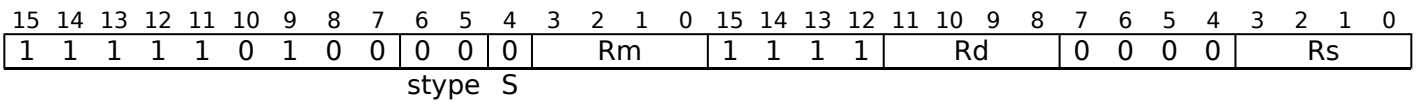
```
LSL<c>{<q>} {<Rdm>,} <Rdm>, <Rs> // (Inside IT block)
```

is equivalent to

```
MOV<c>{<q>} <Rdm>, <Rdm>, LSL <Rs>
```

and is the preferred disassembly when `InITBlock()`.

T2



Not flag setting

```
LSL<c>.W {<Rd>,} <Rm>, <Rs> // (Inside IT block, and <Rd>, <Rm>, <shift>, <Rs> can be represented in T1)
LSL{<c>}{<q>} {<Rd>,} <Rm>, <Rs>
```

is equivalent to

```
MOV{<c>}{<q>} <Rd>, <Rm>, LSL <Rs>
```

and is always the preferred disassembly.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rdm>	Is the first general-purpose source register and the destination register, encoded in the "Rdm" field.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rm>	Is the first general-purpose source register, encoded in the "Rm" field.
<Rs>	Is the second general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.

Operation

The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LSLS (immediate)

Logical Shift Left, setting flags (immediate) shifts a register value left by an immediate number of bits, shifting in zeros, and writes the result to the destination register.

If the destination register is not the PC, this instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

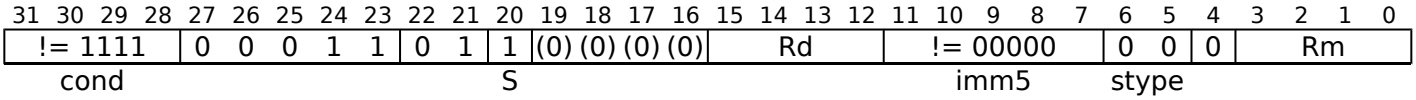
- The PE branches to the address written to the PC, and restores *PSTATE* from SPSR_<current_mode>.
- The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state*.
- The instruction is UNDEFINED in Hyp mode.
- The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

This is an alias of MOV, MOVS (register). This means:

- The encodings in this description are named to match the encodings of MOV, MOVS (register).
- The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T2 and T3) .

A1



MOVS, shift or rotate by value

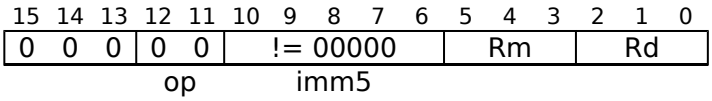
```
LSLS{<c>}{<q>} {<Rd>,} <Rm>, #<imm>
```

is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, LSL #<imm>
```

and is always the preferred disassembly.

T2



T2

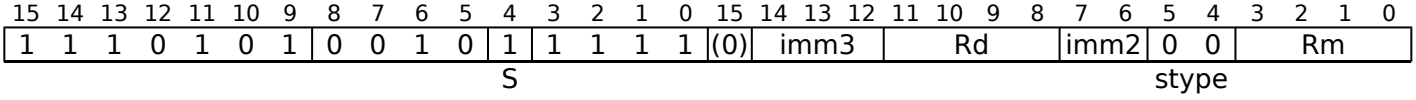
```
LSLS{<q>} {<Rd>,} <Rm>, #<imm> // (Outside IT block)
```

is equivalent to

```
MOVS{<q>} <Rd>, <Rm>, LSL #<imm>
```

and is the preferred disassembly when `!InITBlock()`.

T3



MOVS, shift or rotate by value

```
LSLS.W {<Rd>,} <Rm>, #<imm> // (Outside IT block, and <Rd>, <Rm>, <imm> can be represented in T2)

LSLS{<c>}{<q>} {<Rd>,} <Rm>, #<imm>
```

is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, LSL #<imm>
```

and is always the preferred disassembly.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rd>	For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. Arm deprecates using the PC as the destination register, but if the PC is used, the instruction performs an exception return, that restores *PSTATE* from SPSR_<current_mode>.
	For encoding T2 and T3: is the general-purpose destination register, encoded in the "Rd" field.
<Rm>	For encoding A1: is the general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.
	For encoding T2 and T3: is the general-purpose source register, encoded in the "Rm" field.
<imm>	For encoding A1: is the shift amount, in the range 0 to 31, encoded in the "imm5" field as <imm> modulo 32.
	For encoding T2: is the shift amount, in the range 1 to 31, encoded in the "imm5" field as <amount> modulo 32.
	For encoding T3: is the shift amount, in the range 0 to 31, encoded in the "imm3:imm2" field as <imm> modulo 32.

Operation

The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LSLS (register)

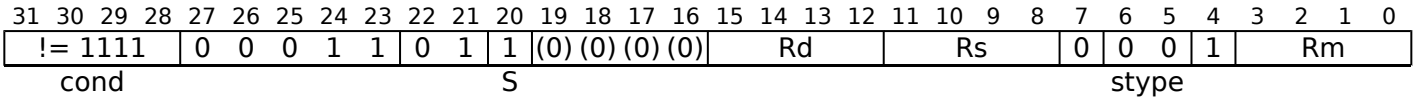
Logical Shift Left, setting flags (register) shifts a register value left by a variable number of bits, shifting in zeros, writes the result to the destination register, and updates the condition flags based on the result. The variable number of bits is read from the bottom byte of a register

This is an alias of MOV, MOVS (register-shifted register). This means:

- The encodings in this description are named to match the encodings of MOV, MOVS (register-shifted register).
- The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2) .

A1



Flag setting

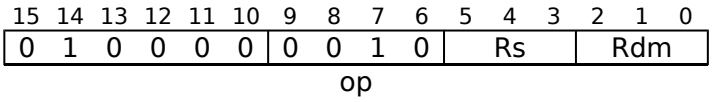
```
LSLS{<c>}{<q>} {<Rd>,} <Rm>, <Rs>
```

is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, LSL <Rs>
```

and is always the preferred disassembly.

T1



Logical shift left

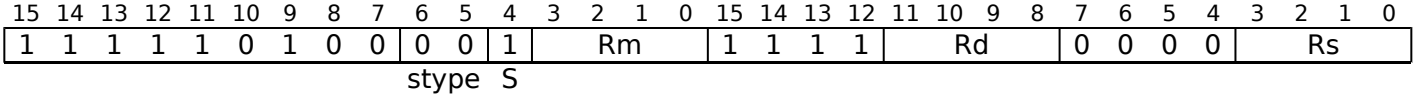
```
LSLS{<q>} {<Rdm>,} <Rdm>, <Rs> // (Outside IT block)
```

is equivalent to

```
MOVS{<q>} <Rdm>, <Rdm>, LSL <Rs>
```

and is the preferred disassembly when `!InITBlock()`.

T2



Flag setting

```
LSLS.W {<Rd>,} <Rm>, <Rs> // (Outside IT block, and <Rd>, <Rm>, <shift>, <Rs> can be represented in T1)
LSLS{<c>}{<q>} {<Rd>,} <Rm>, <Rs>
```

is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, LSL <Rs>
```

and is always the preferred disassembly.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rdm>	Is the first general-purpose source register and the destination register, encoded in the "Rdm" field.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rm>	Is the first general-purpose source register, encoded in the "Rm" field.
<Rs>	Is the second general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.

Operation

The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LSR (immediate)

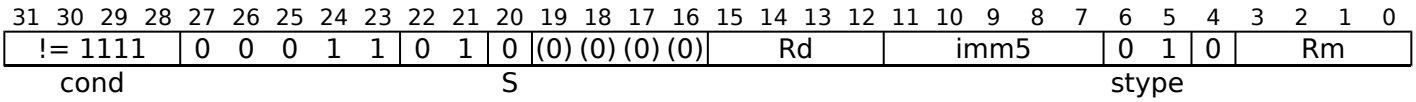
Logical Shift Right (immediate) shifts a register value right by an immediate number of bits, shifting in zeros, and writes the result to the destination register.

This is an alias of MOV, MOVS (register). This means:

- The encodings in this description are named to match the encodings of MOV, MOVS (register).
- The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T2 and T3) .

A1



MOV, shift or rotate by value

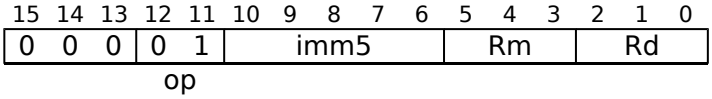
```
LSR{<c>}{<q>} {<Rd>,} <Rm>, #<imm>
```

is equivalent to

```
MOV{<c>}{<q>} <Rd>, <Rm>, LSR #<imm>
```

and is always the preferred disassembly.

T2



T2

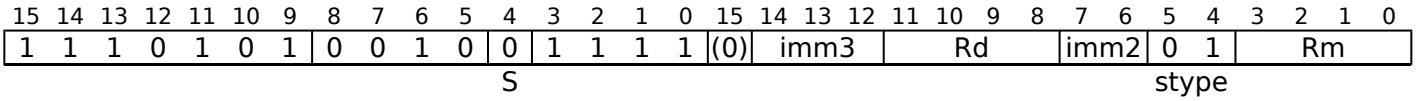
```
LSR<c>{<q>} {<Rd>,} <Rm>, #<imm> // (Inside IT block)
```

is equivalent to

```
MOV<c>{<q>} <Rd>, <Rm>, LSR #<imm>
```

and is the preferred disassembly when `InITBlock()`.

T3



MOV, shift or rotate by value

```
LSR<c>.W {<Rd>,} <Rm>, #<imm> // (Inside IT block, and <Rd>, <Rm>, <imm> can be represented in T2)
LSR{<c>}{<q>} {<Rd>,} <Rm>, #<imm>
```

is equivalent to

```
MOV{<c>}{<q>} <Rd>, <Rm>, LSR #<imm>
```

and is always the preferred disassembly.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rd>	For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. Arm deprecates using the PC as the destination register, but if the PC is used, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*.
	For encoding T2 and T3: is the general-purpose destination register, encoded in the "Rd" field.
<Rm>	For encoding A1: is the general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.
	For encoding T2 and T3: is the general-purpose source register, encoded in the "Rm" field.
<imm>	For encoding A1 and T2: is the shift amount, in the range 1 to 32, encoded in the "imm5" field as <imm> modulo 32.
	For encoding T3: is the shift amount, in the range 1 to 32, encoded in the "imm3:imm2" field as <imm> modulo 32.

Operation

The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LSR (register)

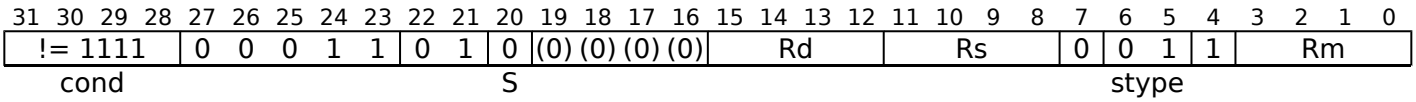
Logical Shift Right (register) shifts a register value right by a variable number of bits, shifting in zeros, and writes the result to the destination register. The variable number of bits is read from the bottom byte of a register

This is an alias of MOV, MOVS (register-shifted register). This means:

- The encodings in this description are named to match the encodings of MOV, MOVS (register-shifted register).
- The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2) .

A1



Not flag setting

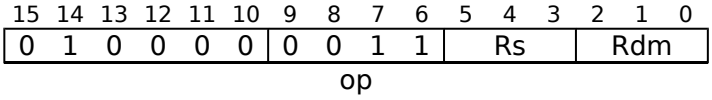
```
LSR{<c>}{<q>} {<Rd>,} <Rm>, <Rs>
```

is equivalent to

```
MOV{<c>}{<q>} <Rd>, <Rm>, LSR <Rs>
```

and is always the preferred disassembly.

T1



Logical shift right

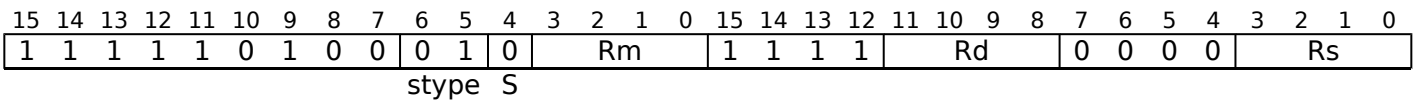
```
LSR<c>{<q>} {<Rdm>,} <Rdm>, <Rs> // (Inside IT block)
```

is equivalent to

```
MOV<c>{<q>} <Rdm>, <Rdm>, LSR <Rs>
```

and is the preferred disassembly when `InITBlock()`.

T2



Not flag setting

```
LSR<c>.W {<Rd>,} <Rm>, <Rs> // (Inside IT block, and <Rd>, <Rm>, <shift>, <Rs> can be represented in T1)
LSR{<c>}{<q>} {<Rd>,} <Rm>, <Rs>
```

is equivalent to

```
MOV{<c>}{<q>} <Rd>, <Rm>, LSR <Rs>
```

and is always the preferred disassembly.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rdm>	Is the first general-purpose source register and the destination register, encoded in the "Rdm" field.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rm>	Is the first general-purpose source register, encoded in the "Rm" field.
<Rs>	Is the second general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.

Operation

The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LSRS (immediate)

Logical Shift Right, setting flags (immediate) shifts a register value right by an immediate number of bits, shifting in zeros, and writes the result to the destination register.

If the destination register is not the PC, this instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

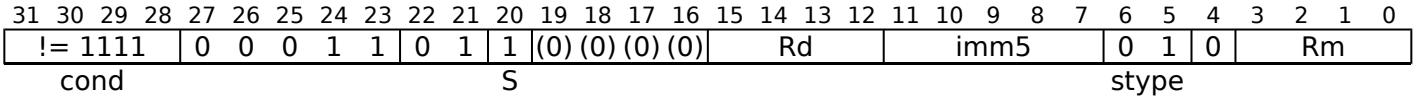
- The PE branches to the address written to the PC, and restores *PSTATE* from SPSR_<current_mode>.
- The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state*.
- The instruction is UNDEFINED in Hyp mode.
- The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

This is an alias of MOV, MOVS (register). This means:

- The encodings in this description are named to match the encodings of MOV, MOVS (register).
- The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T2 and T3) .

A1



MOVS, shift or rotate by value

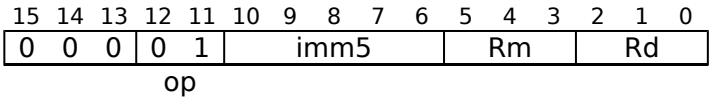
```
LSRS{<c>}{<q>} {<Rd>,} <Rm>, #<imm>
```

is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, LSR #<imm>
```

and is always the preferred disassembly.

T2



T2

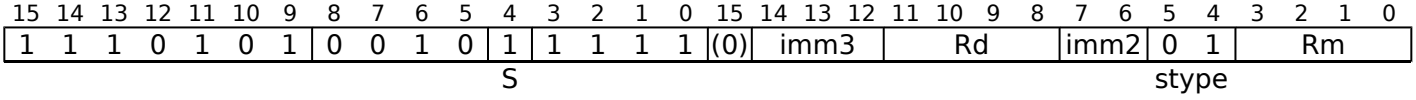
```
LSRS{<q>} {<Rd>,} <Rm>, #<imm> // (Outside IT block)
```

is equivalent to

```
MOVS{<q>} <Rd>, <Rm>, LSR #<imm>
```

and is the preferred disassembly when `!InITBlock()`.

T3



MOVS, shift or rotate by value

```
LSRS.W {<Rd>,} <Rm>, #<imm> // (Outside IT block, and <Rd>, <Rm>, <imm> can be represented in T2)

LSRS{<c>}{<q>} {<Rd>,} <Rm>, #<imm>
```

is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, LSR #<imm>
```

and is always the preferred disassembly.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rd>	For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. Arm deprecates using the PC as the destination register, but if the PC is used, the instruction performs an exception return, that restores *PSTATE* from SPSR_<current_mode>.
	For encoding T2 and T3: is the general-purpose destination register, encoded in the "Rd" field.
<Rm>	For encoding A1: is the general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.
	For encoding T2 and T3: is the general-purpose source register, encoded in the "Rm" field.
<imm>	For encoding A1 and T2: is the shift amount, in the range 1 to 32, encoded in the "imm5" field as <imm> modulo 32.
	For encoding T3: is the shift amount, in the range 1 to 32, encoded in the "imm3:imm2" field as <imm> modulo 32.

Operation

The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LSRS (register)

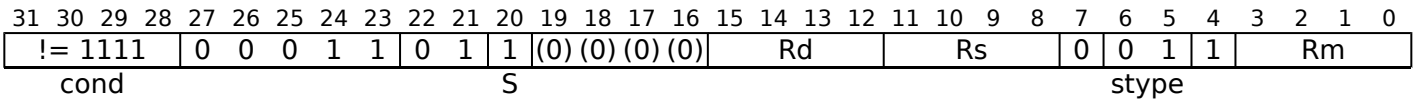
Logical Shift Right, setting flags (register) shifts a register value right by an immediate number of bits, shifting in zeros, writes the result to the destination register, and updates the condition flags based on the result. The variable number of bits is read from the bottom byte of a register

This is an alias of MOV, MOVS (register-shifted register). This means:

- The encodings in this description are named to match the encodings of MOV, MOVS (register-shifted register).
- The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2) .

A1



Flag setting

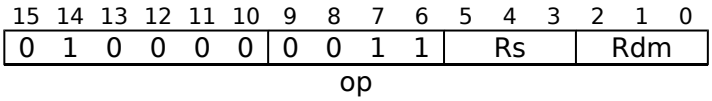
```
LSRS{<c>}{<q>} {<Rd>,} <Rm>, <Rs>
```

is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, LSR <Rs>
```

and is always the preferred disassembly.

T1



Logical shift right

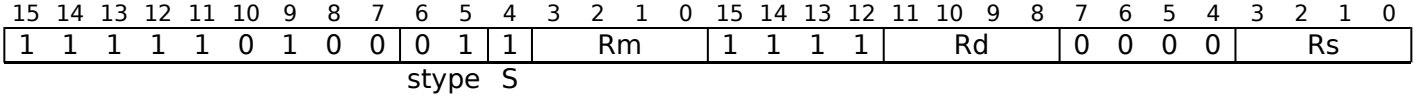
```
LSRS{<q>} {<Rdm>,} <Rdm>, <Rs> // (Outside IT block)
```

is equivalent to

```
MOVS{<q>} <Rdm>, <Rdm>, LSR <Rs>
```

and is the preferred disassembly when `!InITBlock()`.

T2



Flag setting

```
LSRS.W {<Rd>,} <Rm>, <Rs> // (Outside IT block, and <Rd>, <Rm>, <shift>, <Rs> can be represented in T1)
LSRS{<c>}{<q>} {<Rd>,} <Rm>, <Rs>
```

is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, LSR <Rs>
```

and is always the preferred disassembly.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rdm>	Is the first general-purpose source register and the destination register, encoded in the "Rdm" field.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rm>	Is the first general-purpose source register, encoded in the "Rm" field.
<Rs>	Is the second general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.

Operation

The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MCR

Move to System register from general-purpose register or execute a System instruction. This instruction copies the value of a general-purpose register to a System register, or executes a System instruction.

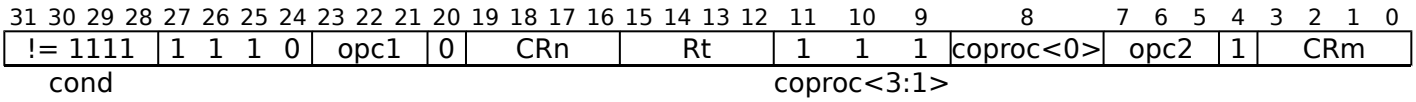
The System register and System instruction descriptions identify valid encodings for this instruction. Other encodings are UNDEFINED. For more information see *About the AArch32 System register interface* and *General behavior of System registers*.

In an implementation that includes EL2, MCR accesses to System registers can be trapped to Hyp mode, meaning that an attempt to execute an MCR instruction in a Non-secure mode other than Hyp mode, that would be permitted in the absence of the Hyp trap controls, generates a Hyp Trap exception. For more information, see *EL2 configurable instruction enables, disables, and traps*.

Because of the range of possible traps to Hyp mode, the MCR pseudocode does not show these possible traps.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

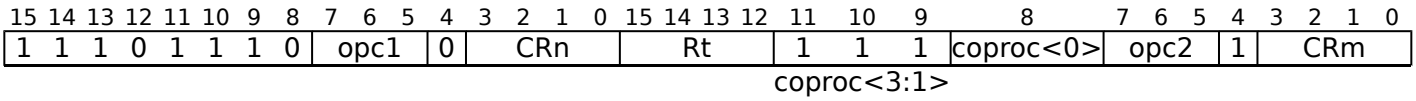
A1



A1

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

t = UInt(Rt);  cp = if coproc<0> == '0' then 14 else 15;
if t == 15  then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

T1



T1

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

t = UInt(Rt);  cp = if coproc<0> == '0' then 14 else 15;
if t == 15  then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<coproc> Is the System register encoding space, encoded in “coproc<0>”:

coproc<0>	<coproc>
0	p14
1	p15

<opc1> Is the opc1 parameter within the System register encoding space, in the range 0 to7, encoded in the "opc1" field.

<Rt> Is the general-purpose register to be transferred, encoded in the "Rt" field.

<CRn> Is the CRn parameter within the System register encoding space, in the range c0 to c15, encoded in the "CRn" field.

<CRm> Is the CRm parameter within the System register encoding space, in the range c0 to c15, encoded in the "CRm" field.

<opc2> Is the opc2 parameter within the System register encoding space, in the range 0 to7, encoded in the "opc2" field.

The possible values of { <coproc>, <opc1>, <CRn>, <CRm>, <opc2> } encode the entire System register and System instruction encoding space. Not all of this space is allocated, and the System register and System instruction descriptions identify the allocated encodings.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    AArch32.SysRegWrite(cp, ThisInstr(), R[t]);
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MCRR

Move to System register from two general-purpose registers. This instruction copies the values of two general-purpose registers to a System register.

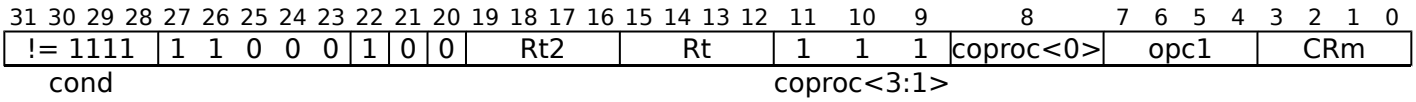
The System register descriptions identify valid encodings for this instruction. Other encodings are UNDEFINED. For more information see *About the AArch32 System register interface* and *General behavior of System registers*.

In an implementation that includes EL2, MCRR accesses to System registers can be trapped to Hyp mode, meaning that an attempt to execute an MCRR instruction in a Non-secure mode other than Hyp mode, that would be permitted in the absence of the Hyp trap controls, generates a Hyp Trap exception. For more information, see *EL2 configurable instruction enables, disables, and traps*.

Because of the range of possible traps to Hyp mode, the MCRR pseudocode does not show these possible traps.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

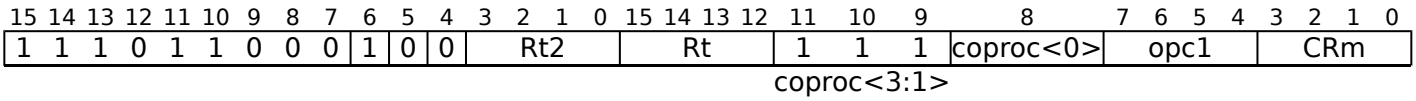
A1



A1

```
MCRR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>

t = UInt(Rt);  t2 = UInt(Rt2);  cp = if coproc<0> == '0' then 14 else 15;
if t == 15 || t2 == 15 then UNPREDICTABLE;
// Armv8-A removes UNPREDICTABLE for R13
```

T1



T1

```
MCRR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>

t = UInt(Rt);  t2 = UInt(Rt2);  cp = if coproc<0> == '0' then 14 else 15;
if t == 15 || t2 == 15 then UNPREDICTABLE;
// Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<coproc> Is the System register encoding space, encoded in “coproc<0>”:

coproc<0>	<coproc>
0	p14
1	p15

<opc1> Is the opc1 parameter within the System register encoding space, in the range 0 to 15, encoded in the "opc1" field.

<Rt> Is the first general-purpose register that is transferred into, encoded in the "Rt" field.

<Rt2> Is the second general-purpose register that is transferred into, encoded in the "Rt2" field.

<CRm> Is the CRm parameter within the System register encoding space, in the range c0 to c15, encoded in the "CRm" field.

The possible values of { <coproc>, <opc1>, <CRm> } encode the entire System register encoding space. Not all of this space is allocated, and the System register descriptions identify the allocated encodings.

For the permitted uses of these instructions, as described in this manual, <Rt2> transfers bits[63:32] of the selected System register, while <Rt> transfers bits[31:0].

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    value = R[t2]:R[t];
    AArch32.SysRegWrite64(cp, ThisInstr(), value);
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

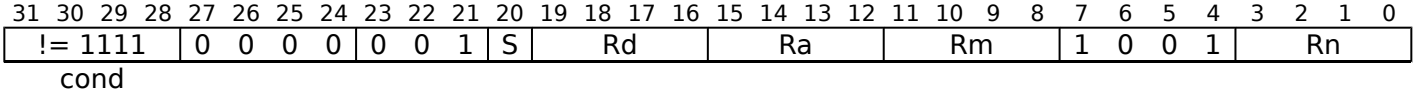
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MLA, MLAS

Multiply Accumulate multiplies two register values, and adds a third register value. The least significant 32 bits of the result are written to the destination register. These 32 bits do not depend on whether the source register values are considered to be signed values or unsigned values.
In an A32 instruction, the condition flags can optionally be updated based on the result. Use of this option adversely affects performance on many implementations.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



Flag setting (S == 1)

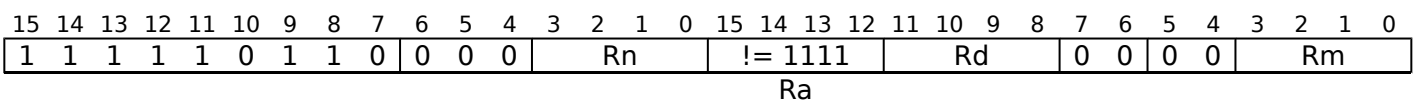
```
MLAS{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

Not flag setting (S == 0)

```
MLA{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);  setflags = (S == '1');
if d == 15 || n == 15 || m == 15 || a == 15 then UNPREDICTABLE;
```

T1



T1

```
MLA{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

```
if Ra == '1111' then SEE "MUL";
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);  setflags = FALSE;
if d == 15 || n == 15 || m  == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
<Ra>	Is the third general-purpose source register holding the addend, encoded in the "Ra" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand1 = SInt(R[n]);  // operand1 = UInt(R[n]) produces the same final results
    operand2 = SInt(R[m]);  // operand2 = UInt(R[m]) produces the same final results
    addend   = SInt(R[a]);  // addend   = UInt(R[a]) produces the same final results
    result = operand1 * operand2 + addend;
    R[d] = result<31:0>;
    if setflags then
        PSTATE.N = result<31>;
        PSTATE.Z = IsZeroBit(result<31:0>);
        // PSTATE.C, PSTATE.V unchanged
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

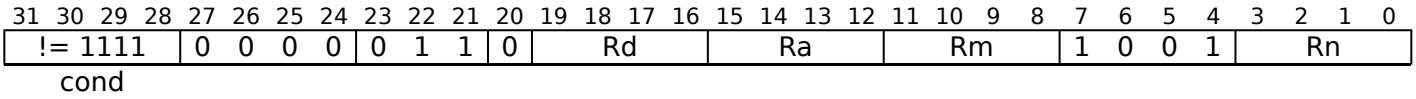
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MLS

Multiply and Subtract multiplies two register values, and subtracts the product from a third register value. The least significant 32 bits of the result are written to the destination register. These 32 bits do not depend on whether the source register values are considered to be signed values or unsigned values.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

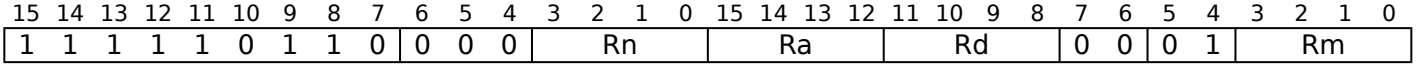
A1



A1

```
MLS{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);
if d == 15 || n == 15 || m == 15 || a == 15 then UNPREDICTABLE;
```

T1



T1

```
MLS{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);
if d == 15 || n == 15 || m == 15 || a == 15 then UNPREDICTABLE;
// Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
<Ra>	Is the third general-purpose source register holding the minuend, encoded in the "Ra" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand1 = SInt(R[n]);  // operand1 = UInt(R[n]) produces the same final results
    operand2 = SInt(R[m]);  // operand2 = UInt(R[m]) produces the same final results
    addend   = SInt(R[a]);  // addend   = UInt(R[a]) produces the same final results
    result = addend - operand1 * operand2;
    R[d] = result<31:0>;
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MOV, MOVS (immediate)

Move (immediate) writes an immediate value to the destination register.

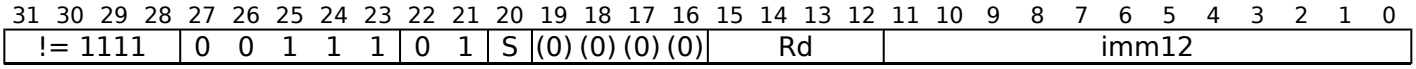
If the destination register is not the PC, the MOVS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

- The MOV variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*.
- The MOVS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores *PSTATE* from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state*.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 , T2 and T3) .

A1



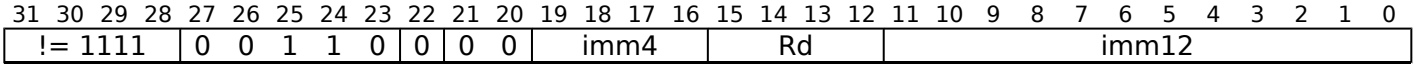
MOV (S == 0)

```
MOV{<c>}{<q>} <Rd>, #<const>
```

MOVS (S == 1)

```
MOVS{<c>}{<q>} <Rd>, #<const>

d = UInt(Rd);  setflags = (S == '1');  (imm32, carry) = A32ExpandImm_C(imm12, PSTATE.C);
```

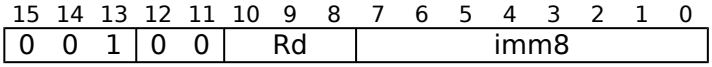
A2



A2

```
MOV{<c>}{<q>} <Rd>, #<imm16> // (<imm16> can not be represented in A1)

MOVW{<c>}{<q>} <Rd>, #<imm16> // (<imm16> can be represented in A1)

d = UInt(Rd);  setflags = FALSE;  imm32 = ZeroExtend(imm4:imm12, 32);
if d == 15 then UNPREDICTABLE;
```

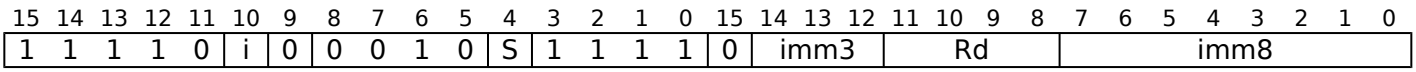
T1



T1

```
MOV<c>{<q>} <Rd>, #<imm8> // (Inside IT block)

MOVS{<q>} <Rd>, #<imm8> // (Outside IT block)

d = UInt(Rd);  setflags = !InITBlock();  imm32 = ZeroExtend(imm8, 32);  carry = PSTATE.C;
```

T2



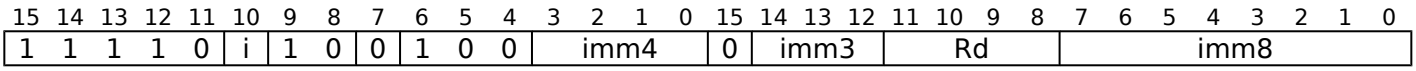
MOV (S == 0)

```
MOV<c>.W <Rd>, #<const> // (Inside IT block, and <Rd>, <const> can be represented in T1)

MOV{<c>}{<q>} <Rd>, #<const>
```

MOVS (S == 1)

```
MOVS.W <Rd>, #<const> // (Outside IT block, and <Rd>, <const> can be represented in T1)

MOVS{<c>}{<q>} <Rd>, #<const>

d = UInt(Rd);  setflags = (S == '1');  (imm32, carry) = T32ExpandImm_C(i:imm3:imm8, PSTATE.C);
if d == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

T3



T3

```
MOV{<c>}{<q>} <Rd>, #<imm16> // (<imm16> cannot be represented in T1 or T2)

MOVW{<c>}{<q>} <Rd>, #<imm16> // (<imm16> can be represented in T1 or T2)

d = UInt(Rd);  setflags = FALSE;  imm32 = ZeroExtend(imm4:i:imm3:imm8, 32);
if d == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rd>	For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. Arm deprecates using the PC as the destination register, but if the PC is used: • For the MOV variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*. • For the MOVS variant, the instruction performs an exception return, that restores *PSTATE* from SPSR_<current_mode>. For encoding A2, T1, T2 and T3: is the general-purpose destination register, encoded in the "Rd" field.
<imm8>	Is a 8-bit unsigned immediate, in the range 0 to 255, encoded in the "imm8" field.
<imm16>	For encoding A2: is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm4:imm12" field.

For encoding T3: is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm4:i:imm3:imm8" field.

<const> For encoding A1: an immediate value. See *Modified immediate constants in A32 instructions* for the range of values.

For encoding T2: an immediate value. See *Modified immediate constants in T32 instructions* for the range of values.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = imm32;
    if d == 15 then          // Can only occur for encoding A1
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.N = result<31>;
            PSTATE.Z = IsZeroBit(result);
            PSTATE.C = carry;
            // PSTATE.V unchanged
```

Operational information

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MOV, MOVS (register)

Move (register) copies a value from a register to the destination register.

If the destination register is not the PC, the MOVS variant of the instruction updates the condition flags based on the result.

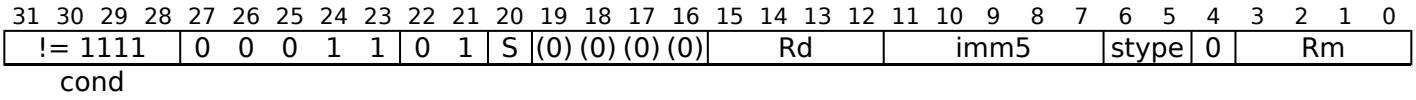
The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. If the destination register is the PC:

- The MOV variant of the instruction is a branch. In the T32 instruction set (encoding T1) this is a simple branch, and in the A32 instruction set it is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*.
- The MOVS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores *PSTATE* from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state*.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

This instruction is used by the aliases ASRS (immediate), ASR (immediate), LSLS (immediate), LSL (immediate), LSRS (immediate), LSR (immediate), RORS (immediate), ROR (immediate), RRXS, and RRX.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 , T2 and T3) .

A1



MOV, rotate right with extend (S == 0 && imm5 == 00000 && stype == 11)

```
MOV{<c>}{<q>} <Rd>, <Rm>, RRX
```

MOV, shift or rotate by value (S == 0 && !(imm5 == 00000 && stype == 11))

```
MOV{<c>}{<q>} <Rd>, <Rm> {, <shift> #<amount>}
```

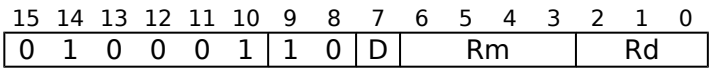
MOVS, rotate right with extend (S == 1 && imm5 == 00000 && stype == 11)

```
MOVS{<c>}{<q>} <Rd>, <Rm>, RRX
```

MOVS, shift or rotate by value (S == 1 && !(imm5 == 00000 && stype == 11))

```
MOVS{<c>}{<q>} <Rd>, <Rm> {, <shift> #<amount>}

d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm5);
```

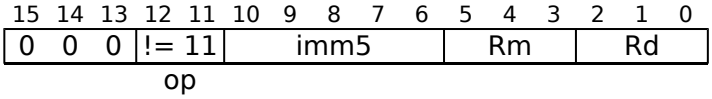
T1



T1

```
MOV{<c>}{<q>} <Rd>, <Rm>

d = UInt(D:Rd);  m = UInt(Rm);  setflags = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
if d == 15 && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

T2



T2

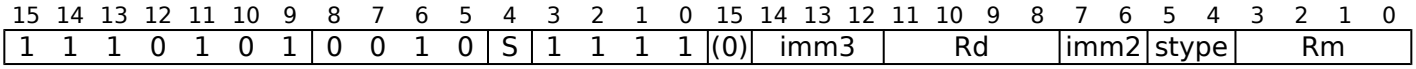
```
MOV<c>{<q>} <Rd>, <Rm> {, <shift> #<amount>} // (Inside IT block)

MOVS{<q>} <Rd>, <Rm> {, <shift> #<amount>} // (Outside IT block)

d = UInt(Rd);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = DecodeImmShift(op, imm5);
if op == '00' && imm5 == '00000' && InITBlock() then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `op == '00' && imm5 == '00000' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passed its condition code check.
- The instruction executes as NOP, as if it failed its condition code check.
- The instruction executes as MOV Rd, Rm.

T3



MOV, rotate right with extend (S == 0 && imm3 == 000 && imm2 == 00 && stype == 11)

```
MOV{<c>}{<q>} <Rd>, <Rm>, RRX
```

MOV, shift or rotate by value (S == 0 && !(imm3 == 000 && imm2 == 00 && stype == 11))

```
MOV{<c>}.W <Rd>, <Rm> {, LSL #0} // (<Rd>, <Rm> can be represented in T1)

MOV<c>.W <Rd>, <Rm> {, <shift> #<amount>} // (Inside IT block, and <Rd>, <Rm>, <shift>, <amount> can be r

MOV{<c>}{<q>} <Rd>, <Rm> {, <shift> #<amount>}
```

MOVS, rotate right with extend (S == 1 && imm3 == 000 && imm2 == 00 && stype == 11)

```
MOVS{<c>}{<q>} <Rd>, <Rm>, RRX
```

MOVS, shift or rotate by value (S == 1 && !(imm3 == 000 && imm2 == 00 && stype == 11))

```
MOVS.W <Rd>, <Rm> {, <shift> #<amount>} // (Outside IT block, and <Rd>, <Rm>, <shift>, <amount> can be re

MOVS{<c>}{<q>} <Rd>, <Rm> {, <shift> #<amount>}

d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm3:imm2);
if d == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<Rd> For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If the PC is used:

- For the MOV variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*. Arm deprecates use of the instruction if <Rn> is the PC.
- For the MOVS variant, the instruction performs an exception return, that restores *PSTATE* from SPSR_<current_mode>. Arm deprecates use of the instruction if <Rn> is not the LR, or if the optional shift or RRX argument is specified.

For encoding T1: is the general-purpose destination register, encoded in the "D:Rd" field. If the PC is used:

- The instruction causes a branch to the address moved to the PC. This is a simple branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*.
- The instruction must either be outside an IT block or the last instruction of an IT block.

For encoding T2 and T3: is the general-purpose destination register, encoded in the "Rd" field.

<Rm> For encoding A1 and T1: is the general-purpose source register, encoded in the "Rm" field. The PC can be used. Arm deprecates use of the instruction if <Rd> is the PC.

For encoding T2 and T3: is the general-purpose source register, encoded in the "Rm" field.

<shift> For encoding A1 and T3: is the type of shift to be applied to the source register, encoded in “stype”:

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

For encoding T2: is the type of shift to be applied to the source register, encoded in “op”:

op	<shift>
00	LSL
01	LSR
10	ASR

<amount> For encoding A1: is the shift amount, in the range 0 to 31 (when <shift> = LSL), or 1 to 31 (when <shift> = ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the "imm5" field as <amount> modulo 32.

For encoding T2: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the "imm5" field as <amount> modulo 32.

For encoding T3: is the shift amount, in the range 0 to 31 (when <shift> = LSL) or 1 to 31 (when <shift> = ROR), or 1 to 32 (when <shift> = LSR or ASR), encoded in the "imm3:imm2" field as <amount> modulo 32.

Alias Conditions

Alias	Of variant	Is preferred when
ASRS (immediate)	T3 (MOVS, shift or rotate by value), A1 (MOVS, shift or rotate by value)	`S == '1' && stype == '10'`
ASRS (immediate)	T2	`op == '10' && !InITBlock()`
ASR (immediate)	T3 (MOV, shift or rotate by value), A1 (MOV, shift or rotate by value)	`S == '0' && stype == '10'`
ASR (immediate)	T2	`op == '10' && InITBlock()`

Alias	Of variant	Is preferred when
LSLS (immediate)	T3 (MOVS, shift or rotate by value)	`S == '1' && imm3:Rd:imm2 != '000xxxx00' && stype == '00'`
LSLS (immediate)	A1 (MOVS, shift or rotate by value)	`S == '1' && imm5 != '00000' && stype == '00'`
LSLS (immediate)	T2	`op == '00' && imm5 != '00000' && !InITBlock()`
LSL (immediate)	T3 (MOV, shift or rotate by value)	`S == '0' && imm3:Rd:imm2 != '000xxxx00' && stype == '00'`
LSL (immediate)	A1 (MOV, shift or rotate by value)	`S == '0' && imm5 != '00000' && stype == '00'`
LSL (immediate)	T2	`op == '00' && imm5 != '00000' && InITBlock()`
LSRS (immediate)	T3 (MOVS, shift or rotate by value), A1 (MOVS, shift or rotate by value)	`S == '1' && stype == '01'`
LSRS (immediate)	T2	`op == '01' && !InITBlock()`
LSR (immediate)	T3 (MOV, shift or rotate by value), A1 (MOV, shift or rotate by value)	`S == '0' && stype == '01'`
LSR (immediate)	T2	`op == '01' && InITBlock()`
RORS (immediate)	T3 (MOVS, shift or rotate by value)	`S == '1' && imm3:Rd:imm2 != '000xxxx00' && stype == '11'`
RORS (immediate)	A1 (MOVS, shift or rotate by value)	`S == '1' && imm5 != '00000' && stype == '11'`
ROR (immediate)	T3 (MOV, shift or rotate by value)	`S == '0' && imm3:Rd:imm2 != '000xxxx00' && stype == '11'`
ROR (immediate)	A1 (MOV, shift or rotate by value)	`S == '0' && imm5 != '00000' && stype == '11'`
RRXS	T3 (MOVS, rotate right with extend)	`S == '1' && imm3 == '000' && imm2 == '00' && stype == '11'`
RRXS	A1 (MOVS, rotate right with extend)	`S == '1' && imm5 == '00000' && stype == '11'`
RRX	T3 (MOV, rotate right with extend)	`S == '0' && imm3 == '000' && imm2 == '00' && stype == '11'`
RRX	A1 (MOV, rotate right with extend)	`S == '0' && imm5 == '00000' && stype == '11'`

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
    result = shifted;
    if d == 15 then
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.N = result<31>;
            PSTATE.Z = IsZeroBit(result);
            PSTATE.C = carry;
            // PSTATE.V unchanged
```

Operational information

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

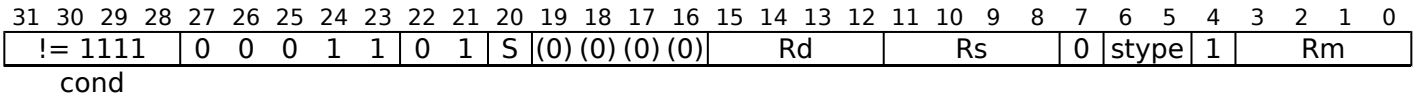
MOV, MOVS (register-shifted register)

Move (register-shifted register) copies a register-shifted register value to the destination register. It can optionally update the condition flags based on the value.

This instruction is used by the aliases ASRS (register), ASR (register), LSLS (register), LSL (register), LSRS (register), LSR (register), RORS (register), and ROR (register).

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2) .

A1



Flag setting (S == 1)

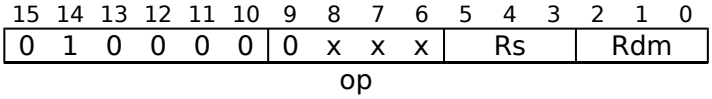
```
MOVS{<c>}{<q>} <Rd>, <Rm>, <shift> <Rs>
```

Not flag setting (S == 0)

```
MOV{<c>}{<q>} <Rd>, <Rm>, <shift> <Rs>
```

```
d = UInt(Rd);  m = UInt(Rm);  s = UInt(Rs);
setflags = (S == '1');  shift_t = DecodeRegShift(stype);
if d == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```

T1



Arithmetic shift right (op == 0100)

```
MOV<c>{<q>} <Rdm>, <Rdm>, ASR <Rs> // (Inside IT block)

MOVS{<q>} <Rdm>, <Rdm>, ASR <Rs> // (Outside IT block)
```

Logical shift left (op == 0010)

```
MOV<c>{<q>} <Rdm>, <Rdm>, LSL <Rs> // (Inside IT block)

MOVS{<q>} <Rdm>, <Rdm>, LSL <Rs> // (Outside IT block)
```

Logical shift right (op == 0011)

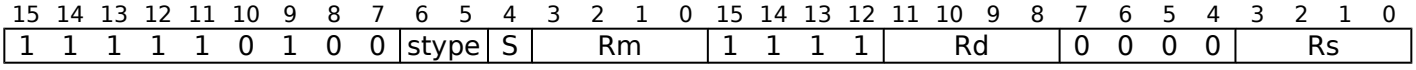
```
MOV<c>{<q>} <Rdm>, <Rdm>, LSR <Rs> // (Inside IT block)

MOVS{<q>} <Rdm>, <Rdm>, LSR <Rs> // (Outside IT block)
```

Rotate right (op == 0111)

```
MOV<c>{<q>} <Rdm>, <Rdm>, ROR <Rs> // (Inside IT block)

MOVS{<q>} <Rdm>, <Rdm>, ROR <Rs> // (Outside IT block)
```

```
if !(op IN {'0010', '0011', '0100', '0111'}) then SEE "Related encodings";
d = UInt(Rdm);  m = UInt(Rdm);  s = UInt(Rs);
setflags = !InITBlock();  shift_t = DecodeRegShift(op<2>:op<0>);
```

T2



Flag setting (S == 1)

```
MOVS.W <Rd>, <Rm>, <shift> <Rs> // (Outside IT block, and <Rd>, <Rm>, <shift>, <Rs> can be represented in

MOVS{<c>}{<q>} <Rd>, <Rm>, <shift> <Rs>
```

Not flag setting (S == 0)

```
MOV<c>.W <Rd>, <Rm>, <shift> <Rs> // (Inside IT block, and <Rd>, <Rm>, <shift>, <Rs> can be represented i

MOV{<c>}{<q>} <Rd>, <Rm>, <shift> <Rs>
```

```
d = UInt(Rd);  m = UInt(Rm);  s = UInt(Rs);
setflags = (S == '1');  shift_t = DecodeRegShift(stype);
if d == 15 || m == 15 || s == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Related encodings: In encoding T1, for an op field value that is not described above, see *Data-processing (two low registers)*.
For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rdm>	Is the general-purpose source register and the destination register, encoded in the "Rdm" field.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rm>	Is the general-purpose source register, encoded in the "Rm" field.

<shift> Is the type of shift to be applied to the second source register, encoded in “stype”:

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<Rs> Is the general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.

Alias Conditions

Alias	Of variant	Is preferred when
ASRS (register)	A1 (flag setting)	`S == '1' && stype == '10'`
ASRS (register)	T1 (arithmetic shift right)	`op == '0100' && !InITBlock()`
ASRS (register)	T2 (flag setting)	`stype == '10' && S == '1'`
ASR (register)	A1 (not flag setting)	`S == '0' && stype == '10'`
ASR (register)	T1 (arithmetic shift right)	`op == '0100' && InITBlock()`
ASR (register)	T2 (not flag setting)	`stype == '10' && S == '0'`
LSLS (register)	A1 (flag setting)	`S == '1' && stype == '00'`
LSLS (register)	T1 (logical shift left)	`op == '0010' && !InITBlock()`
LSLS (register)	T2 (flag setting)	`stype == '00' && S == '1'`
LSL (register)	A1 (not flag setting)	`S == '0' && stype == '00'`
LSL (register)	T1 (logical shift left)	`op == '0010' && InITBlock()`
LSL (register)	T2 (not flag setting)	`stype == '00' && S == '0'`
LSRS (register)	A1 (flag setting)	`S == '1' && stype == '01'`
LSRS (register)	T1 (logical shift right)	`op == '0011' && !InITBlock()`
LSRS (register)	T2 (flag setting)	`stype == '01' && S == '1'`
LSR (register)	A1 (not flag setting)	`S == '0' && stype == '01'`
LSR (register)	T1 (logical shift right)	`op == '0011' && InITBlock()`
LSR (register)	T2 (not flag setting)	`stype == '01' && S == '0'`
RORS (register)	A1 (flag setting)	`S == '1' && stype == '11'`
RORS (register)	T1 (rotate right)	`op == '0111' && !InITBlock()`
RORS (register)	T2 (flag setting)	`stype == '11' && S == '1'`
ROR (register)	A1 (not flag setting)	`S == '0' && stype == '11'`
ROR (register)	T1 (rotate right)	`op == '0111' && InITBlock()`
ROR (register)	T2 (not flag setting)	`stype == '11' && S == '0'`

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[s]<7:0>);
    (result, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
    R[d] = result;
    if setflags then
        PSTATE.N = result<31>;
        PSTATE.Z = IsZeroBit(result);
        PSTATE.C = carry;
        // PSTATE.V unchanged
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:

    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

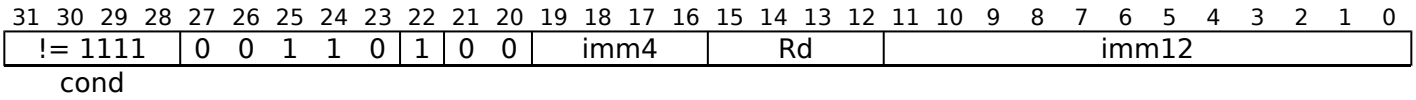
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MOVT

Move Top writes an immediate value to the top halfword of the destination register. It does not affect the contents of the bottom halfword.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

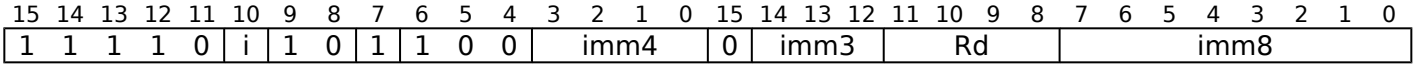


A1

```
MOVT{<c>}{<q>} <Rd>, #<imm16>
```

```
d = UInt(Rd);  imm16 = imm4:imm12;
if d == 15 then UNPREDICTABLE;
```

T1



T1

```
MOVT{<c>}{<q>} <Rd>, #<imm16>
```

```
d = UInt(Rd);  imm16 = imm4:i:imm3:imm8;
if d == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<imm16>	For encoding A1: is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm4:imm12" field. For encoding T1: is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm4:i:imm3:imm8" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    R[d]<31:16> = imm16;
    // R[d]<15:0> unchanged
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MRC

Move to general-purpose register from System register. This instruction copies the value of a System register to a general-purpose register.

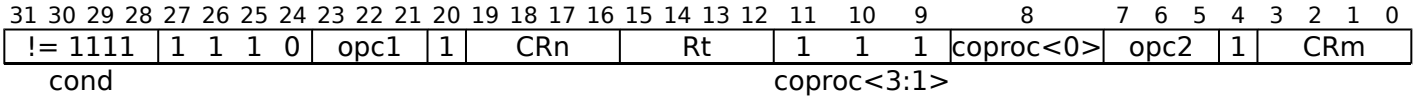
The System register descriptions identify valid encodings for this instruction. Other encodings are UNDEFINED. For more information see *About the AArch32 System register interface* and *General behavior of System registers*.

In an implementation that includes EL2, MRC accesses to system control registers can be trapped to Hyp mode, meaning that an attempt to execute an MRC instruction in a Non-secure mode other than Hyp mode, that would be permitted in the absence of the Hyp trap controls, generates a Hyp Trap exception. For more information, see *EL2 configurable instruction enables, disables, and traps*.

Because of the range of possible traps to Hyp mode, the MRC pseudocode does not show these possible traps.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

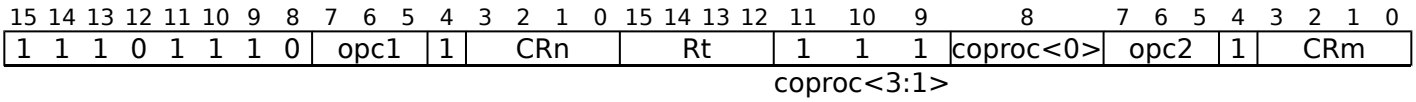
A1



A1

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

t = UInt(Rt);  cp = if coproc<0> == '0' then 14 else 15;
// Armv8-A removes UNPREDICTABLE for R13
```

T1



T1

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

t = UInt(Rt);  cp = if coproc<0> == '0' then 14 else 15;
// Armv8-A removes UNPREDICTABLE for R13
```

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<coproc> Is the System register encoding space, encoded in “coproc<0>”:

coproc<0>	<coproc>
0	p14
1	p15

<opc1> Is the opc1 parameter within the System register encoding space, in the range 0 to7, encoded in the "opc1" field.

<Rt> Is the general-purpose register to be transferred or APSR_nzcv (encoded as 0b1111), encoded in the "Rt" field. If APSR_nzcv is used, bits [31:28] of the transferred value are written to the *PSTATE* condition flags.

<CRn> Is the CRn parameter within the System register encoding space, in the range c0 to c15, encoded in the "CRn" field.

<CRm> Is the CRm parameter within the System register encoding space, in the range c0 to c15, encoded in the "CRm" field.

<opc2> Is the opc2 parameter within the System register encoding space, in the range 0 to7, encoded in the "opc2" field.

The possible values of { <coproc>, <opc1>, <CRn>, <CRm>, <opc2> } encode the entire System register and System instruction encoding space. Not all of this space is allocated, and the System register and System instruction descriptions identify the allocated encodings.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    bits(32) value = AArch32.SysRegRead(cp, ThisInstr());
    if t != 15 then
        R[t] = value;
    elsif AArch32.SysRegReadCanWriteAPSR(cp, ThisInstr()) then
        PSTATE.<N,Z,C,V> = value<31:28>;
        // value<27:0> are not used.
    else
        UNPREDICTABLE;
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MRRC

Move to two general-purpose registers from System register. This instruction copies the value of a System register to two general-purpose registers.

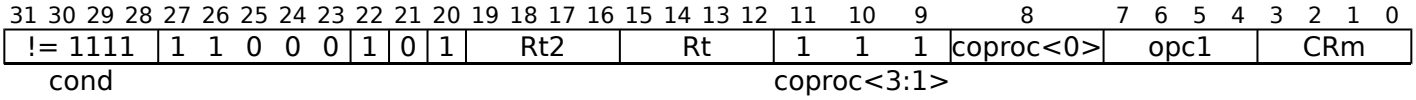
The System register descriptions identify valid encodings for this instruction. Other encodings are UNDEFINED. For more information see *About the AArch32 System register interface* and *General behavior of System registers*.

In an implementation that includes EL2, MRRC accesses to System registers can be trapped to Hyp mode, meaning that an attempt to execute an MRRC instruction in a Non-secure mode other than Hyp mode, that would be permitted in the absence of the Hyp trap controls, generates a Hyp Trap exception. For more information, see *EL2 configurable instruction enables, disables, and traps*.

Because of the range of possible traps to Hyp mode, the MRRC pseudocode does not show these possible traps.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



A1

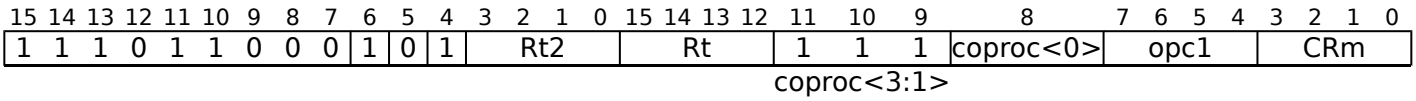
```
MRRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>

t = UInt(Rt);  t2 = UInt(Rt2);  cp = if coproc<0> == '0' then 14 else 15;
if t == 15 || t2 == 15 || t == t2 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

CONSTRAINED UNPREDICTABLE behavior

If `t == t2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

T1



T1

```
MRRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>

t = UInt(Rt);  t2 = UInt(Rt2);  cp = if coproc<0> == '0' then 14 else 15;
if t == 15 || t2 == 15 || t == t2 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

CONSTRAINED UNPREDICTABLE behavior

If `t == t2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<coproc> Is the System register encoding space, encoded in “coproc<0>”:

coproc<0>	<coproc>
0	p14
1	p15

<opc1> Is the opc1 parameter within the System register encoding space, in the range 0 to 15, encoded in the "opc1" field.

<Rt> Is the first general-purpose register that is transferred into, encoded in the "Rt" field.

<Rt2> Is the second general-purpose register that is transferred into, encoded in the "Rt2" field.

<CRm> Is the CRm parameter within the System register encoding space, in the range c0 to c15, encoded in the "CRm" field.

The possible values of { <coproc>, <opc1>, <CRm> } encode the entire System register encoding space. Not all of this space is allocated, and the System register descriptions identify the allocated encodings.

For the permitted uses of these instructions, as described in this manual, <Rt2> transfers bits[63:32] of the selected System register, while <Rt> transfers bits[31:0].

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    value = AArch32.SysRegRead64(cp, ThisInstr());
    R[t] = value<31:0>;
    R[t2] = value<63:32>;
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MRS

Move Special register to general-purpose register moves the value of the *APSR*, *CPSR*, or *SPSR*_<current_mode> into a general-purpose register.

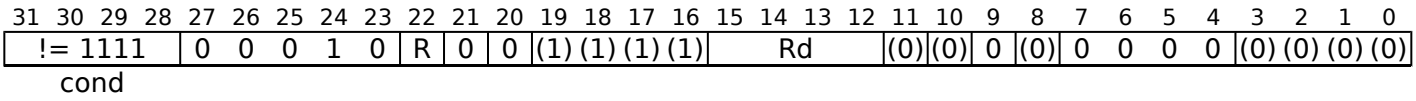
Arm recommends the APSR form when only the N, Z, C, V, Q, and GE[3:0] bits are being written. For more information, see *APSR*.

An MRS that accesses the *SPSRs* is UNPREDICTABLE if executed in User mode or System mode.

An MRS that is executed in User mode and accesses the *CPSR* returns an UNKNOWN value for the *CPSR*.{E, A, I, F, M} fields.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

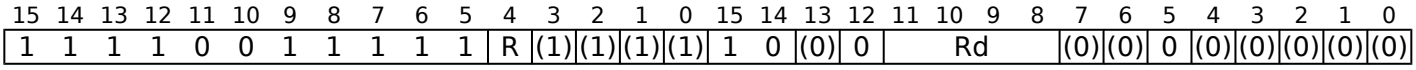
A1



A1

```
MRS{<c>}{<q>} <Rd>, <spec_reg>

d = UInt(Rd);  read_spsr = (R == '1');
if d == 15 then UNPREDICTABLE;
```

T1



T1

```
MRS{<c>}{<q>} <Rd>, <spec_reg>

d = UInt(Rd);  read_spsr = (R == '1');
if d == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<Rd> Is the general-purpose destination register, encoded in the "Rd" field.

<spec_reg> Is the special register to be accessed, encoded in “R”:

R	<spec_reg>
0	CPSR|APSR
1	SPSR

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if read_spsr then
        if PSTATE.M IN {M32_User,M32_System} then
            UNPREDICTABLE;
        else
            R[d] = SPSR[];
    else
        // CPSR has same bit assignments as SPSR, but with the IT, J, SS, IL, and T bits masked out.
        bits(32) mask = '11111000 11101111 00000011 11011111';
        psr_val = GetPSRFromPSTATE(AArch32_NonDebugState) AND mask;
        if PSTATE.EL == EL0 then
            // If accessed from User mode return UNKNOWN values for E, A, I, F bits, bits<9:6>,
            // and for the M field, bits<4:0>
            psr_val<22> = bits(1) UNKNOWN;
            psr_val<9:6> = bits(4) UNKNOWN;
            psr_val<4:0> = bits(5) UNKNOWN;
        R[d] = psr_val;
```

CONSTRAINED UNPREDICTABLE behavior

If `PSTATE.M IN {M32_User, M32_System} && read_spsr`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MRS (Banked register)

Move to Register from Banked or Special register moves the value from the Banked general-purpose register or *Saved Program Status Registers (SPSRs)* of the specified mode, or the value of *ELR_hyp*, to a general-purpose register.

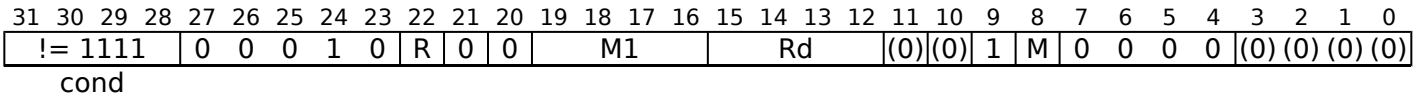
MRS (Banked register) is UNPREDICTABLE if executed in User mode.

When EL3 is using AArch64, if an MRS (Banked register) instruction that is executed in a Secure EL1 mode would access SPSR_mon, SP_mon, or LR_mon, it is trapped to EL3.

The effect of using an MRS (Banked register) instruction with a register argument that is not valid for the current mode is UNPREDICTABLE. For more information see *Usage restrictions on the Banked register transfer instructions*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

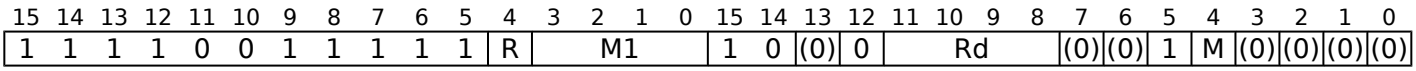
A1



A1

```
MRS{<c>}{<q>} <Rd>, <banked_reg>

d = UInt(Rd);  read_spsr = (R == '1');
if d == 15 then UNPREDICTABLE;
SYSm = M:M1;
```

T1



T1

```
MRS{<c>}{<q>} <Rd>, <banked_reg>

d = UInt(Rd);  read_spsr = (R == '1');
if d == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
SYSm = M:M1;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<banked_reg>	Is the name of the banked register to be transferred to or from, encoded in “R:M:M1”:

R	M	M1	<banked_reg>
0	0	0000	R8_usr
0	0	0001	R9_usr
0	0	0010	R10_usr
0	0	0011	R11_usr
0	0	0100	R12_usr
0	0	0101	SP_usr
0	0	0110	LR_usr
0	0	0111	UNPREDICTABLE
0	0	1000	R8_fiq
0	0	1001	R9_fiq
0	0	1010	R10_fiq
0	0	1011	R11_fiq
0	0	1100	R12_fiq
0	0	1101	SP_fiq
0	0	1110	LR_fiq
0	0	1111	UNPREDICTABLE
0	1	0000	LR_irq
0	1	0001	SP_irq
0	1	0010	LR_svc
0	1	0011	SP_svc
0	1	0100	LR_abt
0	1	0101	SP_abt
0	1	0110	LR_und
0	1	0111	SP_und
0	1	10xx	UNPREDICTABLE
0	1	1100	LR_mon
0	1	1101	SP_mon
0	1	1110	ELR_hyp
0	1	1111	SP_hyp
1	0	0xxx	UNPREDICTABLE
1	0	10xx	UNPREDICTABLE
1	0	110x	UNPREDICTABLE
1	0	1110	SPSR_fiq
1	0	1111	UNPREDICTABLE
1	1	0000	SPSR_irq
1	1	0001	UNPREDICTABLE
1	1	0010	SPSR_svc
1	1	0011	UNPREDICTABLE
1	1	0100	SPSR_abt
1	1	0101	UNPREDICTABLE
1	1	0110	SPSR_und
1	1	0111	UNPREDICTABLE
1	1	10xx	UNPREDICTABLE
1	1	1100	SPSR_mon
1	1	1101	UNPREDICTABLE
1	1	1110	SPSR_hyp
1	1	1111	UNPREDICTABLE

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if PSTATE.EL == EL0 then
       UNPREDICTABLE;
    else
        mode = PSTATE.M;
        if read_spsr then
            SPSRaccessValid(SYSm, mode);           // Check for UNPREDICTABLE cases
            case SYSm of
                when '01110'  R[d] = SPSR_fiq<31:0>;
                when '10000'  R[d] = SPSR_irq<31:0>;
                when '10010'  R[d] = SPSR_svc<31:0>;
                when '10100'  R[d] = SPSR_abt<31:0>;
                when '10110'  R[d] = SPSR_und<31:0>;
                when '11100'
                    if !ELUsingAArch32(EL3) then AArch64.MonitorModeTrap();
                    R[d] = SPSR_mon;
                when '11110'  R[d] = SPSR_hyp<31:0>;
        else
            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_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MSR (Banked register)

Move to Banked or Special register from general-purpose register moves the value of a general-purpose register to the Banked general-purpose register or *Saved Program Status Registers (SPSRs)* of the specified mode, or to *ELR_hyp*.

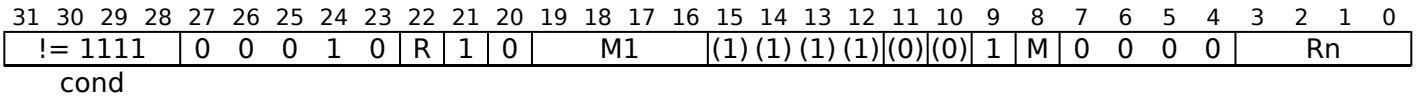
MSR (Banked register) is UNPREDICTABLE if executed in User mode.

When EL3 is using AArch64, if an MSR (Banked register) instruction that is executed in a Secure EL1 mode would access SPSR_mon, SP_mon, or LR_mon, it is trapped to EL3.

The effect of using an MSR (Banked register) instruction with a register argument that is not valid for the current mode is UNPREDICTABLE. For more information see *Usage restrictions on the Banked register transfer instructions*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

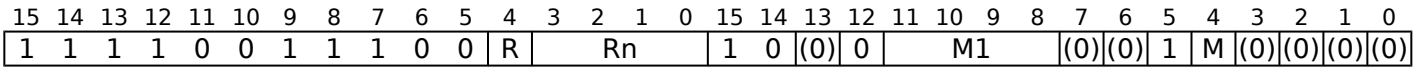


A1

```
MSR{<c>}{<q>} <banked_reg>, <Rn>
```

```
n = UInt(Rn);  write_spsr = (R == '1');
if n == 15 then UNPREDICTABLE;
SYSm = M:M1;
```

T1



T1

```
MSR{<c>}{<q>} <banked_reg>, <Rn>
```

```
n = UInt(Rn);  write_spsr = (R == '1');
if n == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
SYSm = M:M1;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<banked_reg> Is the name of the banked register to be transferred to or from, encoded in “R:M:M1”:

R	M	M1	<banked_reg>
0	0	0000	R8_usr
0	0	0001	R9_usr
0	0	0010	R10_usr
0	0	0011	R11_usr
0	0	0100	R12_usr
0	0	0101	SP_usr
0	0	0110	LR_usr
0	0	0111	UNPREDICTABLE
0	0	1000	R8_fiq
0	0	1001	R9_fiq
0	0	1010	R10_fiq
0	0	1011	R11_fiq
0	0	1100	R12_fiq
0	0	1101	SP_fiq
0	0	1110	LR_fiq
0	0	1111	UNPREDICTABLE
0	1	0000	LR_irq
0	1	0001	SP_irq
0	1	0010	LR_svc
0	1	0011	SP_svc
0	1	0100	LR_abt
0	1	0101	SP_abt
0	1	0110	LR_und
0	1	0111	SP_und
0	1	10xx	UNPREDICTABLE
0	1	1100	LR_mon
0	1	1101	SP_mon
0	1	1110	ELR_hyp
0	1	1111	SP_hyp
1	0	0xxx	UNPREDICTABLE
1	0	10xx	UNPREDICTABLE
1	0	110x	UNPREDICTABLE
1	0	1110	SPSR_fiq
1	0	1111	UNPREDICTABLE
1	1	0000	SPSR_irq
1	1	0001	UNPREDICTABLE
1	1	0010	SPSR_svc
1	1	0011	UNPREDICTABLE
1	1	0100	SPSR_abt
1	1	0101	UNPREDICTABLE
1	1	0110	SPSR_und
1	1	0111	UNPREDICTABLE
1	1	10xx	UNPREDICTABLE
1	1	1100	SPSR_mon
1	1	1101	UNPREDICTABLE
1	1	1110	SPSR_hyp
1	1	1111	UNPREDICTABLE

<Rn> Is the general-purpose source register, encoded in the "Rn" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if PSTATE.EL == EL0 then
        UNPREDICTABLE;
    else
        mode = PSTATE.M;
        if write_spsr then
            SPSRaccessValid(SYSm, mode);             // Check for UNPREDICTABLE cases
            case SYSm of
                when '01110'  SPSR_fiq = ZeroExtend(R[n]);
                when '10000'  SPSR_irq = ZeroExtend(R[n]);
                when '10010'  SPSR_svc = ZeroExtend(R[n]);
                when '10100'  SPSR_abt = ZeroExtend(R[n]);
                when '10110'  SPSR_und = ZeroExtend(R[n]);
                when '11100'
                    if !ELUsingAArch32(EL3) then AArch64.MonitorModeTrap();
                    SPSR_mon = R[n];
                when '11110'  SPSR_hyp = R[n];
        else
            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_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MSR (immediate)

Move immediate value to Special register moves selected bits of an immediate value to the corresponding bits in the *APSR*, *CPSR*, or *SPSR*_<current_mode>.

Because of the Do-Not-Modify nature of its reserved bits, the immediate form of MSR is normally only useful at the Application level for writing to APSR_nzcvq (CPSR_f).

If an MSR (immediate) moves selected bits of an immediate value to the *CPSR*, the PE checks whether the value being written to *PSTATE*.M is legal. See *Illegal changes to PSTATE.M*.

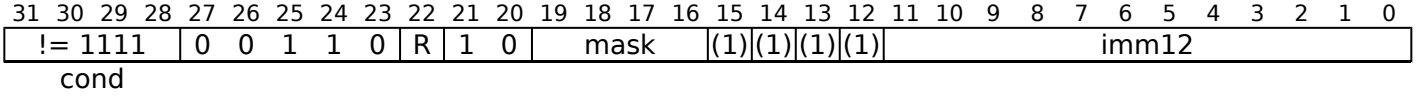
An MSR (immediate) executed in User mode:

- Is CONSTRAINED UNPREDICTABLE if it attempts to update the *SPSR*.
- Otherwise, does not update any *CPSR* field that is accessible only at EL1 or higher,

An MSR (immediate) executed in System mode is CONSTRAINED UNPREDICTABLE if it attempts to update the *SPSR*.

The *CPSR*.E bit is writable from any mode using an MSR instruction. Arm deprecates using this to change its value.

A1



A1 (!(R == 0 && mask == 0000))

```
MSR{<c>}{<q>} <spec_reg>, #<imm>
```

```
if mask == '0000' && R == '0' then SEE "Related encodings";
imm32 = A32ExpandImm(imm12);  write_spsr = (R == '1');
if mask == '0000' then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `mask == '0000' && R == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Related encodings: *Move Special Register and Hints (immediate)*.

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<spec_reg> Is one of:

- APSR_<bits>.
- CPSR_<fields>.
- SPSR_<fields>.

For CPSR and SPSR, <fields> is a sequence of one or more of the following:

c
mask<0> = '1' to enable writing of bits<7:0> of the destination PSR.

x
mask<1> = '1' to enable writing of bits<15:8> of the destination PSR.

s
mask<2> = '1' to enable writing of bits<23:16> of the destination PSR.

f
mask<3> = '1' to enable writing of bits<31:24> of the destination PSR.

For APSR, <bits> is one of nzcvq, g, or nzcvqg. These map to the following CPSR_<fields> values:

- APSR_nzcvq is the same as CPSR_f (mask== '1000').

- APSR_g is the same as CPSR_s (mask == '0100').
- APSR_nzcvqg is the same as CPSR_fs (mask == '1100').

Arm recommends the APSR_<bits> forms when only the N, Z, C, V, Q, and GE[3:0] bits are being written. For more information, see *The Application Program Status Register, APSR*.

<imm> Is an immediate value. See *Modified immediate constants in A32 instructions* for the range of values.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if write_spsr then
        if PSTATE.M IN {M32_User,M32_System} then
            UNPREDICTABLE;
        else
            SPSRWriteByInstr(imm32, mask);
    else
        // Attempts to change to an illegal mode will invoke the Illegal Execution state mechanism
        CPSRWriteByInstr(imm32, mask);
```

CONSTRAINED UNPREDICTABLE behavior

If `PSTATE.M IN {M32_User,M32_System} && write_spsr`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MSR (register)

Move general-purpose register to Special register moves selected bits of a general-purpose register to the *APSR*, *CPSR* or *SPSR*_<current_mode>.

Because of the Do-Not-Modify nature of its reserved bits, a read-modify-write sequence is normally required when the MSR instruction is being used at Application level and its destination is not APSR_nzcvq (CPSR_f).

If an MSR (register) moves selected bits of an immediate value to the *CPSR*, the PE checks whether the value being written to *PSTATE*.M is legal. See *Illegal changes to PSTATE.M*.

An MSR (register) executed in User mode:

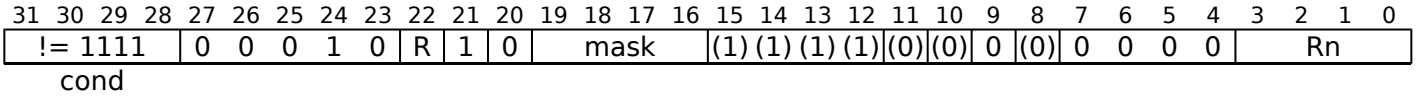
- Is UNPREDICTABLE if it attempts to update the *SPSR*.
- Otherwise, does not update any *CPSR* field that is accessible only at EL1 or higher.

An MSR (register) executed in System mode is UNPREDICTABLE if it attempts to update the *SPSR*.

The *CPSR*.E bit is writable from any mode using an MSR instruction. Arm deprecates using this to change its value.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



A1

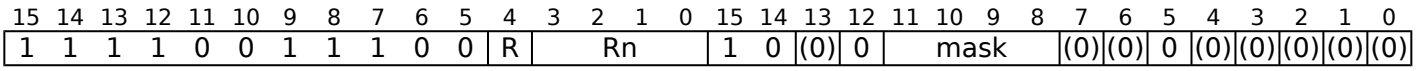
```
MSR{<c>}{<q>} <spec_reg>, <Rn>

n = UInt(Rn);  write_spsr = (R == '1');
if mask == '0000' then UNPREDICTABLE;
if n == 15 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `mask == '0000'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

T1



T1

```
MSR{<c>}{<q>} <spec_reg>, <Rn>

n = UInt(Rn);  write_spsr = (R == '1');
if mask == '0000' then UNPREDICTABLE;
if n == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

CONSTRAINED UNPREDICTABLE behavior

If `mask == '0000'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<spec_reg> Is one of:

- APSR_<bits>.
- CPSR_<fields>.
- SPSR_<fields>.

For CPSR and SPSR, <fields> is a sequence of one or more of the following:

c
mask<0> = '1' to enable writing of bits<7:0> of the destination PSR.

x
mask<1> = '1' to enable writing of bits<15:8> of the destination PSR.

s
mask<2> = '1' to enable writing of bits<23:16> of the destination PSR.

f
mask<3> = '1' to enable writing of bits<31:24> of the destination PSR.

For APSR, <bits> is one of nzcvq, g, or nzcvqg. These map to the following CPSR_<fields> values:

- APSR_nzcvq is the same as CPSR_f (mask== '1000').
- APSR_g is the same as CPSR_s (mask == '0100').
- APSR_nzcvqg is the same as CPSR_fs (mask == '1100').

Arm recommends the APSR_<bits> forms when only the N, Z, C, V, Q, and GE[3:0] bits are being written. For more information, see *The Application Program Status Register, APSR*.

<Rn> Is the general-purpose source register, encoded in the "Rn" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if write_spsr then
        if PSTATE.M IN {M32_User,M32_System} then
            UNPREDICTABLE;
        else
            SPSRWriteByInstr(R[n], mask);
    else
        // Attempts to change to an illegal mode will invoke the Illegal Execution state mechanism
        CPSRWriteByInstr(R[n], mask);
```

CONSTRAINED UNPREDICTABLE behavior

If `write_spsr && PSTATE.M IN {M32_User,M32_System}`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

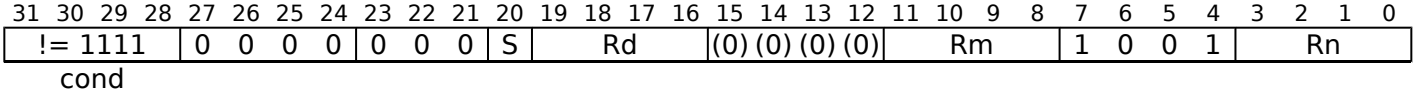
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MUL, MULS

Multiply multiplies two register values. The least significant 32 bits of the result are written to the destination register. These 32 bits do not depend on whether the source register values are considered to be signed values or unsigned values.
Optionally, it can update the condition flags based on the result. In the T32 instruction set, this option is limited to only a few forms of the instruction. Use of this option adversely affects performance on many implementations.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2) .

A1



Flag setting (S == 1)

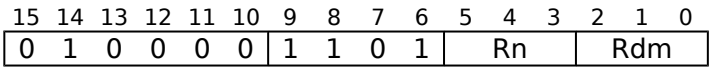
```
MULS{<c>}{<q>} <Rd>, <Rn>{, <Rm>}
```

Not flag setting (S == 0)

```
MUL{<c>}{<q>} <Rd>, <Rn>{, <Rm>}
```

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1

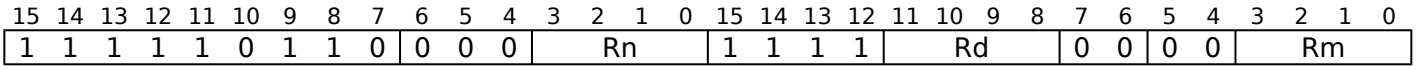


T1

```
MUL<c>{<q>} <Rdm>, <Rn>{, <Rdm>} // (Inside IT block)

MULS{<q>} <Rdm>, <Rn>{, <Rdm>} // (Outside IT block)
```

```
d = UInt(Rdm);  n = UInt(Rn);  m = UInt(Rdm);  setflags = !InITBlock();
```

T2



T2

```
MUL<c>.W <Rd>, <Rn>{, <Rm>} // (Inside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1)

MUL{<c>}{<q>} <Rd>, <Rn>{, <Rm>}
```

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = FALSE;
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<Rdm> Is the second general-purpose source register holding the multiplier and the destination register, encoded in the "Rdm" field.

<Rd> Is the general-purpose destination register, encoded in the "Rd" field.

<Rn> Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

<Rm> Is the second general-purpose source register holding the multiplier, encoded in the "Rm" field. If omitted, <Rd> is used.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand1 = SInt(R[n]);  // operand1 = UInt(R[n]) produces the same final results
    operand2 = SInt(R[m]);  // operand2 = UInt(R[m]) produces the same final results
    result = operand1 * operand2;
    R[d] = result<31:0>;
    if setflags then
        PSTATE.N = result<31>;
        PSTATE.Z = IsZeroBit(result<31:0>);
        // PSTATE.C, PSTATE.V unchanged
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MVN, MVNS (immediate)

Bitwise NOT (immediate) writes the bitwise inverse of an immediate value to the destination register.

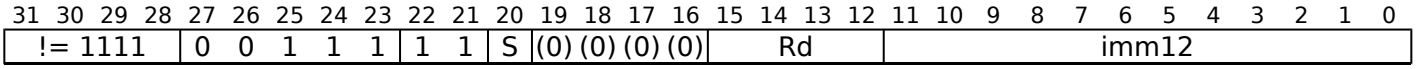
If the destination register is not the PC, the MVNS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. ARM deprecates any use of these encodings. However, when the destination register is the PC:

- The MVN variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*.
- The MVNS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores *PSTATE* from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state*.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



MVN (S == 0)

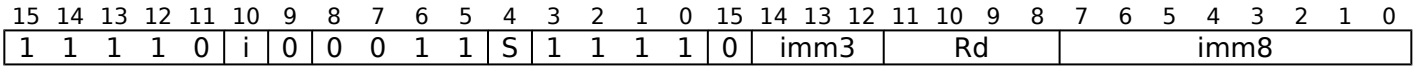
```
MVN{<c>}{<q>} <Rd>, #<const>
```

MVNS (S == 1)

```
MVNS{<c>}{<q>} <Rd>, #<const>
```

```
d = UInt(Rd);  setflags = (S == '1');
(imm32, carry) = A32ExpandImm_C(imm12, PSTATE.C);
```

T1



MVN (S == 0)

```
MVN{<c>}{<q>} <Rd>, #<const>
```

MVNS (S == 1)

```
MVNS{<c>}{<q>} <Rd>, #<const>
```

```
d = UInt(Rd);  setflags = (S == '1');
(imm32, carry) = T32ExpandImm_C(i:imm3:imm8, PSTATE.C);
if d == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<Rd> For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the MVN variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*.
- For the MVNS variant, the instruction performs an exception return, that restores *PSTATE* from SPSR_<current_mode>.

For encoding T1: is the general-purpose destination register, encoded in the "Rd" field.

<const> For encoding A1: an immediate value. See *Modified immediate constants in A32 instructions* for the range of values.

For encoding T1: an immediate value. See *Modified immediate constants in T32 instructions* for the range of values.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = NOT(imm32);
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.N = result<31>;
            PSTATE.Z = IsZeroBit(result);
            PSTATE.C = carry;
            // PSTATE.V unchanged
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MVN, MVNS (register)

Bitwise NOT (register) writes the bitwise inverse of a register value to the destination register.

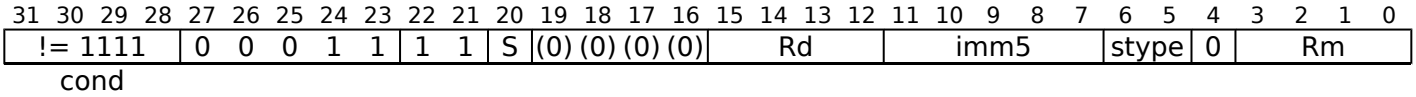
If the destination register is not the PC, the MVNS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. ARM deprecates any use of these encodings. However, when the destination register is the PC:

- The MVN variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*.
- The MVNS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores *PSTATE* from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state*.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2) .

A1



MVN, rotate right with extend (S == 0 && imm5 == 00000 && stype == 11)

```
MVN{<c>}{<q>} <Rd>, <Rm>, RRX
```

MVN, shift or rotate by value (S == 0 && !(imm5 == 00000 && stype == 11))

```
MVN{<c>}{<q>} <Rd>, <Rm> {, <shift> #<amount>}
```

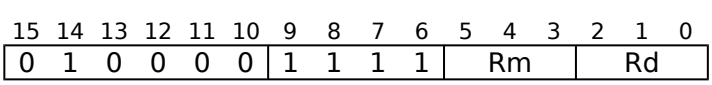
MVNS, rotate right with extend (S == 1 && imm5 == 00000 && stype == 11)

```
MVNS{<c>}{<q>} <Rd>, <Rm>, RRX
```

MVNS, shift or rotate by value (S == 1 && !(imm5 == 00000 && stype == 11))

```
MVNS{<c>}{<q>} <Rd>, <Rm> {, <shift> #<amount>}

d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm5);
```

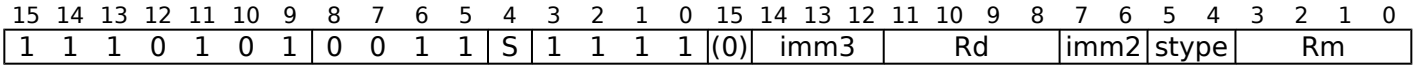
T1



T1

```
MVN<c>{<q>} <Rd>, <Rm> // (Inside IT block)

MVNS{<q>} <Rd>, <Rm> // (Outside IT block)

d = UInt(Rd);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);
```

T2



MVN, rotate right with extend (S == 0 && imm3 == 000 && imm2 == 00 && stype == 11)

```
MVN{<c>}{<q>} <Rd>, <Rm>, RRX
```

MVN, shift or rotate by value (S == 0 && !(imm3 == 000 && imm2 == 00 && stype == 11))

```
MVN<c>.W <Rd>, <Rm> // (Inside IT block, and <Rd>, <Rm> can be represented in T1)

MVN{<c>}{<q>} <Rd>, <Rm> {, <shift> #<amount>}
```

MVNS, rotate right with extend (S == 1 && imm3 == 000 && imm2 == 00 && stype == 11)

```
MVNS{<c>}{<q>} <Rd>, <Rm>, RRX
```

MVNS, shift or rotate by value (S == 1 && !(imm3 == 000 && imm2 == 00 && stype == 11))

```
MVNS.W <Rd>, <Rm> // (Outside IT block, and <Rd>, <Rm> can be represented in T1)

MVNS{<c>}{<q>} <Rd>, <Rm> {, <shift> #<amount>}
```

```
d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm3:imm2);
if d == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<Rd> For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the MVN variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*.
- For the MVNS variant, the instruction performs an exception return, that restores *PSTATE* from SPSR_<current_mode>.

For encoding T1 and T2: is the general-purpose destination register, encoded in the "Rd" field.

<Rm> For encoding A1: is the general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.

For encoding T1 and T2: is the general-purpose source register, encoded in the "Rm" field.

<shift> Is the type of shift to be applied to the source register, encoded in “stype”:

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<amount> For encoding A1: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the "imm5" field as <amount> modulo 32.

For encoding T2: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the "imm3:imm2" field as <amount> modulo 32.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
    result = NOT(shifted);
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.N = result<31>;
            PSTATE.Z = IsZeroBit(result);
            PSTATE.C = carry;
            // PSTATE.V unchanged
```

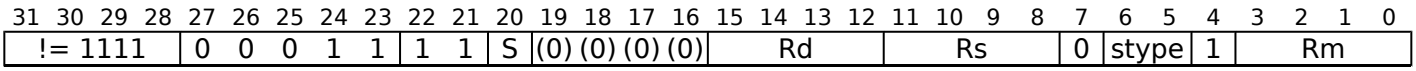
Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MVN, MVNS (register-shifted register)

Bitwise NOT (register-shifted register) writes the bitwise inverse of a register-shifted register value to the destination register. It can optionally update the condition flags based on the result.

A1



Flag setting (S == 1)

```
MVNS{<c>}{<q>} <Rd>, <Rm>, <shift> <Rs>
```

Not flag setting (S == 0)

```
MVN{<c>}{<q>} <Rd>, <Rm>, <shift> <Rs>
```

```
d = UInt(Rd);  m = UInt(Rm);  s = UInt(Rs);
setflags = (S == '1');  shift_t = DecodeRegShift(stype);
if d == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<Rd> Is the general-purpose destination register, encoded in the "Rd" field.

<Rm> Is the general-purpose source register, encoded in the "Rm" field.

<shift> Is the type of shift to be applied to the second source register, encoded in “stype”:

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<Rs> Is the general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[s]<7:0>);
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
    result = NOT(shifted);
    R[d] = result;
    if setflags then
        PSTATE.N = result<31>;
        PSTATE.Z = IsZeroBit(result);
        PSTATE.C = carry;
        // PSTATE.V unchanged
```

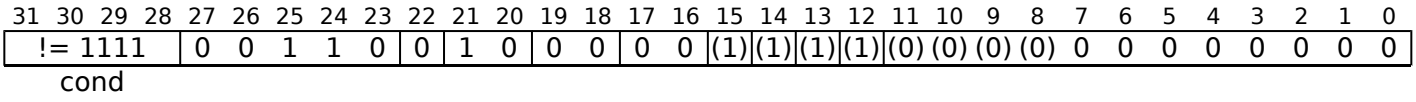
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

NOP

No Operation does nothing. This instruction can be used for instruction alignment purposes.
The timing effects of including a NOP instruction in a program are not guaranteed. It can increase execution time, leave it unchanged, or even reduce it. Therefore, NOP instructions are not suitable for timing loops.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2) .

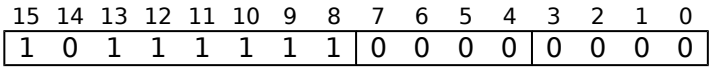
A1



A1

```
NOP{<c>}{<q>}

// No additional decoding required
```

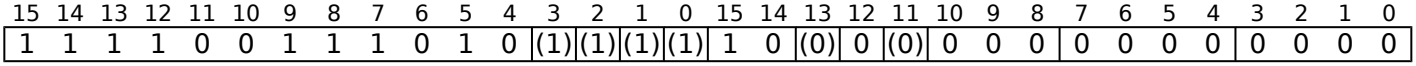
T1



T1

```
NOP{<c>}{<q>}

// No additional decoding required
```

T2



T2

```
NOP{<c>}.W

// No additional decoding required
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    // Do nothing
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

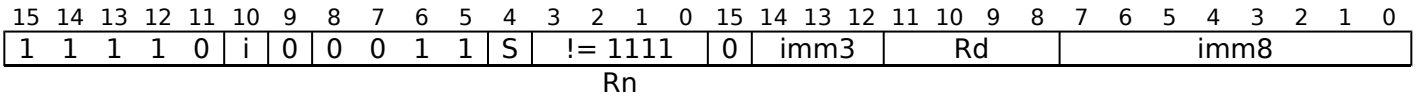
Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ORN, ORNS (immediate)

Bitwise OR NOT (immediate) performs a bitwise (inclusive) OR of a register value and the complement of an immediate value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

T1



Flag setting (S == 1)

```
ORNS{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

Not flag setting (S == 0)

```
ORN{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

```
if Rn == '1111' then SEE "MVN (immediate)";
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');
(imm32, carry) = T32ExpandImm_C(i:imm3:imm8, PSTATE.C);
if d == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.
<Rn>	Is the general-purpose source register, encoded in the "Rn" field.
<const>	An immediate value. See *Modified immediate constants in T32 instructions* for the range of values.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = R[n] OR NOT(imm32);
    R[d] = result;
    if setflags then
        PSTATE.N = result<31>;
        PSTATE.Z = IsZeroBit(result);
        PSTATE.C = carry;
        // PSTATE.V unchanged
```

Operational information

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

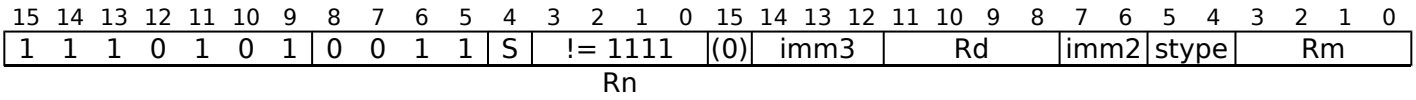
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ORN, ORNS (register)

Bitwise OR NOT (register) performs a bitwise (inclusive) OR of a register value and the complement of an optionally-shifted register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

T1



ORN, rotate right with extend (S == 0 && imm3 == 000 && imm2 == 00 && stype == 11)

```
ORN{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

ORN, shift or rotate by value (S == 0 && !(imm3 == 000 && imm2 == 00 && stype == 11))

```
ORN{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

ORNS, rotate right with extend (S == 1 && imm3 == 000 && imm2 == 00 && stype == 11)

```
ORNS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

ORNS, shift or rotate by value (S == 1 && !(imm3 == 000 && imm2 == 00 && stype == 11))

```
ORNS{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

```
if Rn == '1111' then SEE "MVN (register)";
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm3:imm2);
if d == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<Rd> Is the general-purpose destination register, encoded in the "Rd" field.

<Rn> Is the first general-purpose source register, encoded in the "Rn" field.

<Rm> Is the second general-purpose source register, encoded in the "Rm" field.

<shift> Is the type of shift to be applied to the second source register, encoded in “stype”:

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<amount> Is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the "imm3:imm2" field as <amount> modulo 32.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
    result = R[n] OR NOT(shifted);
    R[d] = result;
    if setflags then
        PSTATE.N = result<31>;
        PSTATE.Z = IsZeroBit(result);
        PSTATE.C = carry;
        // PSTATE.V unchanged
```

Operational information

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ORR, ORRS (immediate)

Bitwise OR (immediate) performs a bitwise (inclusive) OR of a register value and an immediate value, and writes the result to the destination register.

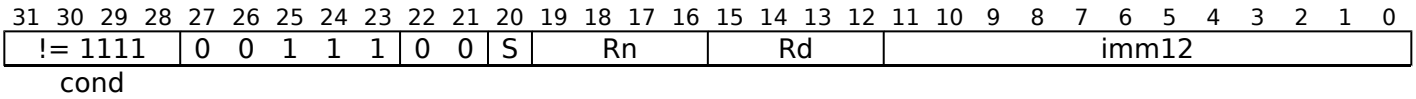
If the destination register is not the PC, the ORRS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. ARM deprecates any use of these encodings. However, when the destination register is the PC:

- The ORR variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*.
- The ORRS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores *PSTATE* from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state*.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



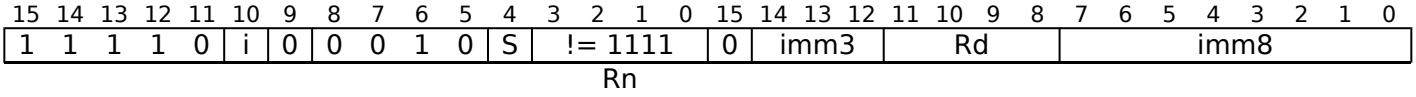
ORR (S == 0)

```
ORR{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

ORRS (S == 1)

```
ORRS{<c>}{<q>} {<Rd>,} <Rn>, #<const>

d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');
(imm32, carry) = A32ExpandImm_C(imm12, PSTATE.C);
```

T1



ORR (S == 0)

```
ORR{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

ORRS (S == 1)

```
ORRS{<c>}{<q>} {<Rd>,} <Rn>, #<const>

if Rn == '1111' then SEE "MOV (immediate)";
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');
(imm32, carry) = T32ExpandImm_C(i:imm3:imm8, PSTATE.C);
if d == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<Rd> For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the ORR variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*.
- For the ORRS variant, the instruction performs an exception return, that restores *PSTATE* from SPSR_<current_mode>.

For encoding T1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.

<Rn> For encoding A1: is the general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.

For encoding T1: is the general-purpose source register, encoded in the "Rn" field.

<const> For encoding A1: an immediate value. See *Modified immediate constants in A32 instructions* for the range of values.

For encoding T1: an immediate value. See *Modified immediate constants in T32 instructions* for the range of values.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = R[n] OR imm32;
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.N = result<31>;
            PSTATE.Z = IsZeroBit(result);
            PSTATE.C = carry;
            // PSTATE.V unchanged
```

Operational information

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ORR, ORRS (register)

Bitwise OR (register) performs a bitwise (inclusive) OR of a register value and an optionally-shifted register value, and writes the result to the destination register.

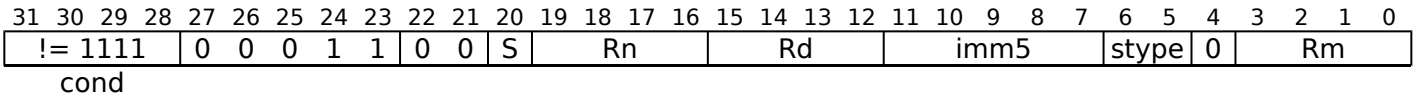
If the destination register is not the PC, the ORRS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. ARM deprecates any use of these encodings. However, when the destination register is the PC:

- The ORR variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*.
- The ORRS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores *PSTATE* from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state*.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2) .

A1



ORR, rotate right with extend (S == 0 && imm5 == 00000 && stype == 11)

```
ORR{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

ORR, shift or rotate by value (S == 0 && !(imm5 == 00000 && stype == 11))

```
ORR{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

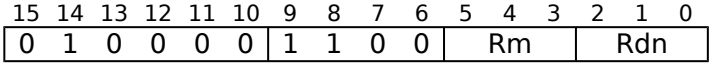
ORRS, rotate right with extend (S == 1 && imm5 == 00000 && stype == 11)

```
ORRS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

ORRS, shift or rotate by value (S == 1 && !(imm5 == 00000 && stype == 11))

```
ORRS{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm5);
```

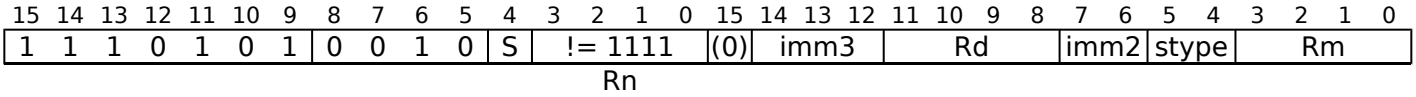
T1



T1

```
ORR<c>{<q>} {<Rdn>,} <Rdn>, <Rm> // (Inside IT block)

ORRS{<q>} {<Rdn>,} <Rdn>, <Rm> // (Outside IT block)

d = UInt(Rdn);  n = UInt(Rdn);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);
```

T2



ORR, rotate right with extend (S == 0 && imm3 == 000 && imm2 == 00 && stype == 11)

```
ORR{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

ORR, shift or rotate by value (S == 0 && !(imm3 == 000 && imm2 == 00 && stype == 11))

```
ORR<c>.W {<Rd>,} <Rn>, <Rm> // (Inside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1)

ORR{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

ORRS, rotate right with extend (S == 1 && imm3 == 000 && imm2 == 00 && stype == 11)

```
ORRS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

ORRS, shift or rotate by value (S == 1 && !(imm3 == 000 && imm2 == 00 && stype == 11))

```
ORRS.W {<Rd>,} <Rn>, <Rm> // (Outside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1)

ORRS{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

```
if Rn == '1111' then SEE "Related encodings";
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm3:imm2);
if d == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Related encodings: *Data-processing (shifted register)*

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<Rdn> Is the first general-purpose source register and the destination register, encoded in the "Rdn" field.

<Rd> For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the ORR variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*.
- For the ORRS variant, the instruction performs an exception return, that restores *PSTATE* from SPSR_<current_mode>.

For encoding T2: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.

<Rn> For encoding A1: is the first general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.

For encoding T2: is the first general-purpose source register, encoded in the "Rn" field.

<Rm> For encoding A1: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.

For encoding T1 and T2: is the second general-purpose source register, encoded in the "Rm" field.

<shift> Is the type of shift to be applied to the second source register, encoded in “stype”:

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<amount> For encoding A1: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the "imm5" field as <amount> modulo 32.

For encoding T2: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the "imm3:imm2" field as <amount> modulo 32.

In T32 assembly:

- Outside an IT block, if ORRS <Rd>, <Rn>, <Rd> is written with <Rd> and <Rn> both in the range R0-R7, it is assembled using encoding T1 as though ORRS <Rd>, <Rn> had been written.
- Inside an IT block, if ORR<c> <Rd>, <Rn>, <Rd> is written with <Rd> and <Rn> both in the range R0-R7, it is assembled using encoding T1 as though ORR<c> <Rd>, <Rn> had been written.

To prevent either of these happening, use the .W qualifier.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
    result = R[n] OR shifted;
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.N = result<31>;
            PSTATE.Z = IsZeroBit(result);
            PSTATE.C = carry;
            // PSTATE.V unchanged
```

Operational information

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

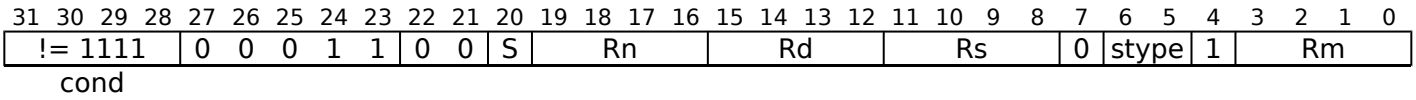
Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ORR, ORRS (register-shifted register)

Bitwise OR (register-shifted register) performs a bitwise (inclusive) OR of a register value and a register-shifted register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

A1



Flag setting (S == 1)

```
ORRS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, <shift> <Rs>
```

Not flag setting (S == 0)

```
ORR{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, <shift> <Rs>
```

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  s = UInt(Rs);
setflags = (S == '1');  shift_t = DecodeRegShift(stype);
if d == 15 || n == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<Rd> Is the general-purpose destination register, encoded in the "Rd" field.

<Rn> Is the first general-purpose source register, encoded in the "Rn" field.

<Rm> Is the second general-purpose source register, encoded in the "Rm" field.

<shift> Is the type of shift to be applied to the second source register, encoded in “stype”:

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<Rs> Is the general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[s]<7:0>);
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
    result = R[n] OR shifted;
    R[d] = result;
    if setflags then
        PSTATE.N = result<31>;
        PSTATE.Z = IsZeroBit(result);
        PSTATE.C = carry;
        // PSTATE.V unchanged
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

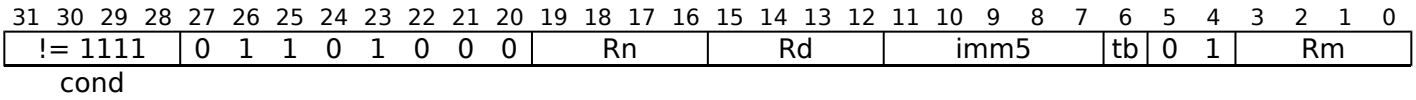
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

PKHBT, PKHTB

Pack Halfword combines one halfword of its first operand with the other halfword of its shifted second operand.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



PKHBT (tb == 0)

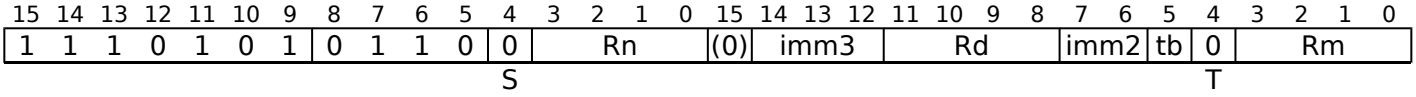
```
PKHBT{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, LSL #<imm>}
```

PKHTB (tb == 1)

```
PKHTB{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, ASR #<imm>}
```

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  tbform = (tb == '1');
(shift_t, shift_n) = DecodeImmShift(tb:'0', imm5);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1



PKHBT (tb == 0)

```
PKHBT{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, LSL #<imm>} // (tbform == FALSE)
```

PKHTB (tb == 1)

```
PKHTB{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, ASR #<imm>} // (tbform == TRUE)
```

```
if S == '1' || T == '1' then UNDEFINED;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  tbform = (tb == '1');
(shift_t, shift_n) = DecodeImmShift(tb:'0', imm3:imm2);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.
<imm>	For encoding A1: the shift to apply to the value read from <Rm>, encoded in the "imm5" field. For PKHBT, it is one of: **omitted** No shift, encoded as 0b00000.

1-31
Left shift by specified number of bits, encoded as a binary number.

For PKHTB, it is one of:

omitted
Instruction is a pseudo-instruction and is assembled as though PKHBT{<c>}{<q>} <Rd>, <Rm>, <Rn> had been written.

1-32
Arithmetic right shift by specified number of bits. A shift by 32 bits is encoded as 0b00000. Other shift amounts are encoded as binary numbers.

An assembler can permit <imm> = 0 to mean the same thing as omitting the shift, but this is not standard UAL and must not be used for disassembly.

For encoding T1: the shift to apply to the value read from <Rm>, encoded in the "imm3:imm2" field.

For PKHBT, it is one of:

omitted
No shift, encoded as 0b00000.

1-31
Left shift by specified number of bits, encoded as a binary number.

For PKHTB, it is one of:

omitted
Instruction is a pseudo-instruction and is assembled as though PKHBT{<c>}{<q>} <Rd>, <Rm>, <Rn> had been written.

1-32
Arithmetic right shift by specified number of bits. A shift by 32 bits is encoded as 0b00000. Other shift amounts are encoded as binary numbers.

An assembler can permit <imm> = 0 to mean the same thing as omitting the shift, but this is not standard UAL and must not be used for disassembly.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand2 = Shift(R[m], shift_t, shift_n, PSTATE.C);  // PSTATE.C ignored
    R[d]<15:0>  = if tbform then operand2<15:0> else R[n]<15:0>;
    R[d]<31:16> = if tbform then R[n]<31:16>    else operand2<31:16>;
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

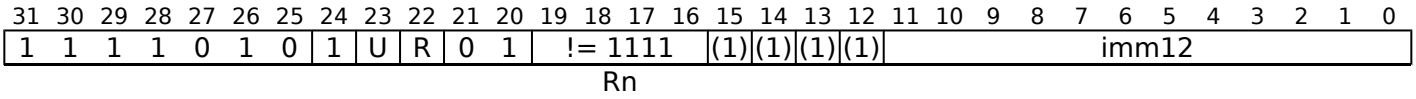
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

PLD, PLDW (immediate)

Preload Data (immediate) signals the memory system that data memory accesses from a specified address are likely in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as preloading the cache line containing the specified address into the data cache. The PLD instruction signals that the likely memory access is a read, and the PLDW instruction signals that it is a write. The effect of a PLD or PLDW instruction is IMPLEMENTATION DEFINED. For more information, see *Preloading caches*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2) .

A1



Preload read (R == 1)

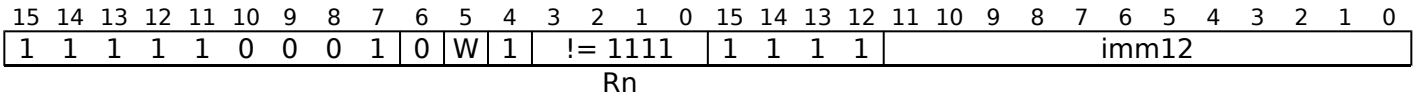
```
PLD{<c>}{<q>} [<Rn> {, #{+/-}<imm>}]
```

Preload write (R == 0)

```
PLDW{<c>}{<q>} [<Rn> {, #{+/-}<imm>}]
```

```
if Rn == '1111' then SEE "PLD (literal)";
n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);  add = (U == '1');  is_pldw = (R == '0');
```

T1



Preload read (W == 0)

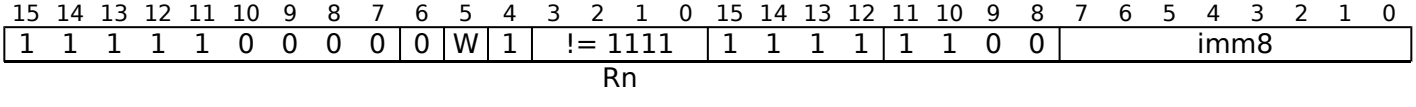
```
PLD{<c>}{<q>} [<Rn> {, #{+}<imm>}]
```

Preload write (W == 1)

```
PLDW{<c>}{<q>} [<Rn> {, #{+}<imm>}]
```

```
if Rn == '1111' then SEE "PLD (literal)";
n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);  add = TRUE;  is_pldw = (W == '1');
```

T2



Preload read (W == 0)

```
PLD{<c>}{<q>} [<Rn> {, #-<imm>}]
```

Preload write (W == 1)

```
PLDW{<c>}{<q>} [<Rn> {, #-<imm>}]
```

```
if Rn == '1111' then SEE "PLD (literal)";
n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);  add = FALSE;  is_pldw = (W == '1');
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c> For encoding A1: see *Standard assembler syntax fields*. Must be AL or omitted.

For encoding T1 and T2: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<Rn> Is the general-purpose base register, encoded in the "Rn" field. If the PC is used, see *PLD (literal)*.

+/- Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

\+ Specifies the offset is added to the base register.

<imm> For encoding A1: is the optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

For encoding T1: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

For encoding T2: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm8" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = if add then (R[n] + imm32) else (R[n] - imm32);
    if is_pldw then
        Hint_PreloadDataForWrite(address);
    else
        Hint_PreloadData(address);
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

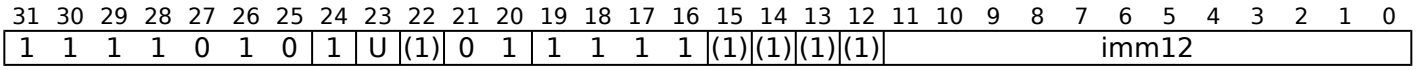
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

PLD (literal)

Preload Data (literal) signals the memory system that data memory accesses from a specified address are likely in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as preloading the cache line containing the specified address into the data cache.
The effect of a PLD instruction is IMPLEMENTATION DEFINED. For more information, see *Preloading caches*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

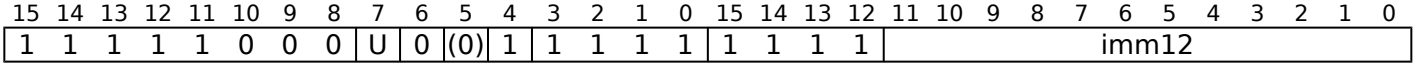
A1



A1

```
PLD{<c>}{<q>} <label> // (Normal form)

PLD{<c>}{<q>} [PC, #{+/-}<imm>] // (Alternative form)

imm32 = ZeroExtend(imm12, 32);  add = (U == '1');
```

T1



T1

```
PLD{<c>}{<q>} <label> // (Preferred syntax)

PLD{<c>}{<q>} [PC, #{+/-}<imm>] // (Alternative syntax)

imm32 = ZeroExtend(imm12, 32);  add = (U == '1');
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c> For encoding A1: see *Standard assembler syntax fields*. Must be AL or omitted.

For encoding T1: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<label> The label of the literal data item that is likely to be accessed in the near future. The assembler calculates the required value of the offset from the Align(PC, 4) value of the instruction to this label. The offset must be in the range –4095 to 4095.
If the offset is zero or positive, imm32 is equal to the offset and add == TRUE.
If the offset is negative, imm32 is equal to minus the offset and add == FALSE.

+/- Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<imm> For encoding A1: is the 12-bit unsigned immediate byte offset, in the range 0 to 4095, encoded in the "imm12" field.

For encoding T1: is a 12-bit unsigned immediate byte offset, in the range 0 to 4095, encoded in the "imm12" field.

The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see *Use of labels in UAL instruction syntax*.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = if add then (Align(PC,4) + imm32) else (Align(PC,4) - imm32);
    Hint_PreloadData(address);
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

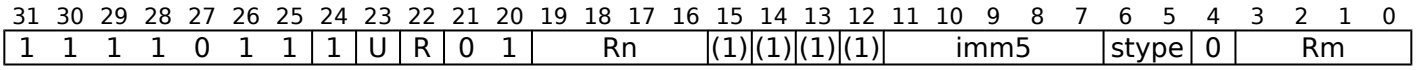
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

PLD, PLDW (register)

Preload Data (register) signals the memory system that data memory accesses from a specified address are likely in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as preloading the cache line containing the specified address into the data cache. The PLD instruction signals that the likely memory access is a read, and the PLDW instruction signals that it is a write. The effect of a PLD or PLDW instruction is IMPLEMENTATION DEFINED. For more information, see *Preloading caches*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



Preload read, optional shift or rotate (R == 1 && !(imm5 == 00000 && stype == 11))

```
PLD{<c>}{<q>} [<Rn>, {+/-}<Rm> {, <shift> #<amount>}]
```

Preload read, rotate right with extend (R == 1 && imm5 == 00000 && stype == 11)

```
PLD{<c>}{<q>} [<Rn>, {+/-}<Rm> , RRX]
```

Preload write, optional shift or rotate (R == 0 && !(imm5 == 00000 && stype == 11))

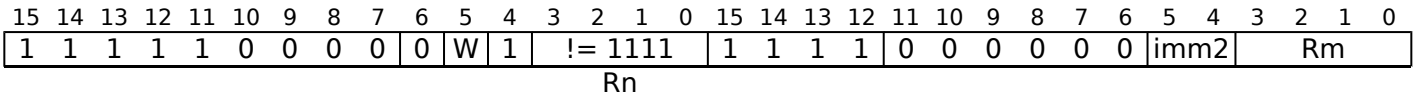
```
PLDW{<c>}{<q>} [<Rn>, {+/-}<Rm> {, <shift> #<amount>}]
```

Preload write, rotate right with extend (R == 0 && imm5 == 00000 && stype == 11)

```
PLDW{<c>}{<q>} [<Rn>, {+/-}<Rm> , RRX]
```

```
n = UInt(Rn);  m = UInt(Rm);  add = (U == '1');  is_pldw = (R == '0');
(shift_t, shift_n) = DecodeImmShift(stype, imm5);
if m == 15 || (n == 15 && is_pldw) then UNPREDICTABLE;
```

T1



Preload read (W == 0)

```
PLD{<c>}{<q>} [<Rn>, {+}<Rm> {, LSL #<amount>}]
```

Preload write (W == 1)

```
PLDW{<c>}{<q>} [<Rn>, {+}<Rm> {, LSL #<amount>}]
```

```
if Rn == '1111' then SEE "PLD (literal)";
n = UInt(Rn);  m = UInt(Rm);  add = TRUE;  is_pldw = (W == '1');
(shift_t, shift_n) = (SRType_LSL, UInt(imm2));
if m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c> For encoding A1: see *Standard assembler syntax fields*. <c> must be AL or omitted.

For encoding T1: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<Rn> For encoding A1: is the general-purpose base register, encoded in the "Rn" field. The PC can be used.

For encoding T1: is the general-purpose base register, encoded in the "Rn" field.

+/- Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

\+ Specifies the index register is added to the base register.

<Rm> Is the general-purpose index register, encoded in the "Rm" field.

<shift> Is the type of shift to be applied to the index register, encoded in “stype”:

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<amount> For encoding A1: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm5" field as <amount> modulo 32.

For encoding T1: is the shift amount, in the range 0 to 3, defaulting to 0 and encoded in the "imm2" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset = Shift(R[m], shift_t, shift_n, PSTATE.C);
    address = if add then (R[n] + offset) else (R[n] - offset);
    if is_pldw then
        Hint_PreloadDataForWrite(address);
    else
        Hint_PreloadData(address);
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

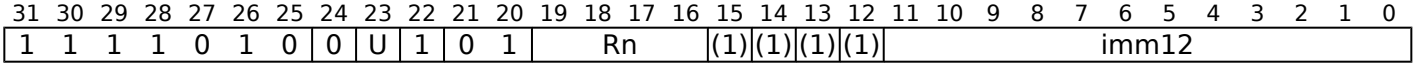
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

PLI (immediate, literal)

Preload Instruction signals the memory system that instruction memory accesses from a specified address are likely in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as pre-loading the cache line containing the specified address into the instruction cache.
The effect of a PLI instruction is IMPLEMENTATION DEFINED. For more information, see *Preloading caches*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 , T2 and T3) .

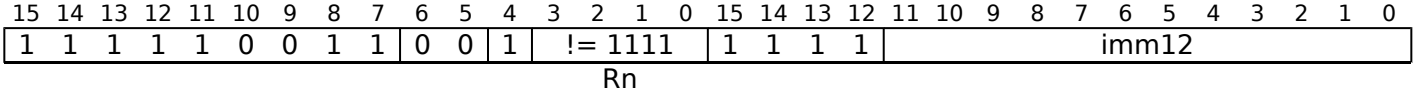
A1



A1

```
PLI{<c>}{<q>} [<Rn> {, #{+/-}<imm>}]

PLI{<c>}{<q>} <label> // (Normal form)

PLI{<c>}{<q>} [PC, #{+/-}<imm>] // (Alternative form)

n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);  add = (U == '1');
```

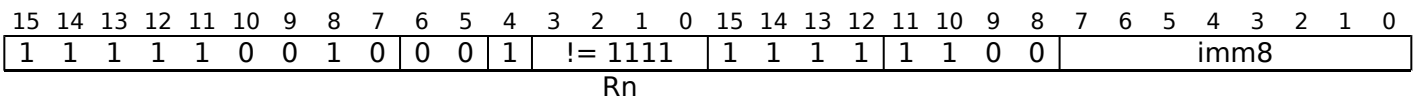
T1



T1

```
PLI{<c>}{<q>} [<Rn> {, #{+}<imm>}]

if Rn == '1111' then SEE "encoding T3";
n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);  add = TRUE;
```

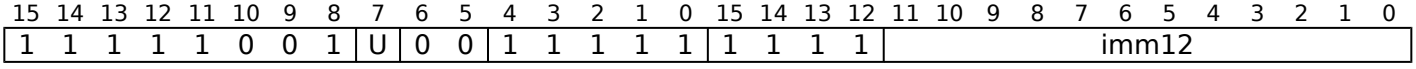
T2



T2

```
PLI{<c>}{<q>} [<Rn> {, #-<imm>}]

if Rn == '1111' then SEE "encoding T3";
n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);  add = FALSE;
```

T3



T3

```
PLI{<c>}{<q>} <label> // (Preferred syntax)

PLI{<c>}{<q>} [PC, #{+/-}<imm>] // (Alternative syntax)

n = 15;  imm32 = ZeroExtend(imm12, 32);  add = (U == '1');
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c> For encoding A1: see *Standard assembler syntax fields*. Must be AL or omitted.

For encoding T1, T2 and T3: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<label> The label of the instruction that is likely to be accessed in the near future. The assembler calculates the required value of the offset from the Align(PC, 4) value of the instruction to this label. The offset must be in the range –4095 to 4095.

If the offset is zero or positive, imm32 is equal to the offset and add == TRUE.

If the offset is negative, imm32 is equal to minus the offset and add == FALSE.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

+/- Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

\+ Specifies the offset is added to the base register.

<imm> For encoding A1: is the optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

For encoding T1: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

For encoding T2: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm8" field.

For encoding T3: is a 12-bit unsigned immediate byte offset, in the range 0 to 4095, encoded in the "imm12" field.

For the literal forms of the instruction, encoding T3 is used, or Rn is encoded as 0b1111 in encoding A1, to indicate that the PC is the base register.

The alternative literal syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see *Use of labels in UAL instruction syntax*.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    base = if n == 15 then Align(PC,4) else R[n];
    address = if add then (base + imm32) else (base - imm32);
    Hint_PreloadInstr(address);
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

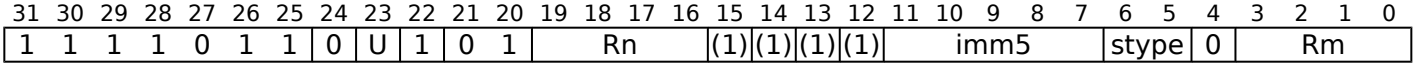
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

PLI (register)

Preload Instruction signals the memory system that instruction memory accesses from a specified address are likely in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as pre-loading the cache line containing the specified address into the instruction cache.
The effect of a PLI instruction is IMPLEMENTATION DEFINED. For more information, see *Preloading caches*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



Rotate right with extend (imm5 == 00000 && stype == 11)

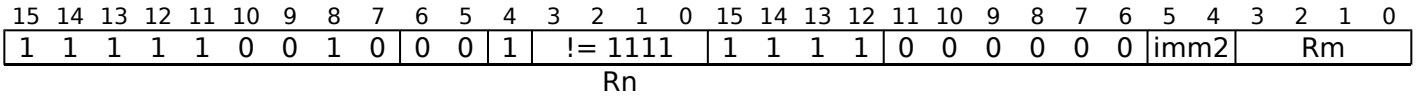
```
PLI{<c>}{<q>} [<Rn>, {+/-}<Rm> , RRX]
```

Shift or rotate by value (!(imm5 == 00000 && stype == 11))

```
PLI{<c>}{<q>} [<Rn>, {+/-}<Rm> {, <shift> #<amount>}]
```

```
n = UInt(Rn);  m = UInt(Rm);  add = (U == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm5);
if m == 15 then UNPREDICTABLE;
```

T1



T1

```
PLI{<c>}{<q>} [<Rn>, {+}<Rm> {, LSL #<amount>}]
```

```
if Rn == '1111' then SEE "PLI (immediate, literal)";
n = UInt(Rn);  m = UInt(Rm);  add = TRUE;
(shift_t, shift_n) = (SRType_LSL, UInt(imm2));
if m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	For encoding A1: see *Standard assembler syntax fields*. <c> must be AL or omitted. For encoding T1: see *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rn>	Is the general-purpose base register, encoded in the "Rn" field.
+/-	Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

+ Specifies the index register is added to the base register.

<Rm> Is the general-purpose index register, encoded in the "Rm" field.

<shift> Is the type of shift to be applied to the index register, encoded in “stype”:

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<amount> For encoding A1: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm5" field as <amount> modulo 32.

For encoding T1: is the shift amount, in the range 0 to 3, defaulting to 0 and encoded in the "imm2" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset = Shift(R[m], shift_t, shift_n, PSTATE.C);
    address = if add then (R[n] + offset) else (R[n] - offset);
    Hint_PreloadInstr(address);
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

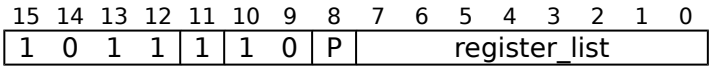
POP

Pop Multiple Registers from Stack loads multiple general-purpose registers from the stack, loading from consecutive memory locations starting at the address in SP, and updates SP to point just above the loaded data.

The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. See also *Encoding of lists of general-purpose registers and the PC*.

The registers loaded can include the PC, causing a branch to a loaded address. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*.

T1



T1

```
POP{<c>}{<q>} <registers> // (Preferred syntax)

LDM{<c>}{<q>} SP!, <registers> // (Alternate syntax)
```

```
registers = P:'0000000':register_list;   UnalignedAllowed = FALSE;
if BitCount(registers) < 1 then UNPREDICTABLE;
if registers<15> == '1' && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction targets an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers loaded.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<registers>	Is a list of one or more registers to be loaded, separated by commas and surrounded by { and }. The registers in the list must be in the range R0-R7, encoded in the "register_list" field, and can optionally include the PC. If the PC is in the list, the "P" field is set to 1, otherwise this field defaults to 0. If the PC is in the list, the instruction must be either outside any IT block, or the last instruction in an IT block.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = SP;
    for i = 0 to 14
        if registers<i> == '1' then
            R[i] = if UnalignedAllowed then MemU[address,4] else MemA[address,4];
            address = address + 4;
    if registers<15> == '1' then
        if UnalignedAllowed then
            if address<1:0> == '00' then
                LoadWritePC(MemU[address,4]);
            else
                UNPREDICTABLE;
        else
            LoadWritePC(MemA[address,4]);
    if registers<13> == '0' then SP = SP + 4*BitCount(registers);
    if registers<13> == '1' then SP = bits(32) UNKNOWN;
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

POP (multiple registers)

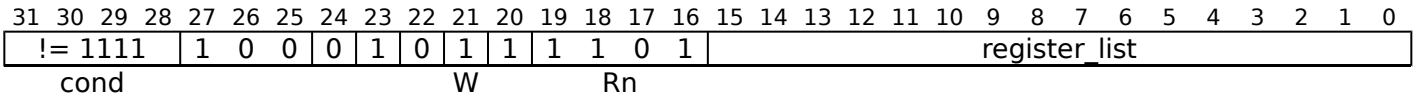
Pop Multiple Registers from Stack loads multiple general-purpose registers from the stack, loading from consecutive memory locations starting at the address in SP, and updates SP to point just above the loaded data

This is an alias of LDM, LDMIA, LDMFD. This means:

- The encodings in this description are named to match the encodings of LDM, LDMIA, LDMFD.
- The description of LDM, LDMIA, LDMFD gives the operational pseudocode for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T2) .

A1



A1

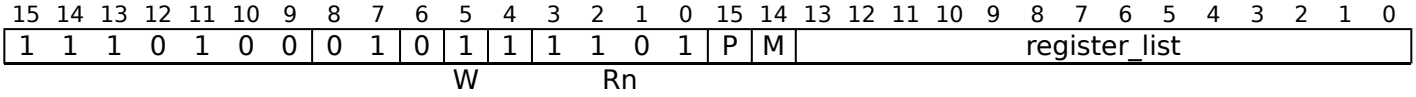
```
POP{<c>}{<q>} <registers>
```

is equivalent to

```
LDM{<c>}{<q>} SP!, <registers>
```

and is the preferred disassembly when `BitCount(register_list) > 1`.

T2



T2

```
POP{<c>}.W <registers> // (All registers in R0-R7, PC)
```

```
POP{<c>}{<q>} <registers>
```

is equivalent to

```
LDM{<c>}{<q>} SP!, <registers>
```

and is the preferred disassembly when `BitCount(P:M:register_list) > 1`.

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<registers> For encoding A1: is a list of two or more registers to be loaded, separated by commas and surrounded by { and }. The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. See also *Encoding of lists of general-purpose registers and the PC*.

If the SP is in the list, the value of the SP after such an instruction is UNKNOWN.

The PC can be in the list. If it is, the instruction branches to the address loaded to the PC. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*.

Arm deprecates the use of this instruction with both the LR and the PC in the list.

For encoding T2: is a list of two or more registers to be loaded, separated by commas and surrounded by { and }. The lowest-numbered register is loaded from the lowest memory address, through to the

highest-numbered register from the highest memory address. See also *Encoding of lists of general-purpose registers and the PC*.

The registers in the list must be in the range R0-R12, encoded in the "register_list" field, and can optionally contain one of the LR or the PC. If the LR is in the list, the "M" field is set to 1, otherwise it defaults to 0. If the PC is in the list, the "P" field is set to 1, otherwise it defaults to 0.

The PC can be in the list. If it is, the instruction branches to the address loaded to the PC. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*. If the PC is in the list:

- The LR must not be in the list.
- The instruction must be either outside any IT block, or the last instruction in an IT block.

Operation

The description of LDM, LDMIA, LDMFD gives the operational pseudocode for this instruction.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

POP (single register)

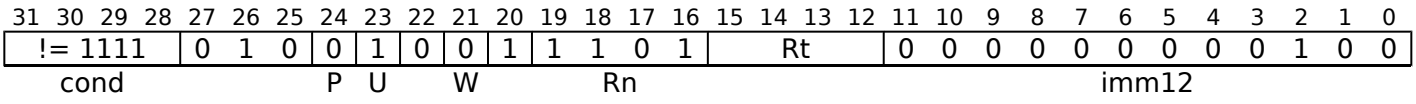
Pop Single Register from Stack loads a single general-purpose register from the stack, loading from the address in SP, and updates SP to point just above the loaded data

This is an alias of LDR (immediate). This means:

- The encodings in this description are named to match the encodings of LDR (immediate).
- The description of LDR (immediate) gives the operational pseudocode for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T4) .

A1



Post-indexed

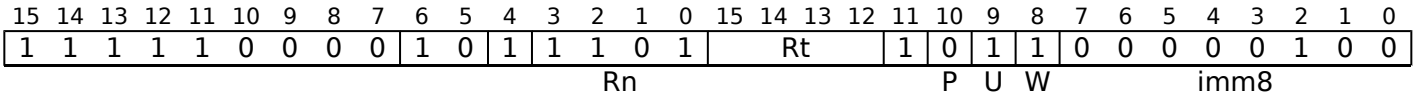
```
POP{<c>}{<q>} <single_register_list>
```

is equivalent to

```
LDR{<c>}{<q>} <Rt>, [SP], #4
```

and is always the preferred disassembly.

T4



Post-indexed

```
POP{<c>}{<q>} <single_register_list>
```

is equivalent to

```
LDR{<c>}{<q>} <Rt>, [SP], #4
```

and is always the preferred disassembly.

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<single_register_list> Is the general-purpose register <Rt> to be loaded surrounded by { and }.

<Rt> For encoding A1: is the general-purpose register to be transferred, encoded in the "Rt" field. The PC can be used. If the PC is used, the instruction branches to the address (data) loaded to the PC. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*.

For encoding T4: is the general-purpose register to be transferred, encoded in the "Rt" field. The PC can be used, provided the instruction is either outside an IT block or the last instruction of an IT block. If the PC is used, the instruction branches to the address (data) loaded to the PC. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*.

Operation

The description of LDR (immediate) gives the operational pseudocode for this instruction.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

PSSBB

Physical Speculative Store Bypass Barrier is a memory barrier which prevents speculative loads from bypassing earlier stores to the same physical address.

The semantics of the Physical Speculative Store Bypass Barrier are:

- When a load to a location appears in program order after the PSSBB, then the load does not speculatively read an entry earlier in the coherence order for that location than the entry generated by the latest store satisfying all of the following conditions:
  - The store is to the same location as the load.
  - The store appears in program order before the PSSBB.
- When a load to a location appears in program order before the PSSBB, then the load does not speculatively read data from any store satisfying all of the following conditions:
  - The store is to the same location as the load.
  - The store appears in program order after the PSSBB.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	1	0	1	1	1	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	0	1	0	0	0	1	0	0

A1

```
PSSBB{<q>}

// No additional decoding required
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	1	(1)	(1)	(1)	(1)	1	0	(0)	0	(1)	(1)	(1)	(1)	0	1	0	0	0	1	0	0

T1

```
PSSBB{<q>}

if InITBlock() then UNPREDICTABLE;
```

Assembler Symbols

<q>	See *Standard assembler syntax fields*.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    SpeculativeStoreBypassBarrierToPA();
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

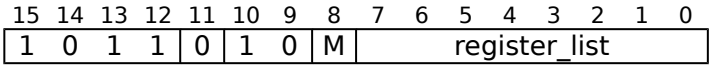
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

PUSH

Push Multiple Registers to Stack stores multiple general-purpose registers to the stack, storing to consecutive memory locations ending just below the address in SP, and updates SP to point to the start of the stored data.

The lowest-numbered register is stored to the lowest memory address, through to the highest-numbered register to the highest memory address. See also *Encoding of lists of general-purpose registers and the PC*.

T1



T1

```
PUSH{<c>}{<q>} <registers> // (Preferred syntax)

STMDB{<c>}{<q>} SP!, <registers> // (Alternate syntax)
```

```
registers = '0':M:'000000':register_list;  UnalignedAllowed = FALSE;
if BitCount(registers) < 1 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction targets an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers loaded.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<registers>	Is a list of one or more registers to be stored, separated by commas and surrounded by { and }. The registers in the list must be in the range R0-R7, encoded in the "register_list" field, and can optionally include the LR. If the LR is in the list, the "M" field is set to 1, otherwise this field defaults to 0.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = SP - 4*BitCount(registers);
    for i = 0 to 14
        if registers<i> == '1' then
            if i == 13 && i != LowestSetBit(registers) then  // Only possible for encoding A1
                MemA[address,4] = bits(32) UNKNOWN;
            else
                if UnalignedAllowed then
                    MemU[address,4] = R[i];
                else
                    MemA[address,4] = R[i];
            address = address + 4;
    if registers<15> == '1' then  // Only possible for encoding A1 or A2
        if UnalignedAllowed then
            MemU[address,4] = PCStoreValue();
        else
            MemA[address,4] = PCStoreValue();
    SP = SP - 4*BitCount(registers);
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

PUSH (multiple registers)

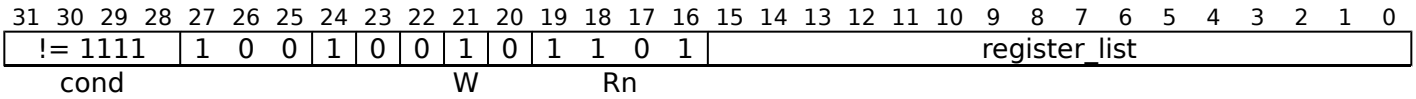
Push multiple registers to Stack stores multiple general-purpose registers to the stack, storing to consecutive memory locations ending just below the address in SP, and updates SP to point to the start of the stored data

This is an alias of STMDB, STMFD. This means:

- The encodings in this description are named to match the encodings of STMDB, STMFD.
- The description of STMDB, STMFD gives the operational pseudocode for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



A1

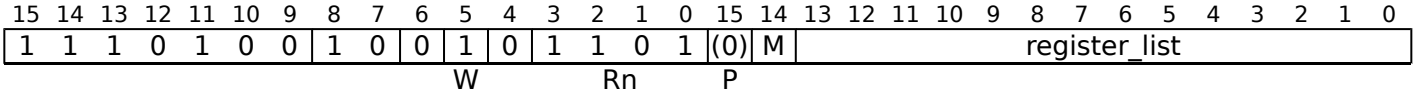
```
PUSH{<c>}{<q>} <registers>
```

is equivalent to

```
STMDB{<c>}{<q>} SP!, <registers>
```

and is the preferred disassembly when `BitCount(register_list) > 1`.

T1



T1

```
PUSH{<c>}.W <registers> // (All registers in R0-R7, LR)
```

```
PUSH{<c>}{<q>} <registers>
```

is equivalent to

```
STMDB{<c>}{<q>} SP!, <registers>
```

and is the preferred disassembly when `BitCount(M:register_list) > 1`.

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<registers> For encoding A1: is a list of two or more registers to be stored, separated by commas and surrounded by { and }. The lowest-numbered register is stored to the lowest memory address, through to the highest-numbered register to the highest memory address. See also *Encoding of lists of general-purpose registers and the PC*.

The SP and PC can be in the list. However:

- Arm deprecates the use of instructions that include the PC in the list.
- If the SP is in the list, and it is not the lowest-numbered register in the list, the instruction stores an UNKNOWN value for the SP.

For encoding T1: is a list of one or more registers to be stored, separated by commas and surrounded by { and }. The lowest-numbered register is stored to the lowest memory address, through to the

highest-numbered register to the highest memory address. See also *Encoding of lists of general-purpose registers and the PC*.

The registers in the list must be in the range R0-R12, encoded in the "register_list" field, and can optionally contain the LR. If the LR is in the list, the "M" field is set to 1, otherwise it defaults to 0.

Operation

The description of STMDB, STMFD gives the operational pseudocode for this instruction.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

PUSH (single register)

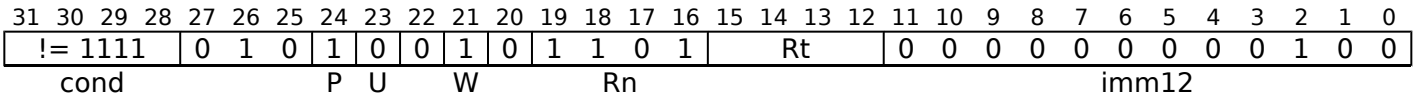
Push Single Register to Stack stores a single general-purpose register to the stack, storing to the 32-bit word below the address in SP, and updates SP to point to the start of the stored data

This is an alias of STR (immediate). This means:

- The encodings in this description are named to match the encodings of STR (immediate).
- The description of STR (immediate) gives the operational pseudocode for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T4) .

A1



Pre-indexed

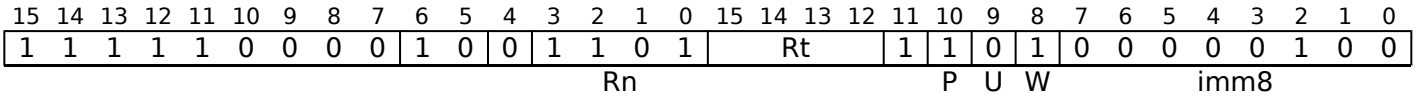
```
PUSH{<c>}{<q>} <single_register_list>
```

is equivalent to

```
STR{<c>}{<q>} <Rt>, [SP, #-4]!
```

and is always the preferred disassembly.

T4



Pre-indexed

```
PUSH{<c>}{<q>} <single_register_list> // (Standard syntax)
```

is equivalent to

```
STR{<c>}{<q>} <Rt>, [SP, #-4]!
```

and is always the preferred disassembly.

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<single_register_list> Is the general-purpose register <Rt> to be stored surrounded by { and }.

<Rt> For encoding A1: is the general-purpose register to be transferred, encoded in the "Rt" field. The PC can be used, but this is deprecated.

For encoding T4: is the general-purpose register to be transferred, encoded in the "Rt" field.

Operation

The description of STR (immediate) gives the operational pseudocode for this instruction.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

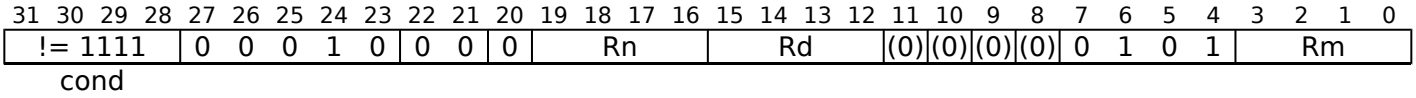
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

QADD

Saturating Add adds two register values, saturates the result to the 32-bit signed integer range -2^{31} to $(2^{31} - 1)$, and writes the result to the destination register. If saturation occurs, it sets *PSTATE*.Q to 1.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

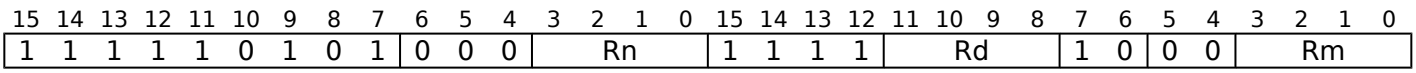
A1



A1

```
QADD{<c>}{<q>} {<Rd>,} <Rm>, <Rn>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1



T1

```
QADD{<c>}{<q>} {<Rd>,} <Rm>, <Rn>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rm>	Is the first general-purpose source register, encoded in the "Rm" field.
<Rn>	Is the second general-purpose source register, encoded in the "Rn" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (R[d], sat) = SignedSatQ(SInt(R[m]) + SInt(R[n]), 32);
    if sat then
        PSTATE.Q = '1';
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

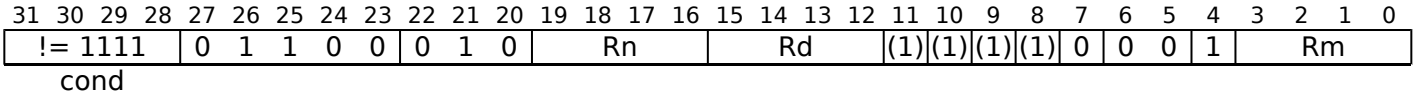
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

QADD16

Saturating Add 16 performs two 16-bit integer additions, saturates the results to the 16-bit signed integer range -2^{15} <= x <= 2^{15} - 1, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

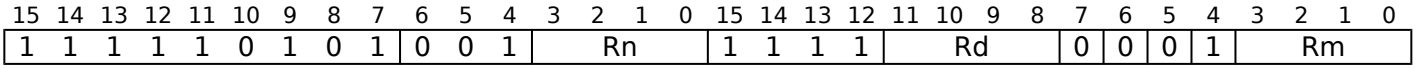
A1



A1

```
QADD16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1



T1

```
QADD16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = SInt(R[n]<15:0>) + SInt(R[m]<15:0>);
    sum2 = SInt(R[n]<31:16>) + SInt(R[m]<31:16>);
    R[d]<15:0>  = SignedSat(sum1, 16);
    R[d]<31:16> = SignedSat(sum2, 16);
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

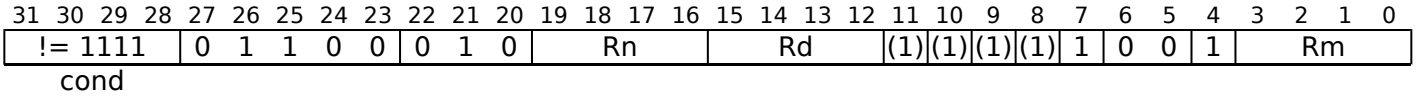
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

QADD8

Saturating Add 8 performs four 8-bit integer additions, saturates the results to the 8-bit signed integer range -2^7 <= x <= 2^7 - 1, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

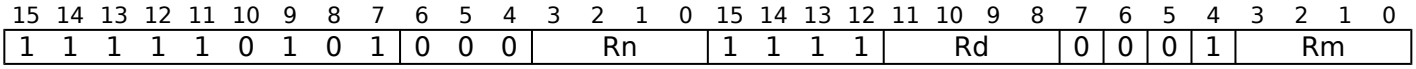
A1



A1

```
QADD8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1



T1

```
QADD8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = SInt(R[n]<7:0>) + SInt(R[m]<7:0>);
    sum2 = SInt(R[n]<15:8>) + SInt(R[m]<15:8>);
    sum3 = SInt(R[n]<23:16>) + SInt(R[m]<23:16>);
    sum4 = SInt(R[n]<31:24>) + SInt(R[m]<31:24>);
    R[d]<7:0>   = SignedSat(sum1, 8);
    R[d]<15:8>  = SignedSat(sum2, 8);
    R[d]<23:16> = SignedSat(sum3, 8);
    R[d]<31:24> = SignedSat(sum4, 8);
```

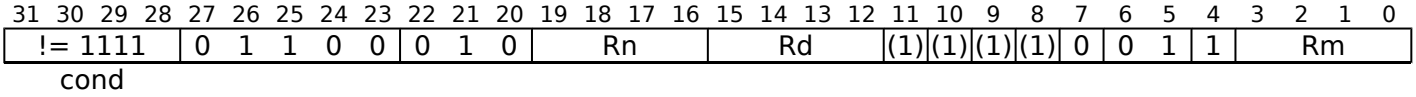

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

QASX

Saturating Add and Subtract with Exchange exchanges the two halfwords of the second operand, performs one 16-bit integer addition and one 16-bit subtraction, saturates the results to the 16-bit signed integer range $-2^{15} <= x <= 2^{15} - 1$, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

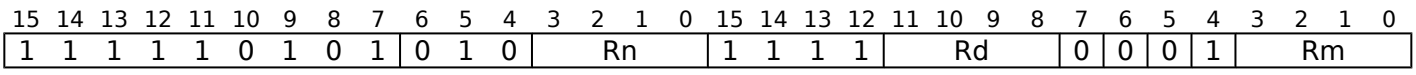


A1

```
QASX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1



T1

```
QASX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff = SInt(R[n]<15:0>) - SInt(R[m]<31:16>);
    sum  = SInt(R[n]<31:16>) + SInt(R[m]<15:0>);
    R[d]<15:0>  = SignedSat(diff, 16);
    R[d]<31:16> = SignedSat(sum, 16);
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

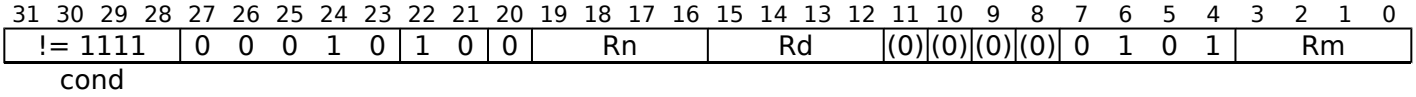
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

QDADD

Saturating Double and Add adds a doubled register value to another register value, and writes the result to the destination register. Both the doubling and the addition have their results saturated to the 32-bit signed integer range $-2^{31} <= x <= 2^{31} - 1$. If saturation occurs in either operation, it sets *PSTATE*.Q to 1.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

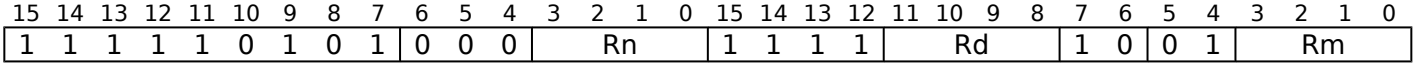
A1



A1

```
QDADD{<c>}{<q>} {<Rd>,} <Rm>, <Rn>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1



T1

```
QDADD{<c>}{<q>} {<Rd>,} <Rm>, <Rn>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rm>	Is the first general-purpose source register, encoded in the "Rm" field.
<Rn>	Is the second general-purpose source register, encoded in the "Rn" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (doubled, sat1) = SignedSatQ(2 * SInt(R[n]), 32);
    (R[d], sat2)  = SignedSatQ(SInt(R[m]) + SInt(doubled), 32);
    if sat1 || sat2 then
        PSTATE.Q = '1';
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

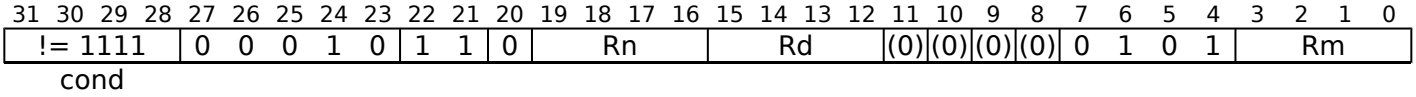
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

QDSUB

Saturating Double and Subtract subtracts a doubled register value from another register value, and writes the result to the destination register. Both the doubling and the subtraction have their results saturated to the 32-bit signed integer range -2^{31} <= x <= 2^{31} - 1. If saturation occurs in either operation, it sets *PSTATE*.Q to 1.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

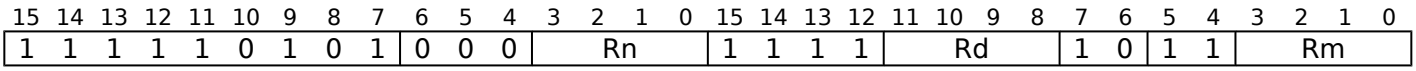


A1

```
QDSUB{<c>}{<q>} {<Rd>,} <Rm>, <Rn>
```

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1



T1

```
QDSUB{<c>}{<q>} {<Rd>,} <Rm>, <Rn>
```

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rm>	Is the first general-purpose source register, encoded in the "Rm" field.
<Rn>	Is the second general-purpose source register, encoded in the "Rn" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (doubled, sat1) = SignedSatQ(2 * SInt(R[n]), 32);
    (R[d], sat2)  = SignedSatQ(SInt(R[m]) - SInt(doubled), 32);
    if sat1 || sat2 then
        PSTATE.Q = '1';
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

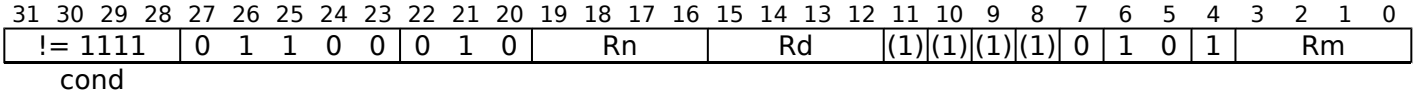
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

QSAX

Saturating Subtract and Add with Exchange exchanges the two halfwords of the second operand, performs one 16-bit integer subtraction and one 16-bit addition, saturates the results to the 16-bit signed integer range $-2^{15} <= x <= 2^{15} - 1$, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

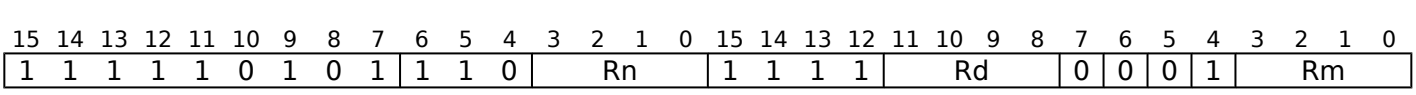


A1

```
QSAX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1



T1

```
QSAX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum  = SInt(R[n]<15:0>) + SInt(R[m]<31:16>);
    diff = SInt(R[n]<31:16>) - SInt(R[m]<15:0>);
    R[d]<15:0>  = SignedSat(sum, 16);
    R[d]<31:16> = SignedSat(diff, 16);
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

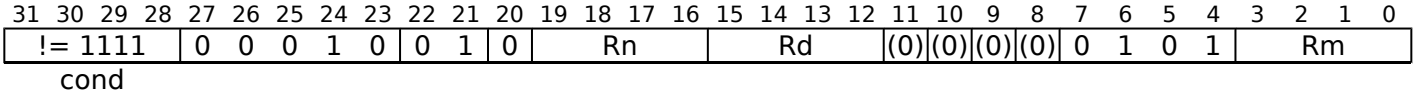
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

QSUB

Saturating Subtract subtracts one register value from another register value, saturates the result to the 32-bit signed integer range $-2^{31} <= x <= 2^{31} - 1$, and writes the result to the destination register. If saturation occurs, it sets *PSTATE*.Q to 1.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

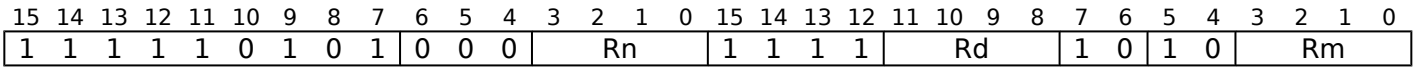


A1

```
QSUB{<c>}{<q>} {<Rd>,} <Rm>, <Rn>
```

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1



T1

```
QSUB{<c>}{<q>} {<Rd>,} <Rm>, <Rn>
```

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rm>	Is the first general-purpose source register, encoded in the "Rm" field.
<Rn>	Is the second general-purpose source register, encoded in the "Rn" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (R[d], sat) = SignedSatQ(SInt(R[m]) - SInt(R[n]), 32);
    if sat then
        PSTATE.Q = '1';
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

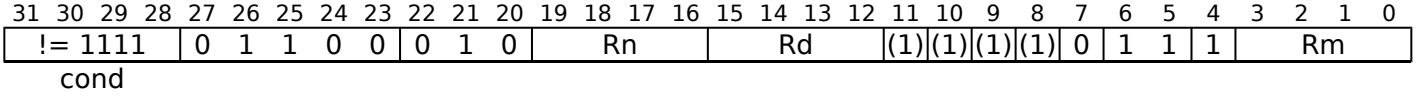
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

QSUB16

Saturating Subtract 16 performs two 16-bit integer subtractions, saturates the results to the 16-bit signed integer range $-2^{15} <= x <= 2^{15} - 1$, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

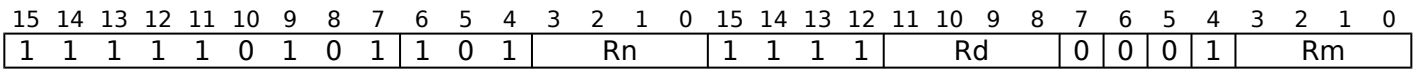
A1



A1

```
QSUB16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1



T1

```
QSUB16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = SInt(R[n]<15:0>) - SInt(R[m]<15:0>);
    diff2 = SInt(R[n]<31:16>) - SInt(R[m]<31:16>);
    R[d]<15:0>  = SignedSat(diff1, 16);
    R[d]<31:16> = SignedSat(diff2, 16);
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

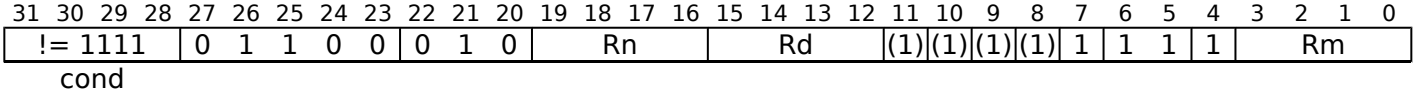
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

QSUB8

Saturating Subtract 8 performs four 8-bit integer subtractions, saturates the results to the 8-bit signed integer range $-2^7 <= x <= 2^7 - 1$, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

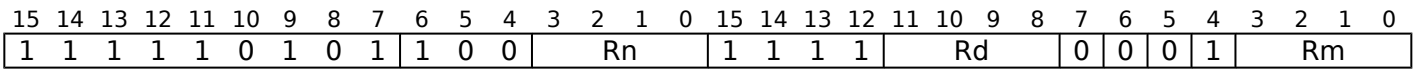
A1



A1

```
QSUB8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1



T1

```
QSUB8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = SInt(R[n]<7:0>) - SInt(R[m]<7:0>);
    diff2 = SInt(R[n]<15:8>) - SInt(R[m]<15:8>);
    diff3 = SInt(R[n]<23:16>) - SInt(R[m]<23:16>);
    diff4 = SInt(R[n]<31:24>) - SInt(R[m]<31:24>);
    R[d]<7:0>   = SignedSat(diff1, 8);
    R[d]<15:8>  = SignedSat(diff2, 8);
    R[d]<23:16> = SignedSat(diff3, 8);
    R[d]<31:24> = SignedSat(diff4, 8);
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

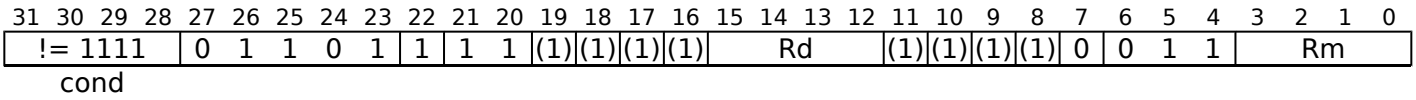
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

RBIT

Reverse Bits reverses the bit order in a 32-bit register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

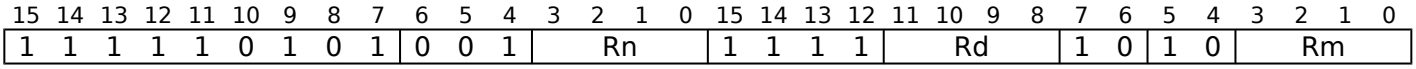


A1

```
RBIT{<c>}{<q>} <Rd>, <Rm>
```

```
d = UInt(Rd);  m = UInt(Rm);
if d == 15 || m == 15 then UNPREDICTABLE;
```

T1



T1

```
RBIT{<c>}{<q>} <Rd>, <Rm>
```

```
d = UInt(Rd);  m = UInt(Rm);  n = UInt(Rn);
if m != n || d == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

CONSTRAINED UNPREDICTABLE behavior

If `m != n`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes with the additional decode: m = UInt(Rn);.
- The instruction executes with the additional decode: m = UInt(Rm);.
- The value in the destination register is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rm>	For encoding A1: is the general-purpose source register, encoded in the "Rm" field. For encoding T1: is the general-purpose source register, encoded in the "Rm" field. It must be encoded with an identical value in the "Rn" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    bits(32) result;
    for i = 0 to 31
        result<31-i> = R[m]<i>;
    R[d] = result;
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

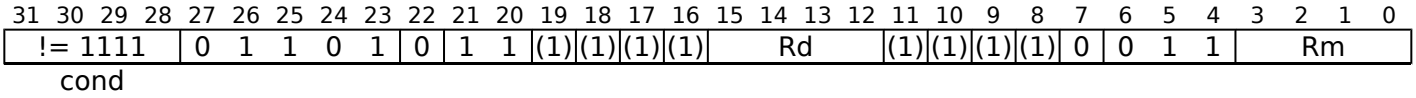
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

REV

Byte-Reverse Word reverses the byte order in a 32-bit register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2) .

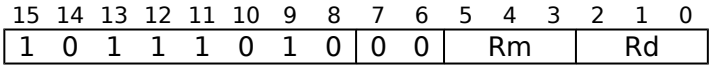
A1



A1

```
REV{<c>}{<q>} <Rd>, <Rm>

d = UInt(Rd);  m = UInt(Rm);
if d == 15 || m == 15 then UNPREDICTABLE;
```

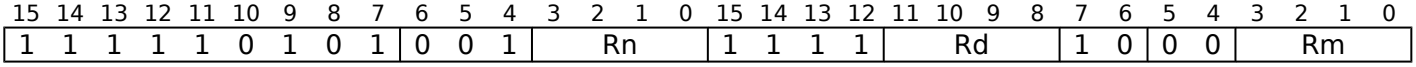
T1



T1

```
REV{<c>}{<q>} <Rd>, <Rm>

d = UInt(Rd);  m = UInt(Rm);
```

T2



T2

```
REV{<c>}.W <Rd>, <Rm> // (<Rd>, <Rm> can be represented in T1)

REV{<c>}{<q>} <Rd>, <Rm>

d = UInt(Rd);  m = UInt(Rm);  n = UInt(Rn);
if m != n || d == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

CONSTRAINED UNPREDICTABLE behavior

If `m != n`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes with the additional decode: m = UInt(Rn);.
- The instruction executes with the additional decode: m = UInt(Rm);.
- The value in the destination register is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q>	See *Standard assembler syntax fields*.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rm>	For encoding A1 and T1: is the general-purpose source register, encoded in the "Rm" field. For encoding T2: is the general-purpose source register, encoded in the "Rm" field. It must be encoded with an identical value in the "Rn" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    bits(32) result;
    result<31:24> = R[m]<7:0>;
    result<23:16> = R[m]<15:8>;
    result<15:8>  = R[m]<23:16>;
    result<7:0>   = R[m]<31:24>;
    R[d] = result;
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

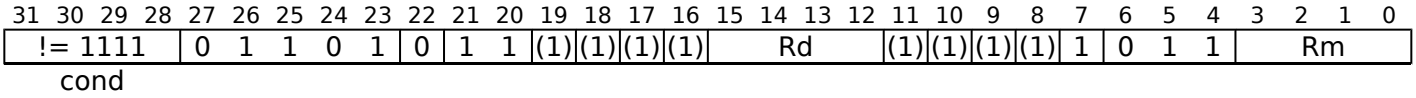
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

REV16

Byte-Reverse Packed Halfword reverses the byte order in each16-bit halfword of a 32-bit register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2) .

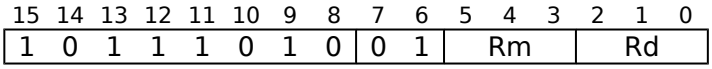
A1



A1

```
REV16{<c>}{<q>} <Rd>, <Rm>

d = UInt(Rd);  m = UInt(Rm);
if d == 15 || m == 15 then UNPREDICTABLE;
```

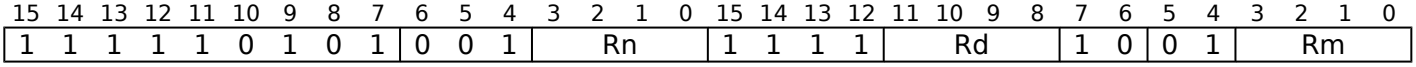
T1



T1

```
REV16{<c>}{<q>} <Rd>, <Rm>

d = UInt(Rd);  m = UInt(Rm);
```

T2



T2

```
REV16{<c>}.W <Rd>, <Rm> // (<Rd>, <Rm> can be represented in T1)

REV16{<c>}{<q>} <Rd>, <Rm>

d = UInt(Rd);  m = UInt(Rm);  n = UInt(Rn);
if m != n || d == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

CONSTRAINED UNPREDICTABLE behavior

If `m != n`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: m = UInt(Rn);.
- The instruction executes with the additional decode: m = UInt(Rm);.
- The value in the destination register is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q>	See *Standard assembler syntax fields*.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rm>	For encoding A1 and T1: is the general-purpose source register, encoded in the "Rm" field. For encoding T2: is the general-purpose source register, encoded in the "Rm" field. It must be encoded with an identical value in the "Rn" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    bits(32) result;
    result<31:24> = R[m]<23:16>;
    result<23:16> = R[m]<31:24>;
    result<15:8>  = R[m]<7:0>;
    result<7:0>   = R[m]<15:8>;
    R[d] = result;
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

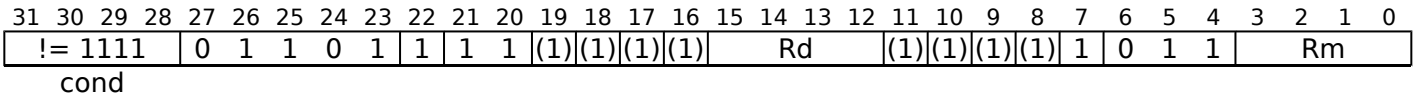
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

REVSH

Byte-Reverse Signed Halfword reverses the byte order in the lower 16-bit halfword of a 32-bit register, and sign-extends the result to 32 bits.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2) .

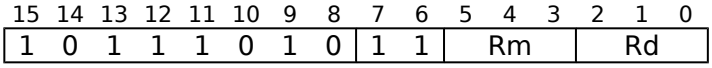
A1



A1

```
REVSH{<c>}{<q>} <Rd>, <Rm>

d = UInt(Rd);  m = UInt(Rm);
if d == 15 || m == 15 then UNPREDICTABLE;
```

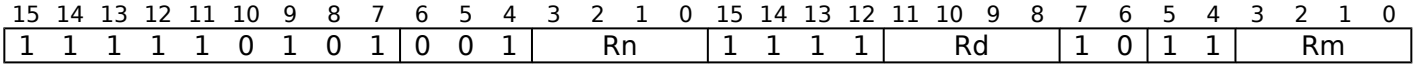
T1



T1

```
REVSH{<c>}{<q>} <Rd>, <Rm>

d = UInt(Rd);  m = UInt(Rm);
```

T2



T2

```
REVSH{<c>}.W <Rd>, <Rm> // (<Rd>, <Rm> can be represented in T1)

REVSH{<c>}{<q>} <Rd>, <Rm>

d = UInt(Rd);  m = UInt(Rm);  n = UInt(Rn);
if m != n || d == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

CONSTRAINED UNPREDICTABLE behavior

If `m != n`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: m = UInt(Rn);.
- The instruction executes with the additional decode: m = UInt(Rm);.
- The value in the destination register is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q>	See *Standard assembler syntax fields*.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rm>	For encoding A1 and T1: is the general-purpose source register, encoded in the "Rm" field. For encoding T2: is the general-purpose source register, encoded in the "Rm" field. It must be encoded with an identical value in the "Rn" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    bits(32) result;
    result<31:8>  = SignExtend(R[m]<7:0>, 24);
    result<7:0>   = R[m]<15:8>;
    R[d] = result;
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

RFE, RFEDA, RFEDB, RFEIA, RFEIB

Return From Exception loads two consecutive memory locations using an address in a base register:

- The word loaded from the lower address is treated as an instruction address. The PE branches to it.
- The word loaded from the higher address is used to restore *PSTATE*. This word must be in the format of an SPSR.

An address adjusted by the size of the data loaded can optionally be written back to the base register.

The PE checks the value of the word loaded from the higher address for an illegal return event. See *Illegal return events from AArch32 state*.

RFE is UNDEFINED in Hyp mode and CONSTRAINED UNPREDICTABLE in User mode.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	P	U	0	W	1	Rn				(0)	(0)	(0)	(0)	(1)	(0)	(1)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)

Decrement After (P == 0 && U == 0)

```
RFEDA{<c>}{<q>} <Rn>{!} // (Preferred syntax)

RFEFA{<c>}{<q>} <Rn>{!} // (Alternate syntax, Full Ascending stack)
```

Decrement Before (P == 1 && U == 0)

```
RFEDB{<c>}{<q>} <Rn>{!} // (Preferred syntax)

RFEEA{<c>}{<q>} <Rn>{!} // (Alternate syntax, Empty Ascending stack)
```

Increment After (P == 0 && U == 1)

```
RFE{IA}{<c>}{<q>} <Rn>{!} // (Preferred syntax)

RFEFD{<c>}{<q>} <Rn>{!} // (Alternate syntax, Full Descending stack)
```

Increment Before (P == 1 && U == 1)

```
RFEIB{<c>}{<q>} <Rn>{!} // (Preferred syntax)

RFEED{<c>}{<q>} <Rn>{!} // (Alternate syntax, Empty Descending stack)
```

```
n = UInt(Rn);
wback = (W == '1');  increment = (U == '1');  wordhigher = (P == U);
if n == 15 then UNPREDICTABLE;
```

T1

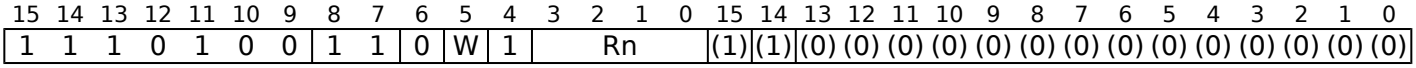
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	0	0	0	W	1	Rn				(1)	(1)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)

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



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, RFEEA, RFEFD, and RFEED are pseudo-instructions for RFEDA, RFEDB, RFEIA, and RFEIB respectively, referring to their use for popping data from Full Ascending, Empty Ascending, Full Descending, and Empty Descending stacks.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if PSTATE.EL == EL2 then
        UNDEFINED;
    elsif PSTATE.EL == EL0 then
        UNPREDICTABLE;                          // UNDEFINED or NOP
    else
        address = if increment then R[n] else R[n]-8;
        if wordhigher then address = address+4;
        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_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ROR (immediate)

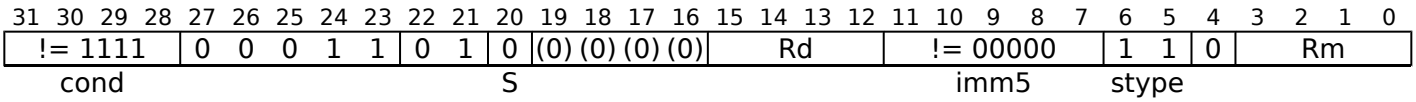
Rotate Right (immediate) provides the value of the contents of a register rotated by a constant value. The bits that are rotated off the right end are inserted into the vacated bit positions on the left.

This is an alias of MOV, MOVS (register). This means:

- The encodings in this description are named to match the encodings of MOV, MOVS (register).
- The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T3) .

A1



MOV, shift or rotate by value

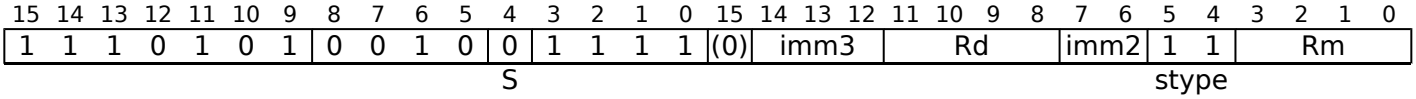
```
ROR{<c>}{<q>} {<Rd>,} <Rm>, #<imm>
```

is equivalent to

```
MOV{<c>}{<q>} <Rd>, <Rm>, ROR #<imm>
```

and is always the preferred disassembly.

T3



MOV, shift or rotate by value (!(imm3 == 000 && imm2 == 00))

```
ROR{<c>}{<q>} {<Rd>,} <Rm>, #<imm>
```

is equivalent to

```
MOV{<c>}{<q>} <Rd>, <Rm>, ROR #<imm>
```

and is always the preferred disassembly.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rd>	For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. Arm deprecates using the PC as the destination register, but if the PC is used, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*.
	For encoding T3: is the general-purpose destination register, encoded in the "Rd" field.
<Rm>	For encoding A1: is the general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.
	For encoding T3: is the general-purpose source register, encoded in the "Rm" field.
<imm>	For encoding A1: is the shift amount, in the range 1 to 31, encoded in the "imm5" field.
	For encoding T3: is the shift amount, in the range 1 to 31, encoded in the "imm3:imm2" field.

Operation

The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ROR (register)

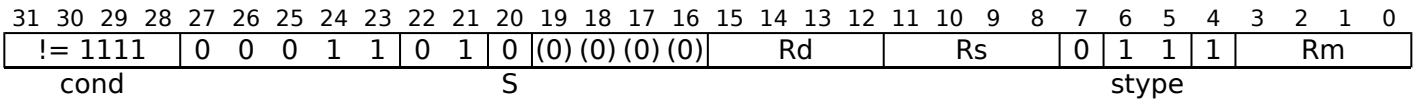
Rotate Right (register) provides the value of the contents of a register rotated by a variable number of bits. The bits that are rotated off the right end are inserted into the vacated bit positions on the left. The variable number of bits is read from the bottom byte of a register

This is an alias of MOV, MOVS (register-shifted register). This means:

- The encodings in this description are named to match the encodings of MOV, MOVS (register-shifted register).
- The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2) .

A1



Not flag setting

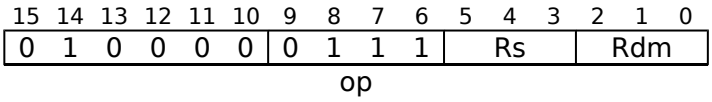
```
ROR{<c>}{<q>} {<Rd>,} <Rm>, <Rs>
```

is equivalent to

```
MOV{<c>}{<q>} <Rd>, <Rm>, ROR <Rs>
```

and is always the preferred disassembly.

T1



Rotate right

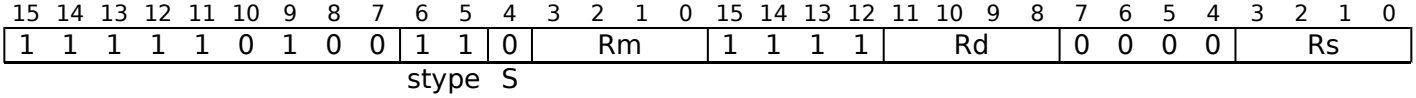
```
ROR<c>{<q>} {<Rdm>,} <Rdm>, <Rs> // (Inside IT block)
```

is equivalent to

```
MOV<c>{<q>} <Rdm>, <Rdm>, ROR <Rs>
```

and is the preferred disassembly when `InITBlock()`.

T2



Not flag setting

```
ROR<c>.W {<Rd>,} <Rm>, <Rs> // (Inside IT block, and <Rd>, <Rm>, <shift>, <Rs> can be represented in T1)
ROR{<c>}{<q>} {<Rd>,} <Rm>, <Rs>
```

is equivalent to

```
MOV{<c>}{<q>} <Rd>, <Rm>, ROR <Rs>
```

and is always the preferred disassembly.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rdm>	Is the first general-purpose source register and the destination register, encoded in the "Rdm" field.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rm>	Is the first general-purpose source register, encoded in the "Rm" field.
<Rs>	Is the second general-purpose source register holding a rotate amount in its bottom 8 bits, encoded in the "Rs" field.

Operation

The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

RORS (immediate)

Rotate Right, setting flags (immediate) provides the value of the contents of a register rotated by a constant value. The bits that are rotated off the right end are inserted into the vacated bit positions on the left.

If the destination register is not the PC, this instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

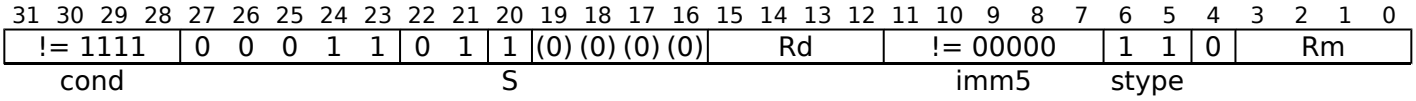
- The PE branches to the address written to the PC, and restores *PSTATE* from SPSR_<current_mode>.
- The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state*.
- The instruction is UNDEFINED in Hyp mode.
- The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

This is an alias of MOV, MOVS (register). This means:

- The encodings in this description are named to match the encodings of MOV, MOVS (register).
- The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T3) .

A1



MOVS, shift or rotate by value

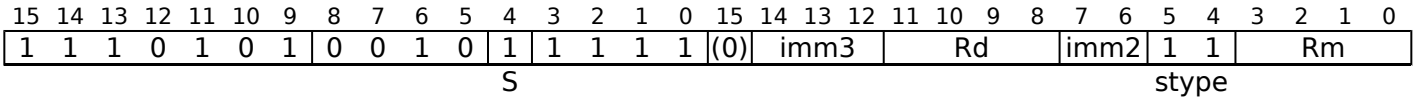
```
RORS{<c>}{<q>} {<Rd>,} <Rm>, #<imm>
```

is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, ROR #<imm>
```

and is always the preferred disassembly.

T3



MOVS, shift or rotate by value (!(imm3 == 000 && imm2 == 00))

```
RORS{<c>}{<q>} {<Rd>,} <Rm>, #<imm>
```

is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, ROR #<imm>
```

and is always the preferred disassembly.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rd>	For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. Arm deprecates using the PC as the destination register, but if the PC is used, the instruction performs an exception return, that restores *PSTATE* from SPSR_<current_mode>. For encoding T3: is the general-purpose destination register, encoded in the "Rd" field.

<Rm> For encoding A1: is the general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.

For encoding T3: is the general-purpose source register, encoded in the "Rm" field.

<imm> For encoding A1: is the shift amount, in the range 1 to 31, encoded in the "imm5" field.

For encoding T3: is the shift amount, in the range 1 to 31, encoded in the "imm3:imm2" field.

Operation

The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

RORS (register)

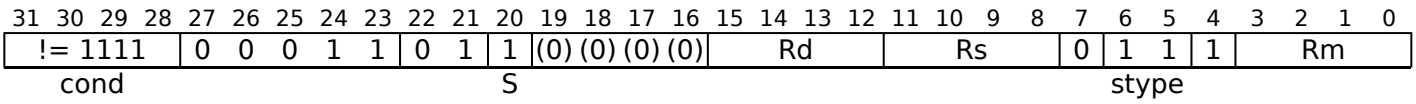
Rotate Right, setting flags (register) provides the value of the contents of a register rotated by a variable number of bits, and updates the condition flags based on the result. The bits that are rotated off the right end are inserted into the vacated bit positions on the left. The variable number of bits is read from the bottom byte of a register

This is an alias of MOV, MOVS (register-shifted register). This means:

- The encodings in this description are named to match the encodings of MOV, MOVS (register-shifted register).
- The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2) .

A1



Flag setting

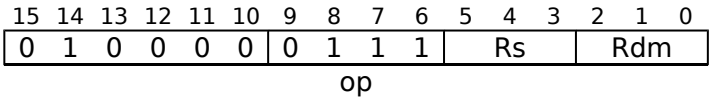
```
RORS{<c>}{<q>} {<Rd>,} <Rm>, <Rs>
```

is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, ROR <Rs>
```

and is always the preferred disassembly.

T1



Rotate right

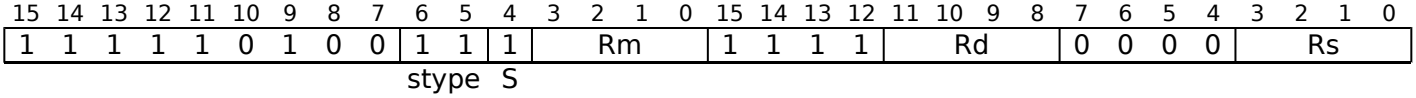
```
RORS{<q>} {<Rdm>,} <Rdm>, <Rs> // (Outside IT block)
```

is equivalent to

```
MOVS{<q>} <Rdm>, <Rdm>, ROR <Rs>
```

and is the preferred disassembly when `!InITBlock()`.

T2



Flag setting

```
RORS.W {<Rd>,} <Rm>, <Rs> // (Outside IT block, and <Rd>, <Rm>, <shift>, <Rs> can be represented in T1)
RORS{<c>}{<q>} {<Rd>,} <Rm>, <Rs>
```

is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, ROR <Rs>
```

and is always the preferred disassembly.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rdm>	Is the first general-purpose source register and the destination register, encoded in the "Rdm" field.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rm>	Is the first general-purpose source register, encoded in the "Rm" field.
<Rs>	Is the second general-purpose source register holding a rotate amount in its bottom 8 bits, encoded in the "Rs" field.

Operation

The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

RRX

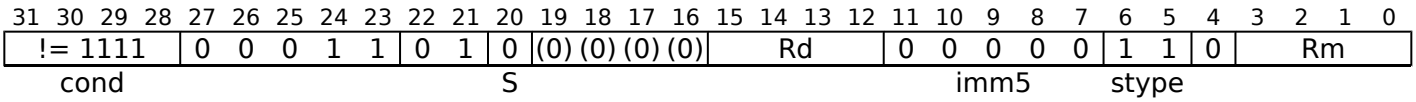
Rotate Right with Extend provides the value of the contents of a register shifted right by one place, with the Carry flag shifted into bit[31].

This is an alias of MOV, MOVS (register). This means:

- The encodings in this description are named to match the encodings of MOV, MOVS (register).
- The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T3) .

A1



MOV, rotate right with extend

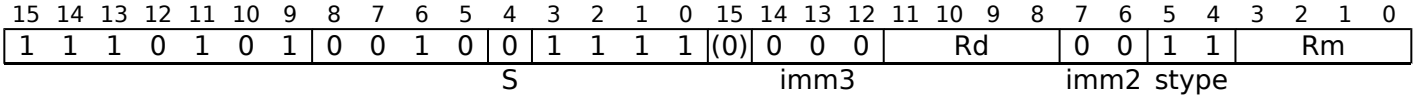
```
RRX{<c>}{<q>} {<Rd>,} <Rm>
```

is equivalent to

```
MOV{<c>}{<q>} <Rd>, <Rm>, RRX
```

and is always the preferred disassembly.

T3



MOV, rotate right with extend

```
RRX{<c>}{<q>} {<Rd>,} <Rm>
```

is equivalent to

```
MOV{<c>}{<q>} <Rd>, <Rm>, RRX
```

and is always the preferred disassembly.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rd>	For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. Arm deprecates using the PC as the destination register, but if the PC is used, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*.
	For encoding T3: is the general-purpose destination register, encoded in the "Rd" field.
<Rm>	For encoding A1: is the general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.
	For encoding T3: is the general-purpose source register, encoded in the "Rm" field.

Operation

The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

RRXS

Rotate Right with Extend, setting flags provides the value of the contents of a register shifted right by one place, with the Carry flag shifted into bit[31].

If the destination register is not the PC, this instruction updates the condition flags based on the result, and bit[0] is shifted into the Carry flag.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. Arm deprecates any use of these encodings. However, when the destination register is the PC:

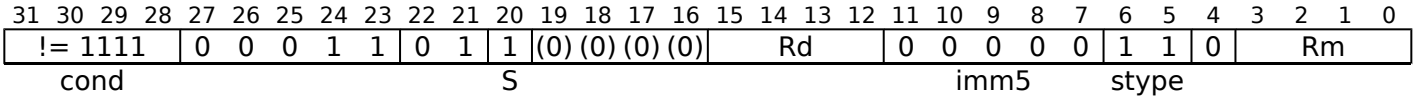
- The PE branches to the address written to the PC, and restores *PSTATE* from SPSR_<current_mode>.
- The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state*.
- The instruction is UNDEFINED in Hyp mode.
- The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

This is an alias of MOV, MOVS (register). This means:

- The encodings in this description are named to match the encodings of MOV, MOVS (register).
- The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T3) .

A1



MOVS, rotate right with extend

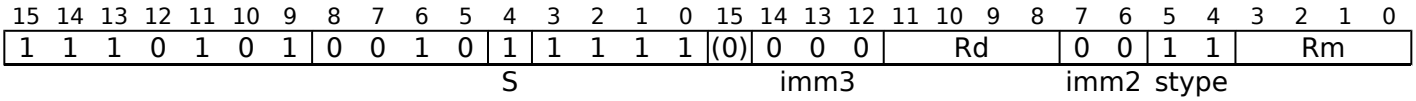
```
RRXS{<c>}{<q>} {<Rd>,} <Rm>
```

is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, RRX
```

and is always the preferred disassembly.

T3



MOVS, rotate right with extend

```
RRXS{<c>}{<q>} {<Rd>,} <Rm>
```

is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, RRX
```

and is always the preferred disassembly.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rd>	For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. Arm deprecates using the PC as the destination register, but if the PC is used, the instruction performs an exception return, that restores *PSTATE* from SPSR_<current_mode>. For encoding T3: is the general-purpose destination register, encoded in the "Rd" field.

<Rm> For encoding A1: is the general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.

For encoding T3: is the general-purpose source register, encoded in the "Rm" field.

Operation

The description of MOV, MOVS (register) gives the operational pseudocode for this instruction.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

RSB, RSBS (immediate)

Reverse Subtract (immediate) subtracts a register value from an immediate value, and writes the result to the destination register.

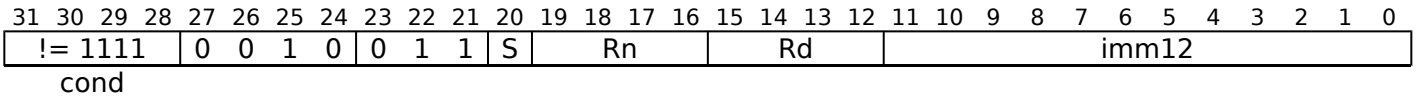
If the destination register is not the PC, the RSBS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. ARM deprecates any use of these encodings. However, when the destination register is the PC:

- The RSB variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*.
- The RSBS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores *PSTATE* from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state*.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2) .

A1



RSB (S == 0)

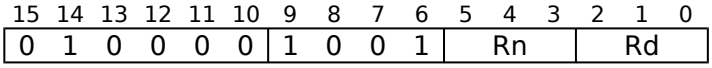
```
RSB{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

RSBS (S == 1)

```
RSBS{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

```
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = A32ExpandImm(imm12);
```

T1

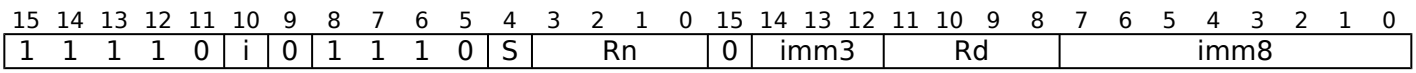


T1

```
RSB<c>{<q>} {<Rd>, }<Rn>, #0 // (Inside IT block)

RSBS{<q>} {<Rd>, }<Rn>, #0 // (Outside IT block)
```

```
d = UInt(Rd);  n = UInt(Rn);  setflags = !InITBlock();  imm32 = Zeros(32); // immediate = #0
```

T2



RSB (S == 0)

```
RSB<c>.W {<Rd>,} <Rn>, #0 // (Inside IT block)

RSB{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

RSBS (S == 1)

```
RSBS.W {<Rd>,} <Rn>, #0 // (Outside IT block)

RSBS{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

```
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = T32ExpandImm(i:imm3:imm8);
if d == 15 || n == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rd>	For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. Arm deprecates using the PC as the destination register, but if the PC is used: • For the RSB variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*. • For the RSBS variant, the instruction performs an exception return, that restores *PSTATE* from SPSR_<current_mode>. For encoding T1 and T2: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.
<Rn>	For encoding A1: is the general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated. For encoding T1 and T2: is the general-purpose source register, encoded in the "Rn" field.
<const>	For encoding A1: an immediate value. See *Modified immediate constants in A32 instructions* for the range of values. For encoding T2: an immediate value. See *Modified immediate constants in T32 instructions* for the range of values.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, nzcv) = AddWithCarry(NOT(R[n]), imm32, '1');
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.<N,Z,C,V> = nzcv;
```

Operational information

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:

- The values of the data supplied in any of its registers.
- The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

RSB, RSBS (register)

Reverse Subtract (register) subtracts a register value from an optionally-shifted register value, and writes the result to the destination register.

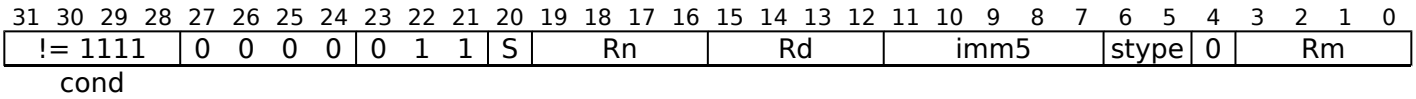
If the destination register is not the PC, the RSBS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. ARM deprecates any use of these encodings. However, when the destination register is the PC:

- The RSB variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*.
- The RSBS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores *PSTATE* from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state*.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



RSB, rotate right with extend (S == 0 && imm5 == 00000 && stype == 11)

```
RSB{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

RSB, shift or rotate by value (S == 0 && !(imm5 == 00000 && stype == 11))

```
RSB{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

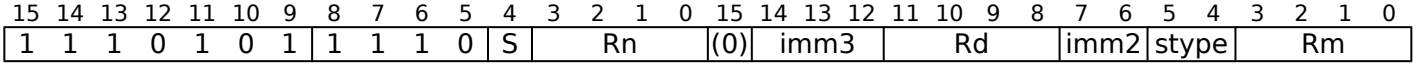
RSBS, rotate right with extend (S == 1 && imm5 == 00000 && stype == 11)

```
RSBS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

RSBS, shift or rotate by value (S == 1 && !(imm5 == 00000 && stype == 11))

```
RSBS{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm5);
```

T1



RSB, rotate right with extend (S == 0 && imm3 == 000 && imm2 == 00 && stype == 11)

```
RSB{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

RSB, shift or rotate by value (S == 0 && !(imm3 == 000 && imm2 == 00 && stype == 11))

```
RSB{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

RSBS, rotate right with extend (S == 1 && imm3 == 000 && imm2 == 00 && stype == 11)

```
RSBS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

RSBS, shift or rotate by value (S == 1 && !(imm3 == 000 && imm2 == 00 && stype == 11))

```
RSBS{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm3:imm2);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<Rd> For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the RSB variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*.
- For the RSBS variant, the instruction performs an exception return, that restores *PSTATE* from SPSR_<current_mode>.

For encoding T1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.

<Rn> For encoding A1: is the first general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.

For encoding T1: is the first general-purpose source register, encoded in the "Rn" field.

<Rm> For encoding A1: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.

For encoding T1: is the second general-purpose source register, encoded in the "Rm" field.

<shift> Is the type of shift to be applied to the second source register, encoded in “stype”:

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<amount> For encoding A1: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm5" field as <amount> modulo 32.

For encoding T1: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the "imm3:imm2" field as <amount> modulo 32.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (result, nzcv) = AddWithCarry(NOT(R[n]), shifted, '1');
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.<N,Z,C,V> = nzcv;
```

Operational information

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

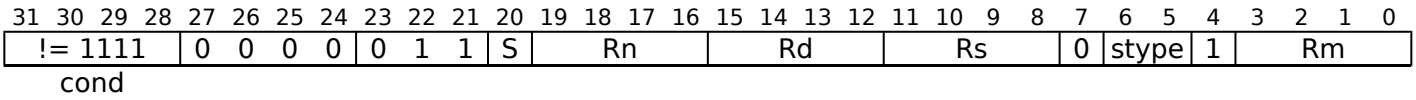
Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

RSB, RSBS (register-shifted register)

Reverse Subtract (register-shifted register) subtracts a register value from a register-shifted register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

A1



Flag setting (S == 1)

```
RSBS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, <shift> <Rs>
```

Not flag setting (S == 0)

```
RSB{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, <shift> <Rs>
```

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  s = UInt(Rs);
setflags = (S == '1');  shift_t = DecodeRegShift(stype);
if d == 15 || n == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<Rd> Is the general-purpose destination register, encoded in the "Rd" field.

<Rn> Is the first general-purpose source register, encoded in the "Rn" field.

<Rm> Is the second general-purpose source register, encoded in the "Rm" field.

<shift> Is the type of shift to be applied to the second source register, encoded in “stype”:

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<Rs> Is the third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[s]<7:0>);
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (result, nzcv) = AddWithCarry(NOT(R[n]), shifted, '1');
    R[d] = result;
    if setflags then
        PSTATE.<N,Z,C,V> = nzcv;
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

RSC, RSCS (immediate)

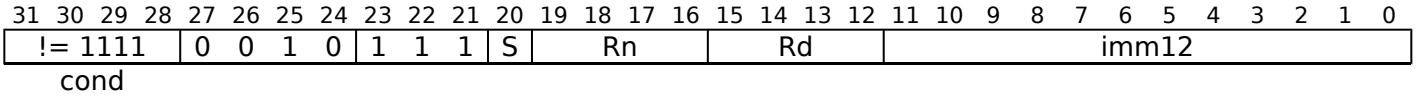
Reverse Subtract with Carry (immediate) subtracts a register value and the value of NOT (Carry flag) from an immediate value, and writes the result to the destination register.

If the destination register is not the PC, the RSCS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. ARM deprecates any use of these encodings. However, when the destination register is the PC:

- The RSC variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*.
- The RSCS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores *PSTATE* from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state*.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

A1



RSC (S == 0)

```
RSC{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

RSCS (S == 1)

```
RSCS{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

```
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = A32ExpandImm(imm12);
```

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. Arm deprecates using the PC as the destination register, but if the PC is used: • For the RSC variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*. • For the RSCS variant, the instruction performs an exception return, that restores *PSTATE* from SPSR_<current_mode>.
<Rn>	Is the general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.
<const>	An immediate value. See *Modified immediate constants in A32 instructions* for the range of values.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, nzcv) = AddWithCarry(NOT(R[n]), imm32, PSTATE.C);
    if d == 15 then
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.<N,Z,C,V> = nzcv;
```

Operational information

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

RSC, RSCS (register)

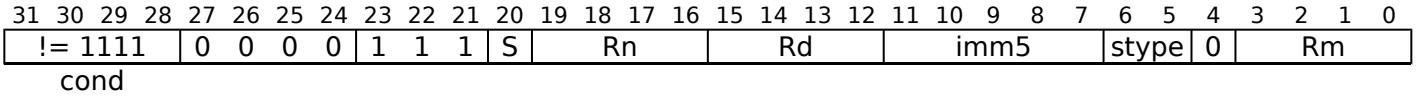
Reverse Subtract with Carry (register) subtracts a register value and the value of NOT (Carry flag) from an optionally-shifted register value, and writes the result to the destination register.

If the destination register is not the PC, the RSCS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. ARM deprecates any use of these encodings. However, when the destination register is the PC:

- The RSC variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*.
- The RSCS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores *PSTATE* from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state*.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

A1



RSC, rotate right with extend (S == 0 && imm5 == 00000 && stype == 11)

```
RSC{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

RSC, shift or rotate by value (S == 0 && !(imm5 == 00000 && stype == 11))

```
RSC{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

RSCS, rotate right with extend (S == 1 && imm5 == 00000 && stype == 11)

```
RSCS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

RSCS, shift or rotate by value (S == 1 && !(imm5 == 00000 && stype == 11))

```
RSCS{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm5);
```

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<Rd> Is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the RSC variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*.
- For the RSCS variant, the instruction performs an exception return, that restores *PSTATE* from SPSR_<current_mode>.

<Rn> Is the first general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.

<Rm> Is the second general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.

<shift> Is the type of shift to be applied to the second source register, encoded in “stype”:

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<amount> Is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm5" field as <amount> modulo 32.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (result, nzcv) = AddWithCarry(NOT(R[n]), shifted, PSTATE.C);
    if d == 15 then
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.<N,Z,C,V> = nzcv;
```

Operational information

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

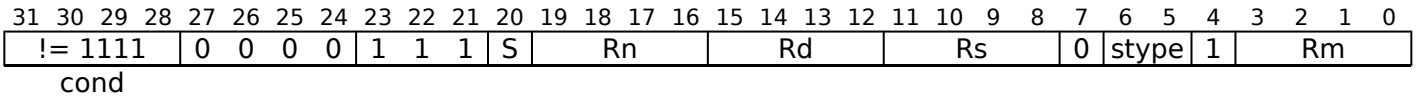
Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

RSC, RSCS (register-shifted register)

Reverse Subtract (register-shifted register) subtracts a register value and the value of NOT (Carry flag) from a register-shifted register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

A1



Flag setting (S == 1)

```
RSCS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, <shift> <Rs>
```

Not flag setting (S == 0)

```
RSC{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, <shift> <Rs>
```

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  s = UInt(Rs);
setflags = (S == '1');  shift_t = DecodeRegShift(stype);
if d == 15 || n == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<Rd> Is the general-purpose destination register, encoded in the "Rd" field.

<Rn> Is the first general-purpose source register, encoded in the "Rn" field.

<Rm> Is the second general-purpose source register, encoded in the "Rm" field.

<shift> Is the type of shift to be applied to the second source register, encoded in “stype”:

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<Rs> Is the third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[s]<7:0>);
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (result, nzcv) = AddWithCarry(NOT(R[n]), shifted, PSTATE.C);
    R[d] = result;
    if setflags then
        PSTATE.<N,Z,C,V> = nzcv;
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

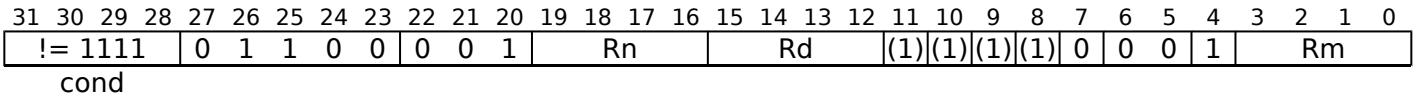
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SADD16

Signed Add 16 performs two 16-bit signed integer additions, and writes the results to the destination register. It sets *PSTATE*.GE according to the results of the additions.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

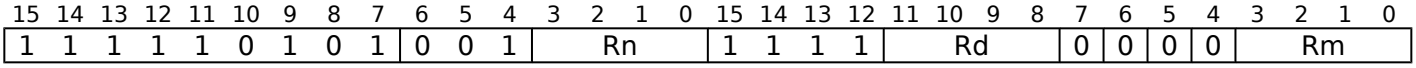
A1



A1

```
SADD16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1



T1

```
SADD16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = SInt(R[n]<15:0>) + SInt(R[m]<15:0>);
    sum2 = SInt(R[n]<31:16>) + SInt(R[m]<31:16>);
    R[d]<15:0>  = sum1<15:0>;
    R[d]<31:16> = sum2<15:0>;
    PSTATE.GE<1:0> = if sum1 >= 0 then '11' else '00';
    PSTATE.GE<3:2> = if sum2 >= 0 then '11' else '00';
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:

    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

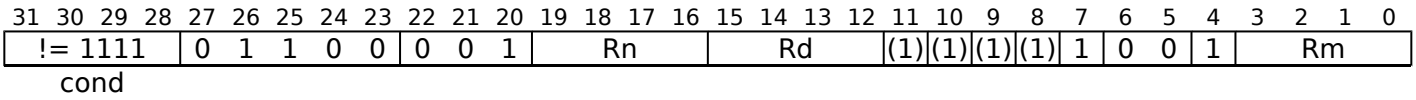
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SADD8

Signed Add 8 performs four 8-bit signed integer additions, and writes the results to the destination register. It sets *PSTATE*.GE according to the results of the additions.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

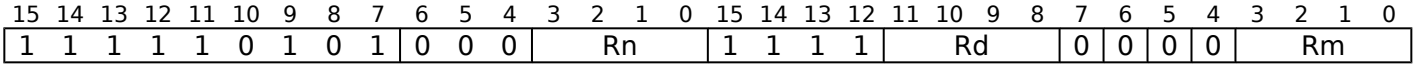
A1



A1

```
SADD8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1



T1

```
SADD8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = SInt(R[n]<7:0>) + SInt(R[m]<7:0>);
    sum2 = SInt(R[n]<15:8>) + SInt(R[m]<15:8>);
    sum3 = SInt(R[n]<23:16>) + SInt(R[m]<23:16>);
    sum4 = SInt(R[n]<31:24>) + SInt(R[m]<31:24>);
    R[d]<7:0>   = sum1<7:0>;
    R[d]<15:8>  = sum2<7:0>;
    R[d]<23:16> = sum3<7:0>;
    R[d]<31:24> = sum4<7:0>;
    PSTATE.GE<0>  = if sum1 >= 0 then '1' else '0';
    PSTATE.GE<1>  = if sum2 >= 0 then '1' else '0';
    PSTATE.GE<2>  = if sum3 >= 0 then '1' else '0';
    PSTATE.GE<3>  = if sum4 >= 0 then '1' else '0';
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

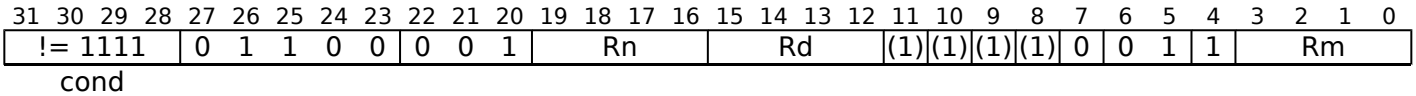
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SASX

Signed Add and Subtract with Exchange exchanges the two halfwords of the second operand, performs one 16-bit integer addition and one 16-bit subtraction, and writes the results to the destination register. It sets *PSTATE*.GE according to the results.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

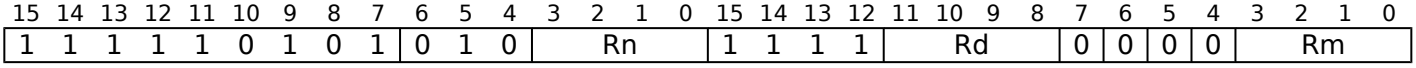
A1



A1

```
SASX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1



T1

```
SASX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff = SInt(R[n]<15:0>) - SInt(R[m]<31:16>);
    sum  = SInt(R[n]<31:16>) + SInt(R[m]<15:0>);
    R[d]<15:0>  = diff<15:0>;
    R[d]<31:16> = sum<15:0>;
    PSTATE.GE<1:0> = if diff >= 0 then '11' else '00';
    PSTATE.GE<3:2> = if sum  >= 0 then '11' else '00';
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SB

Speculation Barrier is a barrier that controls speculation.

The semantics of the Speculation Barrier are that the execution, until the barrier completes, of any instruction that appears later in the program order than the barrier:

- Cannot be performed speculatively to the extent that such speculation can be observed through side-channels as a result of control flow speculation or data value speculation.
- Can be speculatively executed as a result of predicting that a potentially exception generating instruction has not generated an exception.

In particular, any instruction that appears later in the program order than the barrier cannot cause a speculative allocation into any caching structure where the allocation of that entry could be indicative of any data value present in memory or in the registers.

The SB instruction:

- Cannot be speculatively executed as a result of control flow speculation or data value speculation.
- Can be speculatively executed as a result of predicting that a potentially exception generating instruction has not generated an exception. The potentially exception generating instruction can complete once it is known not to be speculative, and all data values generated by instructions appearing in program order before the SB instruction have their predicted values confirmed.

When the prediction of the instruction stream is not informed by data taken from the register outputs of the speculative execution of instructions appearing in program order after an uncompleted SB instruction, the SB instruction has no effect on the use of prediction resources to predict the instruction stream that is being fetched.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	1	0	1	1	1	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	0	1	1	1	(0)	(0)	(0)	(0)

A1

```
SB{<q>}

// No additional decoding required
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	1	(1)	(1)	(1)	(1)	1	0	(0)	0	(1)	(1)	(1)	(1)	0	1	1	1	(0)	(0)	(0)	(0)

T1

```
SB{<q>}

if InITBlock() then UNPREDICTABLE;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<q>	See *Standard assembler syntax fields*.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    SpeculationBarrier();
```

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SBC, SBCS (immediate)

Subtract with Carry (immediate) subtracts an immediate value and the value of NOT (Carry flag) from a register value, and writes the result to the destination register.

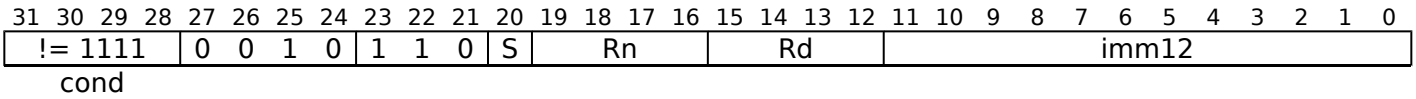
If the destination register is not the PC, the SBCS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. ARM deprecates any use of these encodings. However, when the destination register is the PC:

- The SBC variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*.
- The SBCS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores *PSTATE* from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state*.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



SBC (S == 0)

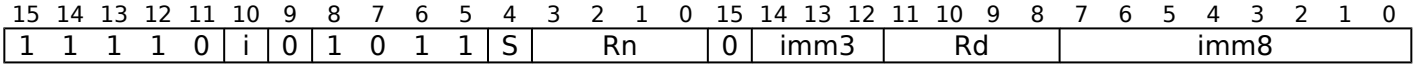
```
SBC{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

SBCS (S == 1)

```
SBCS{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

```
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = A32ExpandImm(imm12);
```

T1



SBC (S == 0)

```
SBC{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

SBCS (S == 1)

```
SBCS{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

```
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = T32ExpandImm(i:imm3:imm8);
if d == 15 || n == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<Rd> For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the SBC variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*.
- For the SBCS variant, the instruction performs an exception return, that restores *PSTATE* from SPSR_<current_mode>.

For encoding T1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.

<Rn> For encoding A1: is the general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.

For encoding T1: is the general-purpose source register, encoded in the "Rn" field.

<const> For encoding A1: an immediate value. See *Modified immediate constants in A32 instructions* for the range of values.

For encoding T1: an immediate value. See *Modified immediate constants in T32 instructions* for the range of values.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, nzcv) = AddWithCarry(R[n], NOT(imm32), PSTATE.C);
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.<N,Z,C,V> = nzcv;
```

Operational information

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SBC, SBCS (register)

Subtract with Carry (register) subtracts an optionally-shifted register value and the value of NOT (Carry flag) from a register value, and writes the result to the destination register.

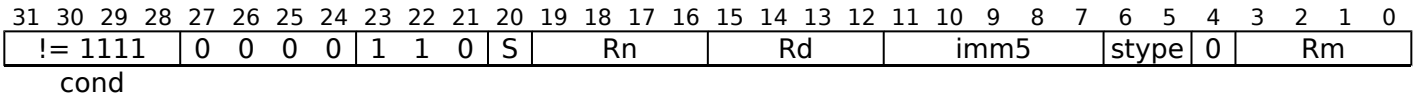
If the destination register is not the PC, the SBCS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. ARM deprecates any use of these encodings. However, when the destination register is the PC:

- The SBC variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*.
- The SBCS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores *PSTATE* from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state*.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2) .

A1



SBC, rotate right with extend (S == 0 && imm5 == 00000 && stype == 11)

```
SBC{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

SBC, shift or rotate by value (S == 0 && !(imm5 == 00000 && stype == 11))

```
SBC{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

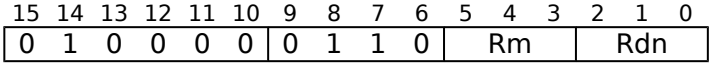
SBCS, rotate right with extend (S == 1 && imm5 == 00000 && stype == 11)

```
SBCS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

SBCS, shift or rotate by value (S == 1 && !(imm5 == 00000 && stype == 11))

```
SBCS{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm5);
```

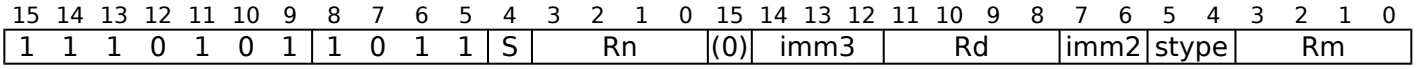
T1



T1

```
SBC<c>{<q>} {<Rdn>,} <Rdn>, <Rm> // (Inside IT block)

SBCS{<q>} {<Rdn>,} <Rdn>, <Rm> // (Outside IT block)

d = UInt(Rdn);  n = UInt(Rdn);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);
```

T2



SBC, rotate right with extend (S == 0 && imm3 == 000 && imm2 == 00 && stype == 11)

```
SBC{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

SBC, shift or rotate by value (S == 0 && !(imm3 == 000 && imm2 == 00 && stype == 11))

```
SBC<c>.W {<Rd>,} <Rn>, <Rm> // (Inside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1)

SBC{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

SBCS, rotate right with extend (S == 1 && imm3 == 000 && imm2 == 00 && stype == 11)

```
SBCS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

SBCS, shift or rotate by value (S == 1 && !(imm3 == 000 && imm2 == 00 && stype == 11))

```
SBCS.W {<Rd>,} <Rn>, <Rm> // (Outside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1)

SBCS{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm3:imm2);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<Rdn> Is the first general-purpose source register and the destination register, encoded in the "Rdn" field.

<Rd> For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the SBC variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*.
- For the SBCS variant, the instruction performs an exception return, that restores *PSTATE* from SPSR_<current_mode>.

For encoding T2: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.

<Rn> For encoding A1: is the first general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.

For encoding T2: is the first general-purpose source register, encoded in the "Rn" field.

<Rm> For encoding A1: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.

For encoding T1 and T2: is the second general-purpose source register, encoded in the "Rm" field.

<shift> Is the type of shift to be applied to the second source register, encoded in “stype”:

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<amount> For encoding A1: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm5" field as <amount> modulo 32.

For encoding T2: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the "imm3:imm2" field as <amount> modulo 32.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (result, nzcv) = AddWithCarry(R[n], NOT(shifted), PSTATE.C);
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.<N,Z,C,V> = nzcv;
```

Operational information

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

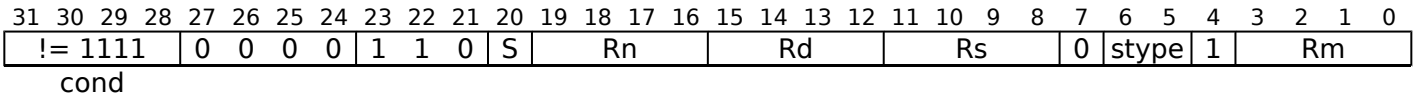
Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SBC, SBCS (register-shifted register)

Subtract with Carry (register-shifted register) subtracts a register-shifted register value and the value of NOT (Carry flag) from a register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

A1



Flag setting (S == 1)

```
SBCS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, <shift> <Rs>
```

Not flag setting (S == 0)

```
SBC{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, <shift> <Rs>
```

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  s = UInt(Rs);
setflags = (S == '1');  shift_t = DecodeRegShift(stype);
if d == 15 || n == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<Rd> Is the general-purpose destination register, encoded in the "Rd" field.

<Rn> Is the first general-purpose source register, encoded in the "Rn" field.

<Rm> Is the second general-purpose source register, encoded in the "Rm" field.

<shift> Is the type of shift to be applied to the second source register, encoded in “stype”:

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<Rs> Is the third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[s]<7:0>);
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (result, nzcv) = AddWithCarry(R[n], NOT(shifted), PSTATE.C);
    R[d] = result;
    if setflags then
        PSTATE.<N,Z,C,V> = nzcv;
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

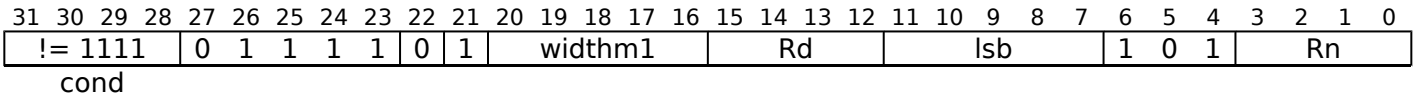
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SBFX

Signed Bit Field Extract extracts any number of adjacent bits at any position from a register, sign-extends them to 32 bits, and writes the result to the destination register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

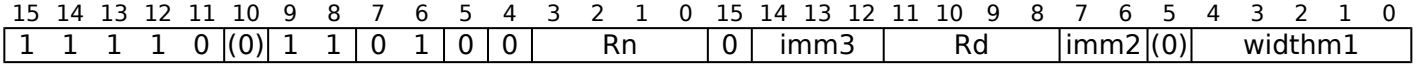
A1



A1

```
SBFX{<c>}{<q>} <Rd>, <Rn>, #<lsb>, #<width>

d = UInt(Rd);  n = UInt(Rn);
lsbit = UInt(lsb);  widthminus1 = UInt(widthm1);
if d == 15 || n == 15 then UNPREDICTABLE;
```

T1



T1

```
SBFX{<c>}{<q>} <Rd>, <Rn>, #<lsb>, #<width>

d = UInt(Rd);  n = UInt(Rn);
lsbit = UInt(imm3:imm2);  widthminus1 = UInt(widthm1);
if d == 15 || n == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the general-purpose source register, encoded in the "Rn" field.
<lsb>	For encoding A1: is the bit number of the least significant bit in the field, in the range 0 to 31, encoded in the "lsb" field. For encoding T1: is the bit number of the least significant bit in the field, in the range 0 to 31, encoded in the "imm3:imm2" field.
<width>	Is the width of the field, in the range 1 to 32-<lsb>, encoded in the "widthm1" field as <width>-1.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    msbit = lsbit + widthminus1;
    if msbit <= 31 then
        R[d] = SignExtend(R[n]<msbit:lsbit>, 32);
    else
        UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `msbit > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The value in the destination register is UNKNOWN.

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

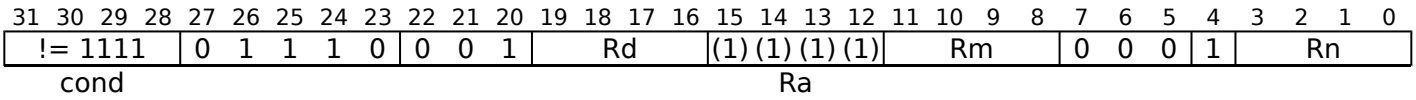
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SDIV

Signed Divide divides a 32-bit signed integer register value by a 32-bit signed integer register value, and writes the result to the destination register. The condition flags are not affected.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



A1

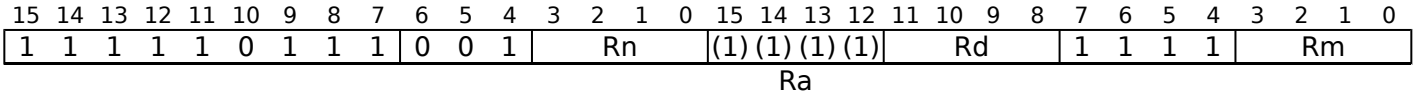
```
SDIV{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);
if d == 15 || n == 15 || m == 15 || a != 15 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `Ra != '1111'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes as described, with no change to its behavior and no additional side effects.
- The instruction executes as described, and the register specified by Ra becomes UNKNOWN.

T1



T1

```
SDIV{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);
if d == 15 || n == 15 || m == 15 || a != 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

CONSTRAINED UNPREDICTABLE behavior

If `Ra != '1111'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes as described, with no change to its behavior and no additional side effects.
- The instruction executes as described, and the register specified by Ra becomes UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register holding the dividend, encoded in the "Rn" field.

<Rm> Is the second general-purpose source register holding the divisor, encoded in the "Rm" field.

Overflow

If the signed integer division 0x80000000 / 0xFFFFFFFF is performed, the pseudocode produces the intermediate integer result $+2^{31}$, that overflows the 32-bit signed integer range. No indication of this overflow case is produced, and the 32-bit result written to <Rd> must be the bottom 32 bits of the binary representation of $+2^{31}$. So the result of the division is 0x80000000.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if SInt(R[m]) == 0 then
        result = 0;
    else
        result = RoundTowardsZero(Real(SInt(R[n])) / Real(SInt(R[m])));
    R[d] = result<31:0>;
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

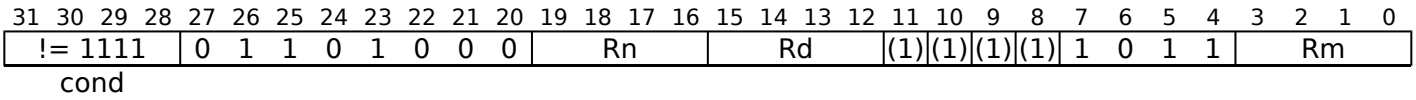
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SEL

Select Bytes selects each byte of its result from either its first operand or its second operand, according to the values of the *PSTATE*.GE flags.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

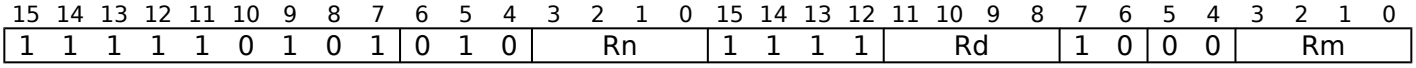


A1

```
SEL{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1



T1

```
SEL{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    R[d]<7:0>   = if PSTATE.GE<0> == '1' then R[n]<7:0>   else R[m]<7:0>;
    R[d]<15:8>  = if PSTATE.GE<1> == '1' then R[n]<15:8>  else R[m]<15:8>;
    R[d]<23:16> = if PSTATE.GE<2> == '1' then R[n]<23:16> else R[m]<23:16>;
    R[d]<31:24> = if PSTATE.GE<3> == '1' then R[n]<31:24> else R[m]<31:24>;
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SETEND

Set Endianness writes a new value to *PSTATE*.E.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	0	1	0	0	0	0	(0)	(0)	(0)	1	(0)	(0)	(0)	(0)	(0)	(0)	E	(0)	0	0	0	0	(0)	(0)	(0)	(0)

A1

```
SETEND{<q>} <endian_specifier> // (Cannot be conditional)

set_bigend = (E == '1');
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	1	1	0	0	1	0	(1)	E	(0)	(0)	(0)

T1

```
SETEND{<q>} <endian_specifier> // (Not permitted in IT block)

set_bigend = (E == '1');
if InITBlock() then UNPREDICTABLE;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<q> See *Standard assembler syntax fields*.

<endian_specifier> Is the endianness to be selected, and the value to be set in PSTATE.E, encoded in “E”:

E	<endian_specifier>
0	LE
1	BE

Operation

```
EncodingSpecificOperations();
AArch32.CheckSETENDEnabled();
PSTATE.E = if set_bigend then '1' else '0';
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SETPAN

Set Privileged Access Never writes a new value to *PSTATE*.PAN.
This instruction is available only in privileged mode and it is a NOP when executed in User mode.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1
(FEAT_PAN)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	0	1	0	0	0	1	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	imm1	(0)	0	0	0	0	(0)	(0)	(0)	(0)

A1

```
SETPAN{<q>} #<imm> // (Cannot be conditional)

if !HavePANExt() then UNDEFINED;
value = imm1;
```

T1
(FEAT_PAN)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	1	1	0	0	0	0	(1)	imm1	(0)	(0)	(0)

T1

```
SETPAN{<q>} #<imm> // (Not permitted in IT block)

if InITBlock() then UNPREDICTABLE;
if !HavePANExt() then UNDEFINED;
value = imm1;
```

Assembler Symbols

<q>	See *Standard assembler syntax fields*.
<imm>	Is the unsigned immediate 0 or 1, encoded in the "imm1" field.

Operation

```
EncodingSpecificOperations();
if PSTATE.EL != EL0 then
    PSTATE.PAN = value;
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

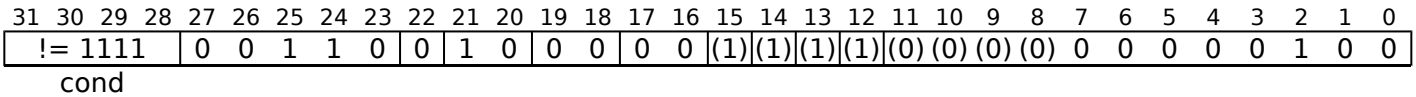
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SEV

Send Event is a hint instruction. It causes an event to be signaled to all PEs in the multiprocessor system. For more information, see *Wait For Event and Send Event*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2) .

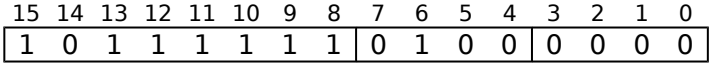
A1



A1

```
SEV{<c>}{<q>}

// No additional decoding required
```

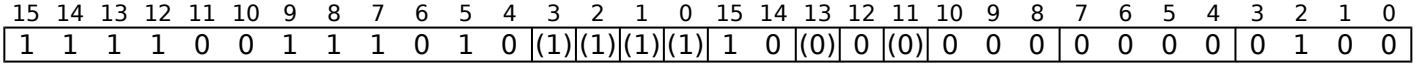
T1



T1

```
SEV{<c>}{<q>}

// No additional decoding required
```

T2



T2

```
SEV{<c>}.W

// No additional decoding required
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    SendEvent();
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

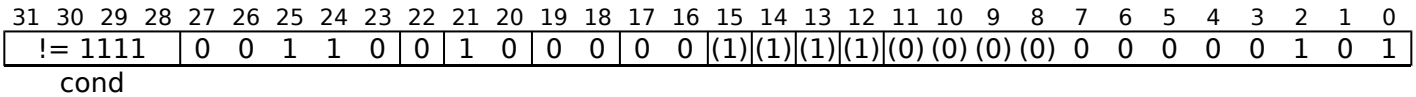
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SEVL

Send Event Local is a hint instruction that causes an event to be signaled locally without requiring the event to be signaled to other PEs in the multiprocessor system. It can prime a wait-loop which starts with a WFE instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2) .

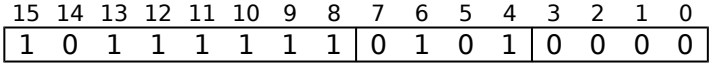
A1



A1

```
SEVL{<c>}{<q>}

// No additional decoding required
```

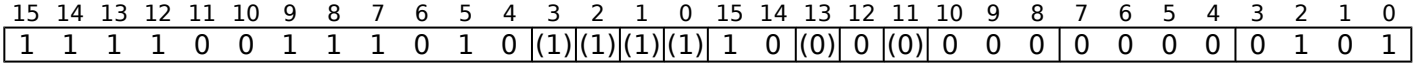
T1



T1

```
SEVL{<c>}{<q>}

// No additional decoding required
```

T2



T2

```
SEVL{<c>}.W

// No additional decoding required
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    SendEventLocal();
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

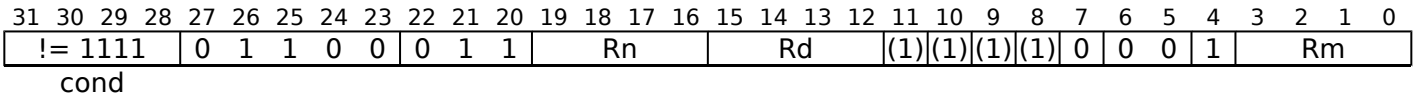
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SHADD16

Signed Halving Add 16 performs two signed 16-bit integer additions, halves the results, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

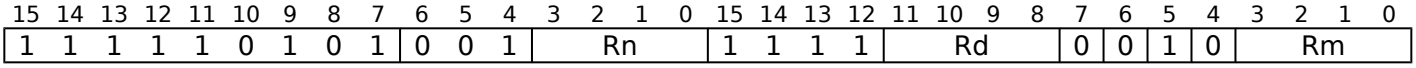
A1



A1

```
SHADD16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1



T1

```
SHADD16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = SInt(R[n]<15:0>) + SInt(R[m]<15:0>);
    sum2 = SInt(R[n]<31:16>) + SInt(R[m]<31:16>);
    R[d]<15:0>  = sum1<16:1>;
    R[d]<31:16> = sum2<16:1>;
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

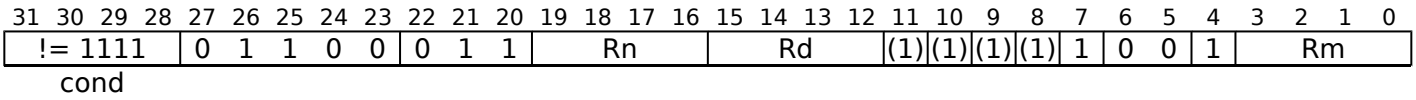
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SHADD8

Signed Halving Add 8 performs four signed 8-bit integer additions, halves the results, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

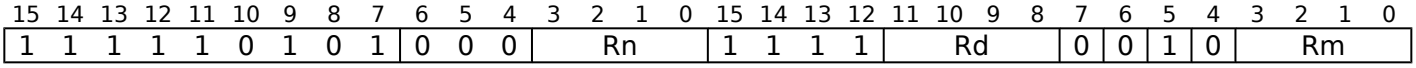
A1



A1

```
SHADD8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1



T1

```
SHADD8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = SInt(R[n]<7:0>) + SInt(R[m]<7:0>);
    sum2 = SInt(R[n]<15:8>) + SInt(R[m]<15:8>);
    sum3 = SInt(R[n]<23:16>) + SInt(R[m]<23:16>);
    sum4 = SInt(R[n]<31:24>) + SInt(R[m]<31:24>);
    R[d]<7:0>   = sum1<8:1>;
    R[d]<15:8>  = sum2<8:1>;
    R[d]<23:16> = sum3<8:1>;
    R[d]<31:24> = sum4<8:1>;
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

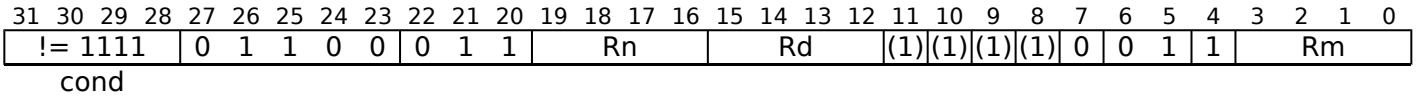
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SHASX

Signed Halving Add and Subtract with Exchange exchanges the two halfwords of the second operand, performs one signed 16-bit integer addition and one signed 16-bit subtraction, halves the results, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

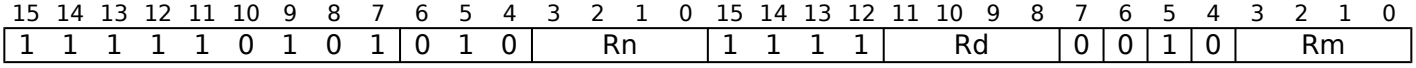
A1



A1

```
SHASX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1



T1

```
SHASX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff = SInt(R[n]<15:0>) - SInt(R[m]<31:16>);
    sum  = SInt(R[n]<31:16>) + SInt(R[m]<15:0>);
    R[d]<15:0>  = diff<16:1>;
    R[d]<31:16> = sum<16:1>;
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.

    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

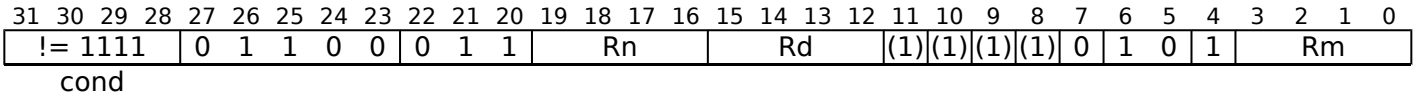
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SHSAX

Signed Halving Subtract and Add with Exchange exchanges the two halfwords of the second operand, performs one signed 16-bit integer subtraction and one signed 16-bit addition, halves the results, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

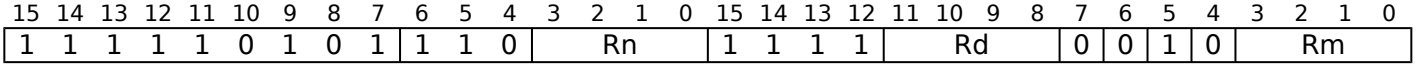
A1



A1

```
SHSAX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1



T1

```
SHSAX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum  = SInt(R[n]<15:0>) + SInt(R[m]<31:16>);
    diff = SInt(R[n]<31:16>) - SInt(R[m]<15:0>);
    R[d]<15:0>  = sum<16:1>;
    R[d]<31:16> = diff<16:1>;
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.

    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

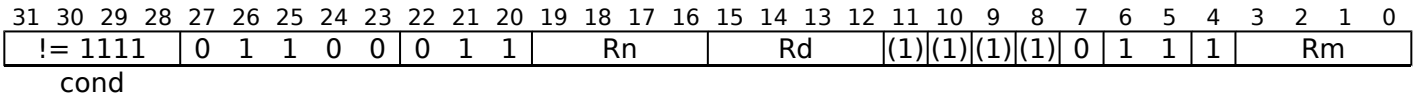
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SHSUB16

Signed Halving Subtract 16 performs two signed 16-bit integer subtractions, halves the results, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

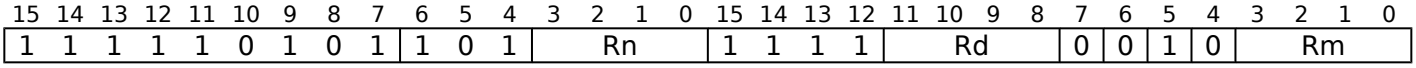
A1



A1

```
SHSUB16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1



T1

```
SHSUB16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = SInt(R[n]<15:0>) - SInt(R[m]<15:0>);
    diff2 = SInt(R[n]<31:16>) - SInt(R[m]<31:16>);
    R[d]<15:0>  = diff1<16:1>;
    R[d]<31:16> = diff2<16:1>;
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

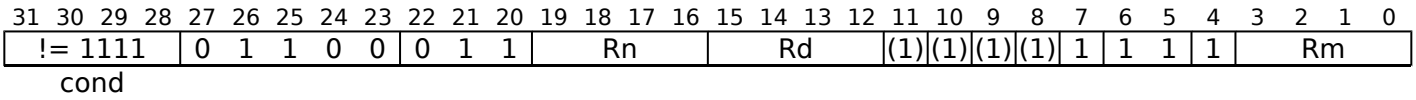
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SHSUB8

Signed Halving Subtract 8 performs four signed 8-bit integer subtractions, halves the results, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

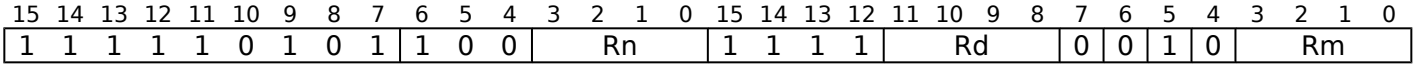
A1



A1

```
SHSUB8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1



T1

```
SHSUB8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = SInt(R[n]<7:0>) - SInt(R[m]<7:0>);
    diff2 = SInt(R[n]<15:8>) - SInt(R[m]<15:8>);
    diff3 = SInt(R[n]<23:16>) - SInt(R[m]<23:16>);
    diff4 = SInt(R[n]<31:24>) - SInt(R[m]<31:24>);
    R[d]<7:0>   = diff1<8:1>;
    R[d]<15:8>  = diff2<8:1>;
    R[d]<23:16> = diff3<8:1>;
    R[d]<31:24> = diff4<8:1>;
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SMC

Secure Monitor Call causes a Secure Monitor Call exception. For more information see *Secure Monitor Call (SMC) exception*.

SMC is available only for software executing at EL1 or higher. It is UNDEFINED in User mode.

If the values of *HCR*.TSC and *SCR*.SCD are both 0, execution of an SMC instruction at EL1 or higher generates a Secure Monitor Call exception that is taken to EL3. When EL3 is using AArch32 this exception is taken to Monitor mode. When EL3 is using AArch64, it is the *SCR_EL3*.SMD bit, rather than the *SCR*.SCD bit, that can change the effect of executing an SMC instruction.

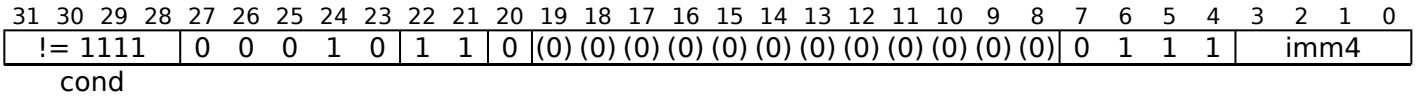
If the value of *HCR*.TSC is 1, execution of an SMC instruction in a Non-secure EL1 mode generates an exception that is taken to EL2, regardless of the value of *SCR*.SCD. When EL2 is using AArch32, this is a Hyp Trap exception that is taken to Hyp mode. For more information see *Traps to Hyp mode of Non-secure EL1 execution of SMC instructions*.

If the value of *HCR*.TSC is 0 and the value of *SCR*.SCD is 1, the SMC instruction is:

- UNDEFINED in Non-secure state.
- CONSTRAINED UNPREDICTABLE if executed in Secure state at EL1 or higher.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

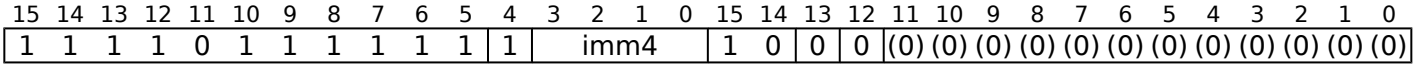
A1



A1

```
SMC{<c>}{<q>} {#}<imm4>

// imm4 is for assembly/disassembly only and is ignored by hardware
```

T1



T1

```
SMC{<c>}{<q>} {#}<imm4>

// imm4 is for assembly/disassembly only and is ignored by hardware
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<imm4>	Is a 4-bit unsigned immediate value, in the range 0 to 15, encoded in the "imm4" field. This is ignored by the PE. The Secure Monitor Call exception handler (Secure Monitor code) can use this value to determine what service is being requested, but Arm does not recommend this.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();

    AArch32.CheckForSMCUndefOrTrap();

    if !ELUsingAArch32(EL3) then
        if SCR_EL3.SMD == '1' then
            // SMC disabled.
            UNDEFINED;
    else
        if SCR.SCD == '1' then
            // SMC disabled
            if IsSecure() then
                // Executes either as a NOP or UNALLOCATED.
                c = ConstrainUnpredictable(Unpredictable_SMD);
                assert c IN {Constraint_NOP, Constraint_UNDEF};
                if c == Constraint_NOP then EndOfInstruction();
            UNDEFINED;

    if !ELUsingAArch32(EL3) then
        AArch64.CallSecureMonitor(Zeros(16));
    else
        AArch32.TakeSMCException();
```

CONSTRAINED UNPREDICTABLE behavior

If `SCR.SCD == '1' && IsSecure()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

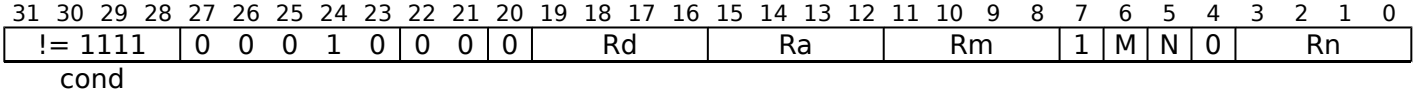
SMLABB, SMLABT, SMLATB, SMLATT

Signed Multiply Accumulate (halfwords) performs a signed multiply accumulate operation. The multiply acts on two signed 16-bit quantities, taken from either the bottom or the top half of their respective source registers. The other halves of these source registers are ignored. The 32-bit product is added to a 32-bit accumulate value and the result is written to the destination register.

If overflow occurs during the addition of the accumulate value, the instruction sets *PSTATE*.Q to 1. It is not possible for overflow to occur during the multiplication.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



SMLABB (M == 0 && N == 0)

```
SMLABB{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

SMLABT (M == 1 && N == 0)

```
SMLABT{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

SMLATB (M == 0 && N == 1)

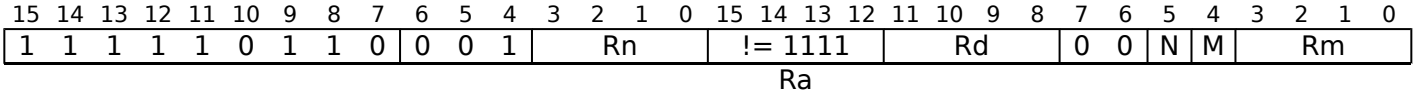
```
SMLATB{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

SMLATT (M == 1 && N == 1)

```
SMLATT{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);
n_high = (N == '1');  m_high = (M == '1');
if d == 15 || n == 15 || m == 15 || a == 15 then UNPREDICTABLE;
```

T1



SMLABB (N == 0 && M == 0)

```
SMLABB{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

SMLABT (N == 0 && M == 1)

```
SMLABT{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

SMLATB (N == 1 && M == 0)

```
SMLATB{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

SMLATT (N == 1 && M == 1)

```
SMLATT{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

```
if Ra == '1111' then SEE "SMULBB, SMULBT, SMULTB, SMULTT";
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);
n_high = (N == '1');  m_high = (M == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register holding the multiplicand in the bottom or top half (selected by <x>), encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register holding the multiplier in the bottom or top half (selected by <y>), encoded in the "Rm" field.
<Ra>	Is the third general-purpose source register holding the addend, encoded in the "Ra" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand1 = if n_high then R[n]<31:16> else R[n]<15:0>;
    operand2 = if m_high then R[m]<31:16> else R[m]<15:0>;
    result = SInt(operand1) * SInt(operand2) + SInt(R[a]);
    R[d] = result<31:0>;
    if result != SInt(result<31:0>) then  // Signed overflow
        PSTATE.Q = '1';
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

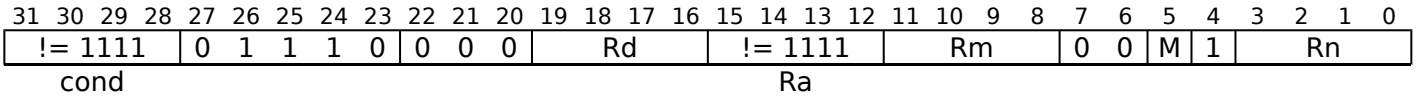
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SMLAD, SMLADX

Signed Multiply Accumulate Dual performs two signed 16 x 16-bit multiplications. It adds the products to a 32-bit accumulate operand.
Optionally, the instruction can exchange the halfwords of the second operand before performing the arithmetic. This produces top x bottom and bottom x top multiplication.
This instruction sets *PSTATE*.Q to 1 if the accumulate operation overflows. Overflow cannot occur during the multiplications.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



SMLAD (M == 0)

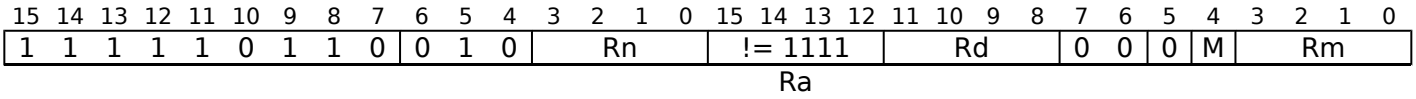
```
SMLAD{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

SMLADX (M == 1)

```
SMLADX{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

```
if Ra == '1111' then SEE "SMUAD";
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);
m_swap = (M == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1



SMLAD (M == 0)

```
SMLAD{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

SMLADX (M == 1)

```
SMLADX{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

```
if Ra == '1111' then SEE "SMUAD";
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);
m_swap = (M == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.

<Rm> Is the second general-purpose source register, encoded in the "Rm" field.

<Ra> Is the third general-purpose source register holding the addend, encoded in the "Ra" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand2 = if m_swap then ROR(R[m],16) else R[m];
    product1 = SInt(R[n]<15:0>) * SInt(operand2<15:0>);
    product2 = SInt(R[n]<31:16>) * SInt(operand2<31:16>);
    result = product1 + product2 + SInt(R[a]);
    R[d] = result<31:0>;
    if result != SInt(result<31:0>) then  // Signed overflow
        PSTATE.Q = '1';
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

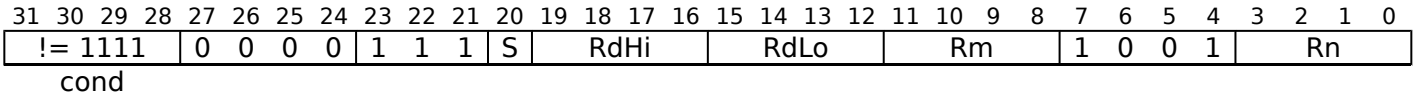
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SMLAL, SMLALS

Signed Multiply Accumulate Long multiplies two signed 32-bit values to produce a 64-bit value, and accumulates this with a 64-bit value.
In A32 instructions, the condition flags can optionally be updated based on the result. Use of this option adversely affects performance on many implementations.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



Flag setting (S == 1)

```
SMLALS{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

Not flag setting (S == 0)

```
SMLAL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

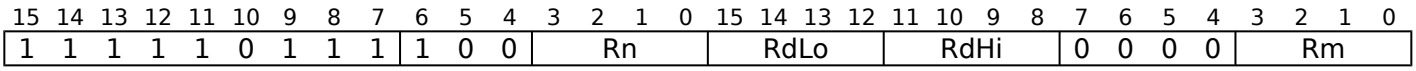
```
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
if dHi == dLo then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `dHi == dLo`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

T1



T1

```
SMLAL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

```
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);  setflags = FALSE;
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
// Armv8-A removes UNPREDICTABLE for R13
if dHi == dLo then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `dHi == dLo`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<RdLo>	Is the general-purpose source register holding the lower 32 bits of the addend, and the destination register for the lower 32 bits of the result, encoded in the "RdLo" field.
<RdHi>	Is the general-purpose source register holding the upper 32 bits of the addend, and the destination register for the upper 32 bits of the result, encoded in the "RdHi" field.
<Rn>	Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = SInt(R[n]) * SInt(R[m]) + SInt(R[dHi]:R[dLo]);
    R[dHi] = result<63:32>;
    R[dLo] = result<31:0>;
    if setflags then
        PSTATE.N = result<63>;
        PSTATE.Z = IsZeroBit(result<63:0>);
        // PSTATE.C, PSTATE.V unchanged
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

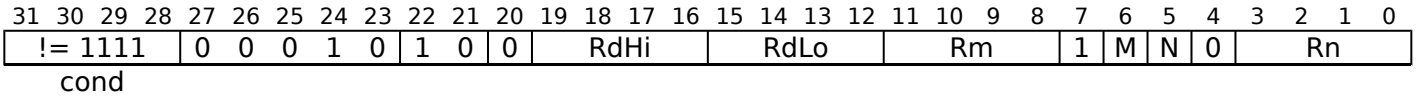
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SMLALBB, SMLALBT, SMLALTB, SMLALTT

Signed Multiply Accumulate Long (halfwords) multiplies two signed 16-bit values to produce a 32-bit value, and accumulates this with a 64-bit value. The multiply acts on two signed 16-bit quantities, taken from either the bottom or the top half of their respective source registers. The other halves of these source registers are ignored. The 32-bit product is sign-extended and accumulated with a 64-bit accumulate value.
Overflow is possible during this instruction, but only as a result of the 64-bit addition. This overflow is not detected if it occurs. Instead, the result wraps around modulo 2^{64}.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



SMLALBB (M == 0 && N == 0)

```
SMLALBB{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

SMLALBT (M == 1 && N == 0)

```
SMLALBT{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

SMLALTB (M == 0 && N == 1)

```
SMLALTB{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

SMLALTT (M == 1 && N == 1)

```
SMLALTT{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

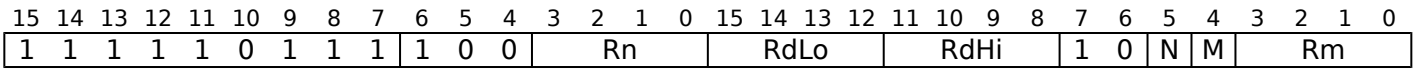
```
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);
n_high = (N == '1');  m_high = (M == '1');
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
if dHi == dLo then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `dHi == dLo`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The value in the destination register is UNKNOWN.

T1



SMLALBB (N == 0 && M == 0)

```
SMLALBB{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

SMLALBT (N == 0 && M == 1)

```
SMLALBT{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

SMLALTB (N == 1 && M == 0)

```
SMLALTB{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

SMLALTT (N == 1 && M == 1)

```
SMLALTT{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

```
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);
n_high = (N == '1');  m_high = (M == '1');
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
// Armv8-A removes UNPREDICTABLE for R13
if dHi == dLo then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `dHi == dLo`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The value in the destination register is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<RdLo>	Is the general-purpose source register holding the lower 32 bits of the addend, and the destination register for the lower 32 bits of the result, encoded in the "RdLo" field.
<RdHi>	Is the general-purpose source register holding the upper 32 bits of the addend, and the destination register for the upper 32 bits of the result, encoded in the "RdHi" field.
<Rn>	For encoding A1: is the first general-purpose source register holding the multiplicand in the bottom or top half (selected by <x>), encoded in the "Rn" field. For encoding T1: is the first general-purpose source register holding the multiplicand in the bottom or top half (selected by <x>), encoded in the "Rn" field.
<Rm>	For encoding A1: is the second general-purpose source register holding the multiplier in the bottom or top half (selected by <y>), encoded in the "Rm" field. For encoding T1: is the second general-purpose source register holding the multiplier in the bottom or top half (selected by <x>), encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand1 = if n_high then R[n]<31:16> else R[n]<15:0>;
    operand2 = if m_high then R[m]<31:16> else R[m]<15:0>;
    result = SInt(operand1) * SInt(operand2) + SInt(R[dHi]:R[dLo]);
    R[dHi] = result<63:32>;
    R[dLo] = result<31:0>;
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

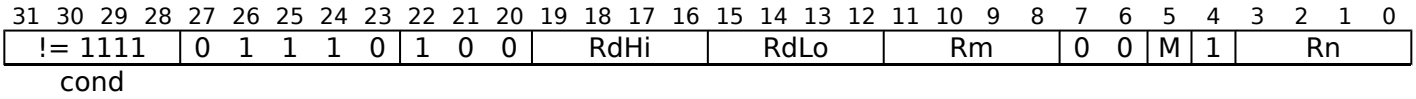
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SMLALD, SMLALDX

Signed Multiply Accumulate Long Dual performs two signed 16 x 16-bit multiplications. It adds the products to a 64-bit accumulate operand.
Optionally, the instruction can exchange the halfwords of the second operand before performing the arithmetic. This produces top x bottom and bottom x top multiplication.
Overflow is possible during this instruction, but only as a result of the 64-bit addition. This overflow is not detected if it occurs. Instead, the result wraps around modulo 2^{64}.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



SMLALD (M == 0)

```
SMLALD{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

SMLALDX (M == 1)

```
SMLALDX{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

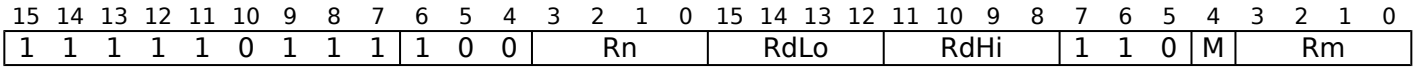
```
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);  m_swap = (M == '1');
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
if dHi == dLo then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `dHi == dLo`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The value in the destination register is UNKNOWN.

T1



SMLALD (M == 0)

```
SMLALD{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

SMLALDX (M == 1)

```
SMLALDX{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

```
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);  m_swap = (M == '1');
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
// Armv8-A removes UNPREDICTABLE for R13
if dHi == dLo then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `dHi == dLo`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.

- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<RdLo>	Is the general-purpose source register holding the lower 32 bits of the addend, and the destination register for the lower 32 bits of the result, encoded in the "RdLo" field.
<RdHi>	Is the general-purpose source register holding the upper 32 bits of the addend, and the destination register for the upper 32 bits of the result, encoded in the "RdHi" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand2 = if m_swap then ROR(R[m],16) else R[m];
    product1 = SInt(R[n]<15:0>) * SInt(operand2<15:0>);
    product2 = SInt(R[n]<31:16>) * SInt(operand2<31:16>);
    result = product1 + product2 + SInt(R[dHi]:R[dLo]);
    R[dHi] = result<63:32>;
    R[dLo] = result<31:0>;
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

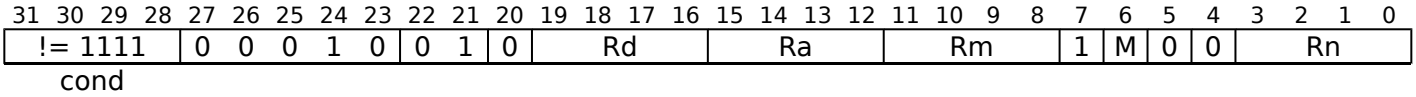
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SMLAWB, SMLAWT

Signed Multiply Accumulate (word by halfword) performs a signed multiply accumulate operation. The multiply acts on a signed 32-bit quantity and a signed 16-bit quantity. The signed 16-bit quantity is taken from either the bottom or the top half of its source register. The other half of the second source register is ignored. The top 32 bits of the 48-bit product are added to a 32-bit accumulate value and the result is written to the destination register. The bottom 16 bits of the 48-bit product are ignored.
If overflow occurs during the addition of the accumulate value, the instruction sets *PSTATE*.Q to 1. No overflow can occur during the multiplication.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



SMLAWB (M == 0)

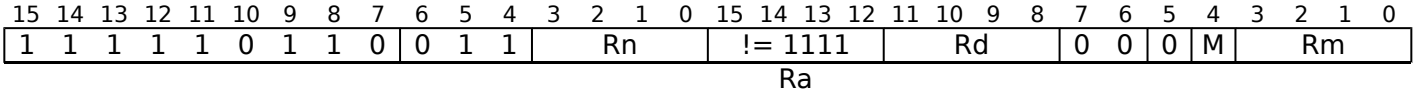
```
SMLAWB{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

SMLAWT (M == 1)

```
SMLAWT{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);  m_high = (M == '1');
if d == 15 || n == 15 || m == 15 || a == 15 then UNPREDICTABLE;
```

T1



SMLAWB (M == 0)

```
SMLAWB{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

SMLAWT (M == 1)

```
SMLAWT{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

```
if Ra == '1111' then SEE "SMULWB, SMULWT";
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);  m_high = (M == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register holding the multiplier in the bottom or top half (selected by <y>), encoded in the "Rm" field.

<Ra> Is the third general-purpose source register holding the addend, encoded in the "Ra" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand2 = if m_high then R[m]<31:16> else R[m]<15:0>;
    result = SInt(R[n]) * SInt(operand2) + (SInt(R[a]) << 16);
    R[d] = result<47:16>;
    if (result >> 16) != SInt(R[d]) then  // Signed overflow
        PSTATE.Q = '1';
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

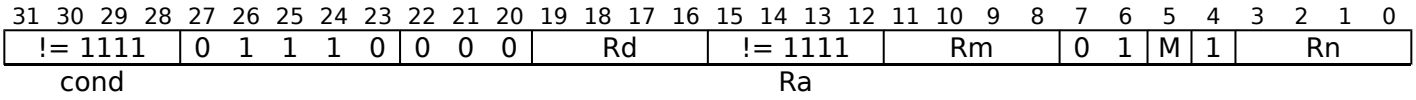
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SMLSD, SMLSDX

Signed Multiply Subtract Dual performs two signed 16 x 16-bit multiplications. It adds the difference of the products to a 32-bit accumulate operand.
Optionally, the instruction can exchange the halfwords of the second operand before performing the arithmetic. This produces top x bottom and bottom x top multiplication.
This instruction sets *PSTATE*.Q to 1 if the accumulate operation overflows. Overflow cannot occur during the multiplications or subtraction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



SMLSD (M == 0)

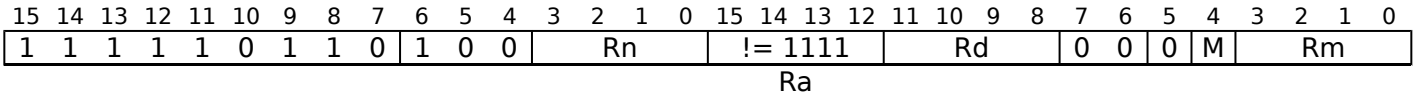
```
SMLSD{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

SMLSDX (M == 1)

```
SMLSDX{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

```
if Ra == '1111' then SEE "SMUSD";
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);  m_swap = (M == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1



SMLSD (M == 0)

```
SMLSD{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

SMLSDX (M == 1)

```
SMLSDX{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

```
if Ra == '1111' then SEE "SMUSD";
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);  m_swap = (M == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.

<Ra> Is the third general-purpose source register holding the addend, encoded in the "Ra" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand2 = if m_swap then ROR(R[m],16) else R[m];
    product1 = SInt(R[n]<15:0>) * SInt(operand2<15:0>);
    product2 = SInt(R[n]<31:16>) * SInt(operand2<31:16>);
    result = product1 - product2 + SInt(R[a]);
    R[d] = result<31:0>;
    if result != SInt(result<31:0>) then  // Signed overflow
        PSTATE.Q = '1';
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

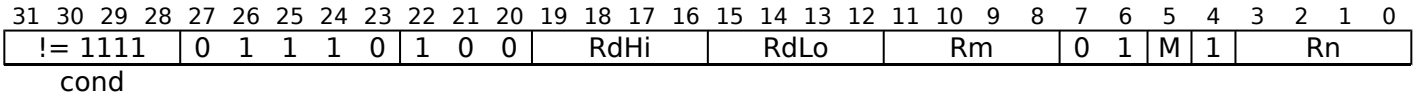
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SMLSLD, SMLSLDX

Signed Multiply Subtract Long Dual performs two signed 16 x 16-bit multiplications. It adds the difference of the products to a 64-bit accumulate operand.
Optionally, the instruction can exchange the halfwords of the second operand before performing the arithmetic. This produces top x bottom and bottom x top multiplication.
Overflow is possible during this instruction, but only as a result of the 64-bit addition. This overflow is not detected if it occurs. Instead, the result wraps around modulo 2^{64}.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



SMLSLD (M == 0)

```
SMLSLD{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

SMLSLDX (M == 1)

```
SMLSLDX{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

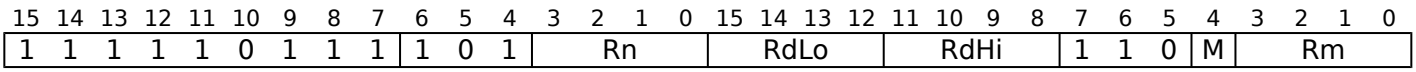
```
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);  m_swap = (M == '1');
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
if dHi == dLo then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `dHi == dLo`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The value in the destination register is UNKNOWN.

T1



SMLSLD (M == 0)

```
SMLSLD{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

SMLSLDX (M == 1)

```
SMLSLDX{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

```
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);  m_swap = (M == '1');
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
// Armv8-A removes UPREDICTABLE for R13
if dHi == dLo then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `dHi == dLo`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.

- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<RdLo>	Is the general-purpose source register holding the lower 32 bits of the addend, and the destination register for the lower 32 bits of the result, encoded in the "RdLo" field.
<RdHi>	Is the general-purpose source register holding the upper 32 bits of the addend, and the destination register for the upper 32 bits of the result, encoded in the "RdHi" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand2 = if m_swap then ROR(R[m],16) else R[m];
    product1 = SInt(R[n]<15:0>) * SInt(operand2<15:0>);
    product2 = SInt(R[n]<31:16>) * SInt(operand2<31:16>);
    result = product1 - product2 + SInt(R[dHi]:R[dLo]);
    R[dHi] = result<63:32>;
    R[dLo] = result<31:0>;
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

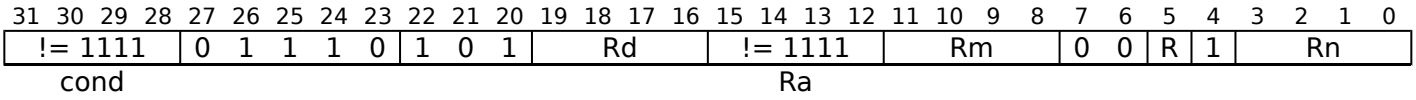
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SMMLA, SMMLAR

Signed Most Significant Word Multiply Accumulate multiplies two signed 32-bit values, extracts the most significant 32 bits of the result, and adds an accumulate value.
Optionally, the instruction can specify that the result is rounded instead of being truncated. In this case, the constant 0x80000000 is added to the product before the high word is extracted.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



SMMLA (R == 0)

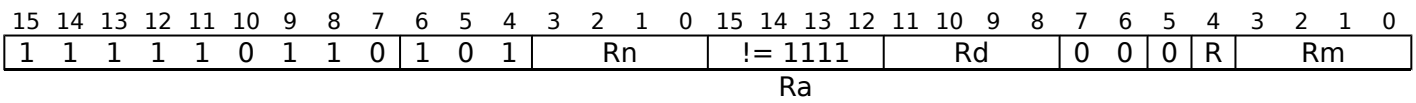
```
SMMLA{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

SMMLAR (R == 1)

```
SMMLAR{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

```
if Ra == '1111' then SEE "SMMUL";
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);  round = (R == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1



SMMLA (R == 0)

```
SMMLA{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

SMMLAR (R == 1)

```
SMMLAR{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

```
if Ra == '1111' then SEE "SMMUL";
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);  round = (R == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
<Ra>	Is the third general-purpose source register holding the addend, encoded in the "Ra" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = (SInt(R[a]) << 32) + SInt(R[n]) * SInt(R[m]);
    if round then result = result + 0x80000000;
    R[d] = result<63:32>;
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

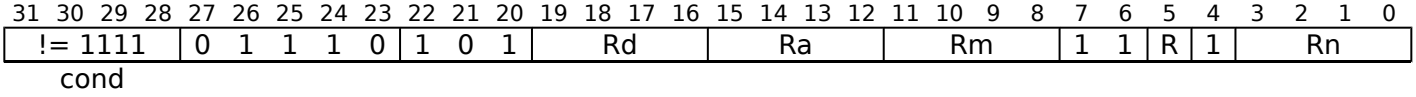
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SMMLS, SMMLSR

Signed Most Significant Word Multiply Subtract multiplies two signed 32-bit values, subtracts the result from a 32-bit accumulate value that is shifted left by 32 bits, and extracts the most significant 32 bits of the result of that subtraction.
Optionally, the instruction can specify that the result of the instruction is rounded instead of being truncated. In this case, the constant 0x80000000 is added to the result of the subtraction before the high word is extracted.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



SMMLS (R == 0)

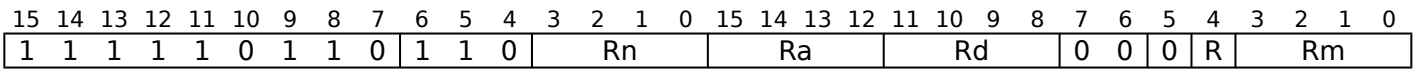
```
SMMLS{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

SMMLSR (R == 1)

```
SMMLSR{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);  round = (R == '1');
if d == 15 || n == 15 || m == 15 || a == 15 then UNPREDICTABLE;
```

T1



SMMLS (R == 0)

```
SMMLS{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

SMMLSR (R == 1)

```
SMMLSR{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);  round = (R == '1');
if d == 15 || n == 15 || m == 15 || a == 15 then UNPREDICTABLE;
// Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
<Ra>	Is the third general-purpose source register holding the addend, encoded in the "Ra" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = (SInt(R[a]) << 32) - SInt(R[n]) * SInt(R[m]);
    if round then result = result + 0x80000000;
    R[d] = result<63:32>;
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

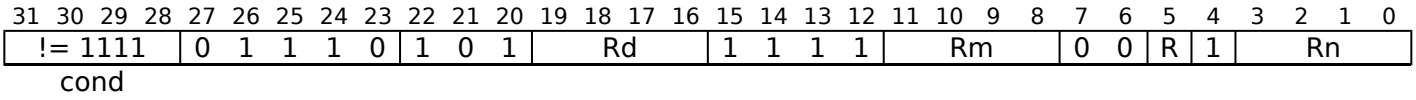
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SMMUL, SMMULR

Signed Most Significant Word Multiply multiplies two signed 32-bit values, extracts the most significant 32 bits of the result, and writes those bits to the destination register.
Optionally, the instruction can specify that the result is rounded instead of being truncated. In this case, the constant 0x80000000 is added to the product before the high word is extracted.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



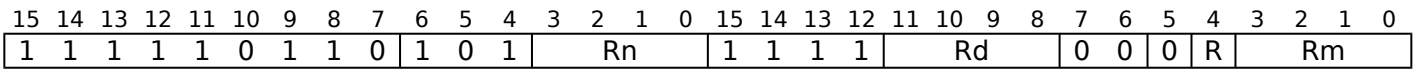
SMMUL (R == 0)

```
SMMUL{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

SMMULR (R == 1)

```
SMMULR{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  round = (R == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1



SMMUL (R == 0)

```
SMMUL{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

SMMULR (R == 1)

```
SMMULR{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  round = (R == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = SInt(R[n]) * SInt(R[m]);
    if round then result = result + 0x80000000;
    R[d] = result<63:32>;
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

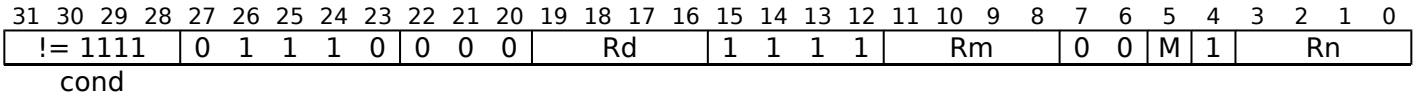
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SMUAD, SMUADX

Signed Dual Multiply Add performs two signed 16 x 16-bit multiplications. It adds the products together, and writes the result to the destination register.
Optionally, the instruction can exchange the halfwords of the second operand before performing the arithmetic. This produces top x bottom and bottom x top multiplication.
This instruction sets *PSTATE*.Q to 1 if the addition overflows. The multiplications cannot overflow.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



SMUAD (M == 0)

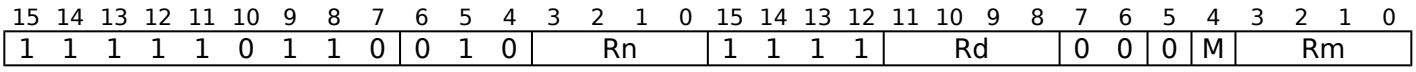
```
SMUAD{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

SMUADX (M == 1)

```
SMUADX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  m_swap = (M == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1



SMUAD (M == 0)

```
SMUAD{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

SMUADX (M == 1)

```
SMUADX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  m_swap = (M == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand2 = if m_swap then ROR(R[m],16) else R[m];
    product1 = SInt(R[n]<15:0>) * SInt(operand2<15:0>);
    product2 = SInt(R[n]<31:16>) * SInt(operand2<31:16>);
    result = product1 + product2;
    R[d] = result<31:0>;
    if result != SInt(result<31:0>) then  // Signed overflow
        PSTATE.Q = '1';
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

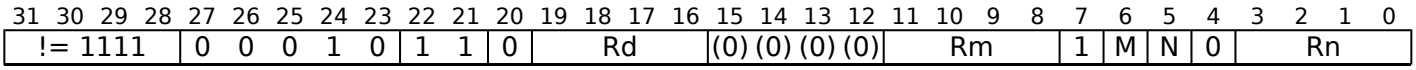
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SMULBB, SMULBT, SMULTB, SMULTT

Signed Multiply (halfwords) multiplies two signed 16-bit quantities, taken from either the bottom or the top half of their respective source registers. The other halves of these source registers are ignored. The 32-bit product is written to the destination register. No overflow is possible during this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



SMULBB (M == 0 && N == 0)

```
SMULBB{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

SMULBT (M == 1 && N == 0)

```
SMULBT{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

SMULTB (M == 0 && N == 1)

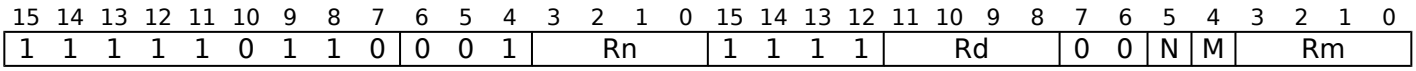
```
SMULTB{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

SMULTT (M == 1 && N == 1)

```
SMULTT{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
n_high = (N == '1');  m_high = (M == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1



SMULBB (N == 0 && M == 0)

```
SMULBB{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

SMULBT (N == 0 && M == 1)

```
SMULBT{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

SMULTB (N == 1 && M == 0)

```
SMULTB{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

SMULTT (N == 1 && M == 1)

```
SMULTT{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
n_high = (N == '1');  m_high = (M == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register holding the multiplicand in the bottom or top half (selected by <x>), encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register holding the multiplier in the bottom or top half (selected by <y>), encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand1 = if n_high then R[n]<31:16> else R[n]<15:0>;
    operand2 = if m_high then R[m]<31:16> else R[m]<15:0>;
    result = SInt(operand1) * SInt(operand2);
    R[d] = result<31:0>;
    // Signed overflow cannot occur
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

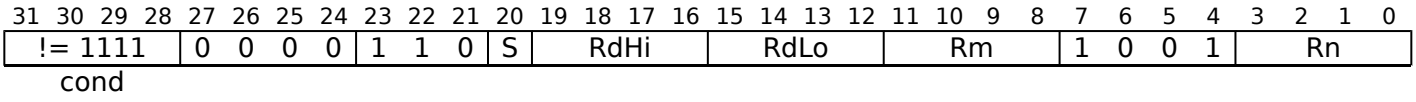
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SMULL, SMULLS

Signed Multiply Long multiplies two 32-bit signed values to produce a 64-bit result.
In A32 instructions, the condition flags can optionally be updated based on the result. Use of this option adversely affects performance on many implementations.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



Flag setting (S == 1)

```
SMULLS{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

Not flag setting (S == 0)

```
SMULL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

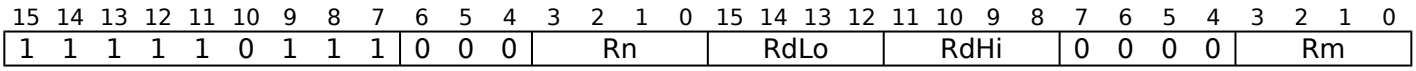
```
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
if dHi == dLo then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `dHi == dLo`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

T1



T1

```
SMULL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

```
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);  setflags = FALSE;
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
// Armv8-A removes UNPREDICTABLE for R13
if dHi == dLo then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `dHi == dLo`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<RdLo>	Is the general-purpose destination register for the lower 32 bits of the result, encoded in the "RdLo" field.
<RdHi>	Is the general-purpose destination register for the upper 32 bits of the result, encoded in the "RdHi" field.
<Rn>	Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = SInt(R[n]) * SInt(R[m]);
    R[dHi] = result<63:32>;
    R[dLo] = result<31:0>;
    if setflags then
        PSTATE.N = result<63>;
        PSTATE.Z = IsZeroBit(result<63:0>);
        // PSTATE.C, PSTATE.V unchanged
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

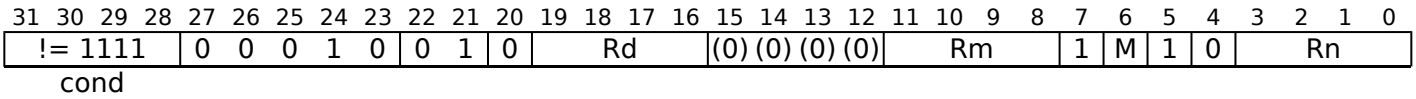
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SMULWB, SMULWT

Signed Multiply (word by halfword) multiplies a signed 32-bit quantity and a signed 16-bit quantity. The signed 16-bit quantity is taken from either the bottom or the top half of its source register. The other half of the second source register is ignored. The top 32 bits of the 48-bit product are written to the destination register. The bottom 16 bits of the 48-bit product are ignored. No overflow is possible during this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



SMULWB (M == 0)

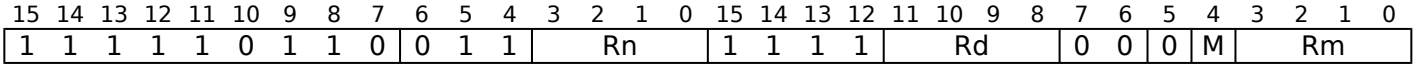
```
SMULWB{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

SMULWT (M == 1)

```
SMULWT{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  m_high = (M == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1



SMULWB (M == 0)

```
SMULWB{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

SMULWT (M == 1)

```
SMULWT{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  m_high = (M == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register holding the multiplier in the bottom or top half (selected by <y>), encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand2 = if m_high then R[m]<31:16> else R[m]<15:0>;
    product = SInt(R[n]) * SInt(operand2);
    R[d] = product<47:16>;
    // Signed overflow cannot occur
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

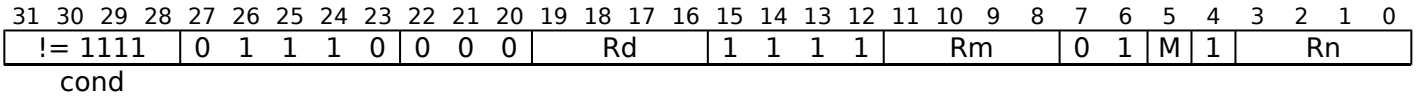
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SMUSD, SMUSDX

Signed Multiply Subtract Dual performs two signed 16 x 16-bit multiplications. It subtracts one of the products from the other, and writes the result to the destination register.
Optionally, the instruction can exchange the halfwords of the second operand before performing the arithmetic. This produces top x bottom and bottom x top multiplication.
Overflow cannot occur.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



SMUSD (M == 0)

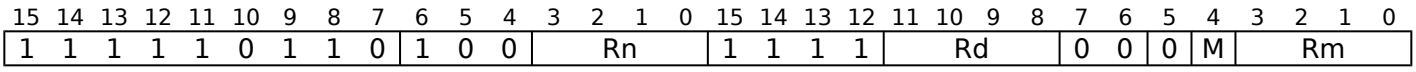
```
SMUSD{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

SMUSDX (M == 1)

```
SMUSDX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  m_swap = (M == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1



SMUSD (M == 0)

```
SMUSD{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

SMUSDX (M == 1)

```
SMUSDX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  m_swap = (M == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand2 = if m_swap then ROR(R[m],16) else R[m];
    product1 = SInt(R[n]<15:0>) * SInt(operand2<15:0>);
    product2 = SInt(R[n]<31:16>) * SInt(operand2<31:16>);
    result = product1 - product2;
    R[d] = result<31:0>;
    // Signed overflow cannot occur
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SRS, SRSDA, SRSDB, SRSIA, SRSIB

Store Return State stores the LR_<current_mode> and *SPSR*_<current_mode> to the stack of a specified mode. For information about memory accesses see *Memory accesses*.

SRS is UNDEFINED in Hyp mode.

SRS is CONSTRAINED UNPREDICTABLE if it is executed in User or System mode, or if the specified mode is any of the following:

- Not implemented.
- A mode that *Table G1-5* does not show.
- Hyp mode.
- Monitor mode, if the SRS instruction is executed in Non-secure state.

If EL3 is using AArch64 and an SRS instruction that is executed in a Secure EL1 mode specifies Monitor mode, it is trapped to EL3.

See *Traps to EL3 of Secure monitor functionality from Secure EL1 using AArch32*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	P	U	1	W	0	(1)	(1)	(0)	(1)	(0)	(0)	(0)	(0)	(0)	(1)	(0)	(1)	(0)	(0)	(0)	mode				

Decrement After (P == 0 && U == 0)

```
SRSDA{<c>}{<q>} SP{!}, #<mode>
```

Decrement Before (P == 1 && U == 0)

```
SRSDB{<c>}{<q>} SP{!}, #<mode>
```

Increment After (P == 0 && U == 1)

```
SRS{IA}{<c>}{<q>} SP{!}, #<mode>
```

Increment Before (P == 1 && U == 1)

```
SRSIB{<c>}{<q>} SP{!}, #<mode>
```

```
wback = (W == '1');  increment = (U == '1');  wordhigher = (P == U);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	0	0	0	W	0	(1)	(1)	(0)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	mode				

T1

```
SRSDB{<c>}{<q>} SP{!}, #<mode>
```

```
wback = (W == '1');  increment = FALSE;  wordhigher = FALSE;
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	1	1	0	W	0	(1)	(1)	(0)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	mode				

T2

```
SRS{IA}{<c>}{<q>} SP{!}, #<mode>
```

```
wback = (W == '1');  increment = TRUE;  wordhigher = FALSE;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *SRS (T32)* and *SRS (A32)*.

Assembler Symbols

IA	For encoding A1: is an optional suffix to indicate the Increment After variant. For encoding T2: is an optional suffix for the Increment After form.
<c>	For encoding A1: see *Standard assembler syntax fields*. <c> must be AL or omitted. For encoding T1 and T2: see *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
!	The address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the "W" field as 1, otherwise this field defaults to 0.
<mode>	Is the number of the mode whose Banked SP is used as the base register, encoded in the "mode" field. For details of PE modes and their numbers see *AArch32 PE mode descriptions*.

SRSFA, SRSEA, SRSFD, and SRSED are pseudo-instructions for SRSIB, SRSIA, SRSDB, and SRSDA respectively, referring to their use for pushing data onto Full Ascending, Empty Ascending, Full Descending, and Empty Descending stacks.

Operation

```
if CurrentInstrSet() == InstrSet_A32 then
    if ConditionPassed() then
        EncodingSpecificOperations();
        if PSTATE.EL == EL2 then          // UNDEFINED at EL2
            UNDEFINED;

        // Check for UNPREDICTABLE cases. The definition of UNPREDICTABLE does not permit these
        // to be security holes
        if PSTATE.M IN {M32_User,M32_System} then
            UNPREDICTABLE;
        elsif mode == M32_Hyp then        // Check for attempt to access Hyp mode SP
            UNPREDICTABLE;
        elsif mode == M32_Monitor then    // Check for attempt to access Monitor mode SP
            if !HaveEL(EL3) || !IsSecure()  then
                UNPREDICTABLE;
            elsif !ELUsingAArch32(EL3) then
                AArch64.MonitorModeTrap();
        elsif BadMode(mode) then
            UNPREDICTABLE;

        base = Rmode[13,mode];
        address = if increment then base else base-8;
        if wordhigher then address = address+4;
        MemA[address,4]   = LR;
        MemA[address+4,4] = SPSR[];
        if wback then Rmode[13,mode] = if increment then base+8 else base-8;
else
    if ConditionPassed() then
        EncodingSpecificOperations();
        if PSTATE.EL == EL2 then          // UNDEFINED at EL2
            UNDEFINED;

        // Check for UNPREDICTABLE cases. The definition of UNPREDICTABLE does not permit these
        // to be security holes
        if PSTATE.M IN {M32_User,M32_System} then
            UNPREDICTABLE;
        elsif mode == M32_Hyp then        // Check for attempt to access Hyp mode SP
            UNPREDICTABLE;
        elsif mode == M32_Monitor then    // Check for attempt to access Monitor mode SP
            if !HaveEL(EL3) || !IsSecure()  then
                UNPREDICTABLE;
            elsif !ELUsingAArch32(EL3) then
                AArch64.MonitorModeTrap();
        elsif BadMode(mode) then
            UNPREDICTABLE;

        base = Rmode[13,mode];
        address = if increment then base else base-8;
        if wordhigher then address = address+4;
        MemA[address,4]   = LR;
        MemA[address+4,4] = SPSR[];
        if wback then Rmode[13,mode] = if increment then base+8 else base-8;
```

CONSTRAINED UNPREDICTABLE behavior

If `PSTATE.M IN {M32_User,M32_System}`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

If `mode == M32_Hyp`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

If `mode == M32_Monitor && (!HaveEL(EL3) || !IsSecure())`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.

If `BadMode(mode)`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction stores to the stack of the mode in which it is executed.
- The instruction stores to an UNKNOWN address, and if the instruction specifies writeback then any general-purpose register that can be accessed from the current Exception level without a privilege violation becomes UNKNOWN.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

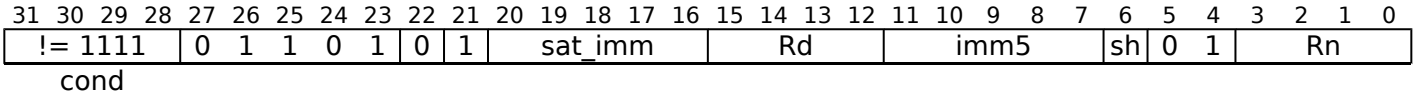
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SSAT

Signed Saturate saturates an optionally-shifted signed value to a selectable signed range. This instruction sets *PSTATE*.Q to 1 if the operation saturates.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



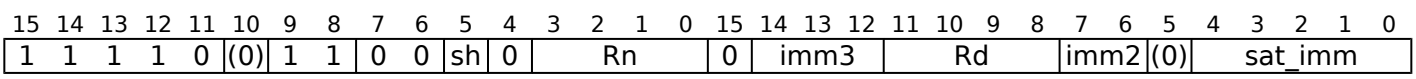
Arithmetic shift right (sh == 1)

```
SSAT{<c>}{<q>} <Rd>, #<imm>, <Rn>, ASR #<amount>
```

Logical shift left (sh == 0)

```
SSAT{<c>}{<q>} <Rd>, #<imm>, <Rn> {, LSL #<amount>}

d = UInt(Rd);  n = UInt(Rn);  saturate_to = UInt(sat_imm)+1;
(shift_t, shift_n) = DecodeImmShift(sh:'0', imm5);
if d == 15 || n == 15 then UNPREDICTABLE;
```

T1



Arithmetic shift right (sh == 1 && !(imm3 == 000 && imm2 == 00))

```
SSAT{<c>}{<q>} <Rd>, #<imm>, <Rn>, ASR #<amount>
```

Logical shift left (sh == 0)

```
SSAT{<c>}{<q>} <Rd>, #<imm>, <Rn> {, LSL #<amount>}

if sh == '1' && (imm3:imm2) == '00000' then SEE "SSAT16";
d = UInt(Rd);  n = UInt(Rn);  saturate_to = UInt(sat_imm)+1;
(shift_t, shift_n) = DecodeImmShift(sh:'0', imm3:imm2);
if d == 15 || n == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<imm>	Is the bit position for saturation, in the range 1 to 32, encoded in the "sat_imm" field as <imm>-1.
<Rn>	Is the general-purpose source register, encoded in the "Rn" field.
<amount>	For encoding A1: is the optional shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm5" field. For encoding A1: is the shift amount, in the range 1 to 32 encoded in the "imm5" field as <amount> modulo 32.

For encoding T1: is the optional shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm3:imm2" field.

For encoding T1: is the shift amount, in the range 1 to 31 encoded in the "imm3:imm2" field as <amount>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand = Shift(R[n], shift_t, shift_n, PSTATE.C);  // PSTATE.C ignored
    (result, sat) = SignedSatQ(SInt(operand), saturate_to);
    R[d] = SignExtend(result, 32);
    if sat then
        PSTATE.Q = '1';
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

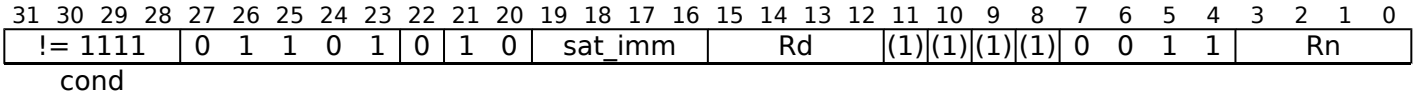
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SSAT16

Signed Saturate 16 saturates two signed 16-bit values to a selected signed range.
This instruction sets *PSTATE*.Q to 1 if the operation saturates.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

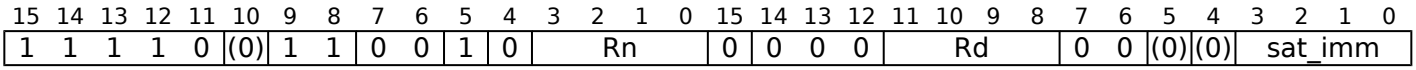


A1

```
SSAT16{<c>}{<q>} <Rd>, #<imm>, <Rn>
```

```
d = UInt(Rd);  n = UInt(Rn);  saturate_to = UInt(sat_imm)+1;
if d == 15 || n == 15 then UNPREDICTABLE;
```

T1



T1

```
SSAT16{<c>}{<q>} <Rd>, #<imm>, <Rn>
```

```
d = UInt(Rd);  n = UInt(Rn);  saturate_to = UInt(sat_imm)+1;
if d == 15 || n == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<imm>	Is the bit position for saturation, in the range 1 to 16, encoded in the "sat_imm" field as <imm>-1.
<Rn>	Is the general-purpose source register, encoded in the "Rn" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result1, sat1) = SignedSatQ(SInt(R[n]<15:0>), saturate_to);
    (result2, sat2) = SignedSatQ(SInt(R[n]<31:16>), saturate_to);
    R[d]<15:0> = SignExtend(result1, 16);
    R[d]<31:16> = SignExtend(result2, 16);
    if sat1 || sat2 then
        PSTATE.Q = '1';
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

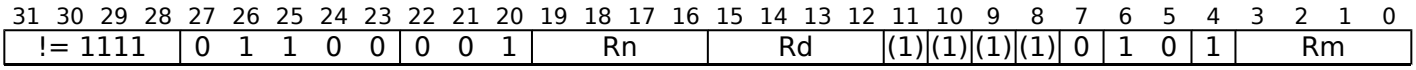
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SSAX

Signed Subtract and Add with Exchange exchanges the two halfwords of the second operand, performs one 16-bit integer subtraction and one 16-bit addition, and writes the results to the destination register. It sets *PSTATE*.GE according to the results.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

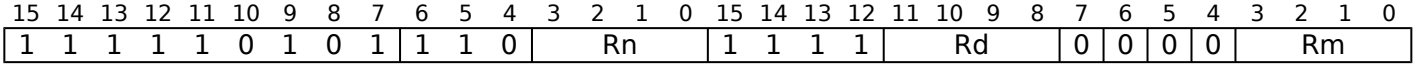
A1



A1

```
SSAX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1



T1

```
SSAX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum  = SInt(R[n]<15:0>) + SInt(R[m]<31:16>);
    diff = SInt(R[n]<31:16>) - SInt(R[m]<15:0>);
    R[d]<15:0>  = sum<15:0>;
    R[d]<31:16> = diff<15:0>;
    PSTATE.GE<1:0> = if sum  >= 0 then '11' else '00';
    PSTATE.GE<3:2> = if diff >= 0 then '11' else '00';
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SSBB

Speculative Store Bypass Barrier is a memory barrier which prevents speculative loads from bypassing earlier stores to the same virtual address under certain conditions.

The semantics of the Speculative Store Bypass Barrier are:

- When a load to a location appears in program order after the SSBB, then the load does not speculatively read an entry earlier in the coherence order for that location than the entry generated by the latest store satisfying all of the following conditions:
  - The store is to the same location as the load.
  - The store uses the same virtual address as the load.
  - The store appears in program order before the SSBB.
- When a load to a location appears in program order before the SSBB, then the load does not speculatively read data from any store satisfying all of the following conditions:
  - The store is to the same location as the load.
  - The store uses the same virtual address as the load.
  - The store appears in program order after the SSBB.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	1	0	1	1	1	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	0	1	0	0	0	0	0	0

A1

```
SSBB{<q>}

// No additional decoding required
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	1	(1)	(1)	(1)	(1)	1	0	(0)	0	(1)	(1)	(1)	(1)	0	1	0	0	0	0	0	0

T1

```
SSBB{<q>}

if InITBlock() then UNPREDICTABLE;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<q>	See *Standard assembler syntax fields*.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    SpeculativeStoreBypassBarrierToVA();
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

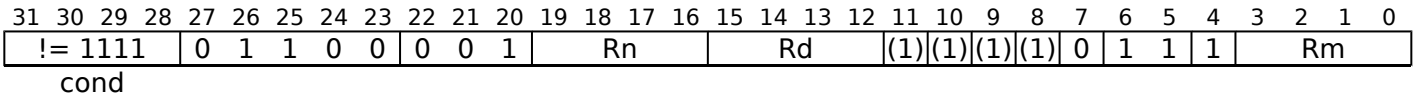
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SSUB16

Signed Subtract 16 performs two 16-bit signed integer subtractions, and writes the results to the destination register. It sets *PSTATE*.GE according to the results of the subtractions.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

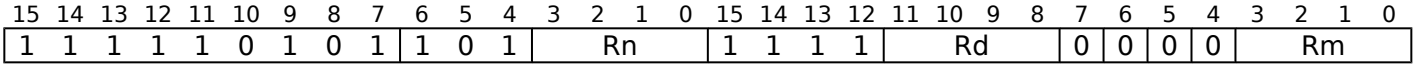
A1



A1

```
SSUB16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1



T1

```
SSUB16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = SInt(R[n]<15:0>) - SInt(R[m]<15:0>);
    diff2 = SInt(R[n]<31:16>) - SInt(R[m]<31:16>);
    R[d]<15:0>  = diff1<15:0>;
    R[d]<31:16> = diff2<15:0>;
    PSTATE.GE<1:0> = if diff1 >= 0 then '11' else '00';
    PSTATE.GE<3:2> = if diff2 >= 0 then '11' else '00';
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:

    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

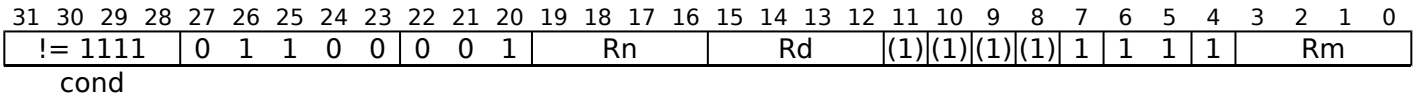
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SSUB8

Signed Subtract 8 performs four 8-bit signed integer subtractions, and writes the results to the destination register. It sets *PSTATE*.GE according to the results of the subtractions.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

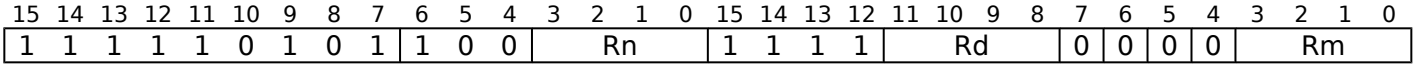
A1



A1

```
SSUB8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1



T1

```
SSUB8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = SInt(R[n]<7:0>) - SInt(R[m]<7:0>);
    diff2 = SInt(R[n]<15:8>) - SInt(R[m]<15:8>);
    diff3 = SInt(R[n]<23:16>) - SInt(R[m]<23:16>);
    diff4 = SInt(R[n]<31:24>) - SInt(R[m]<31:24>);
    R[d]<7:0>   = diff1<7:0>;
    R[d]<15:8>  = diff2<7:0>;
    R[d]<23:16> = diff3<7:0>;
    R[d]<31:24> = diff4<7:0>;
    PSTATE.GE<0>  = if diff1 >= 0 then '1' else '0';
    PSTATE.GE<1>  = if diff2 >= 0 then '1' else '0';
    PSTATE.GE<2>  = if diff3 >= 0 then '1' else '0';
    PSTATE.GE<3>  = if diff4 >= 0 then '1' else '0';
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STC

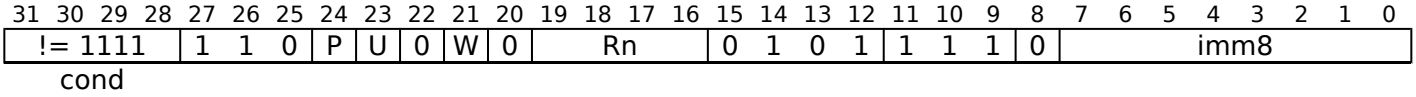
Store data to System register calculates an address from a base register value and an immediate offset, and stores a word from the *DBGDTRRXint* System register to memory. It can use offset, post-indexed, pre-indexed, or unindexed addressing. For information about memory accesses, see *Memory accesses*.

In an implementation that includes EL2, the permitted STC access to *DBGDTRRXint* can be trapped to Hyp mode, meaning that an attempt to execute an STC instruction in a Non-secure mode other than Hyp mode, that would be permitted in the absence of the Hyp trap controls, generates a Hyp Trap exception. For more information, see *Trapping general Non-secure System register accesses to debug registers*.

For simplicity, the STC pseudocode does not show this possible trap to Hyp mode.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



Offset (P == 1 && W == 0)

```
STC{<c>}{<q>} p14, c5, [<Rn>{, #{+/-}<imm>}]
```

Post-indexed (P == 0 && W == 1)

```
STC{<c>}{<q>} p14, c5, [<Rn>], #{+/-}<imm>
```

Pre-indexed (P == 1 && W == 1)

```
STC{<c>}{<q>} p14, c5, [<Rn>, #{+/-}<imm>]!
```

Unindexed (P == 0 && U == 1 && W == 0)

```
STC{<c>}{<q>} p14, c5, [<Rn>], <option>
```

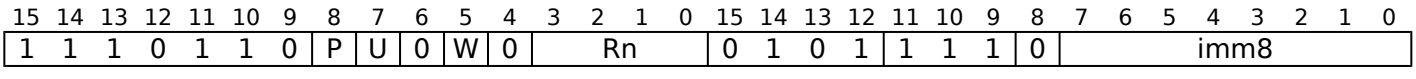
```
if P == '0' && U == '0' && W == '0' then UNDEFINED;
n = UInt(Rn);  cp = 14;
imm32 = ZeroExtend(imm8:'00', 32);  index = (P == '1');  add = (U == '1');  wback = (W == '1');
if n == 15 && (wback || CurrentInstrSet() != InstrSet_A32) then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `n == 15 && wback`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes without writeback of the base address.
- The instruction executes with writeback to the PC. The instruction is handled as described in *Using R15*.

T1



Offset (P == 1 && W == 0)

```
STC{<c>}{<q>} p14, c5, [<Rn>{, #{+/-}<imm>}]
```

Post-indexed (P == 0 && W == 1)

```
STC{<c>}{<q>} p14, c5, [<Rn>], #{+/-}<imm>
```

Pre-indexed (P == 1 && W == 1)

```
STC{<c>}{<q>} p14, c5, [<Rn>, #{+/-}<imm>]!
```

Unindexed (P == 0 && U == 1 && W == 0)

```
STC{<c>}{<q>} p14, c5, [<Rn>], <option>
```

```
if P == '0' && U == '0' && W == '0' then UNDEFINED;
n = UInt(Rn);  cp = 14;
imm32 = ZeroExtend(imm8:'00', 32);  index = (P == '1');  add = (U == '1');  wback = (W == '1');
if n == 15 && (wback || CurrentInstrSet() != InstrSet_A32) then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `n == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes without writeback of the base address.
- The instruction executes with writeback to the PC. The instruction is handled as described in *Using R15*.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rn>	For the offset or unindexed variant: is the general-purpose base register, encoded in the "Rn" field. The PC can be used, but this is deprecated. For the offset, post-indexed or pre-indexed variant: is the general-purpose base register, encoded in the "Rn" field.
<option>	Is an 8-bit immediate, in the range 0 to 255 enclosed in { }, encoded in the "imm8" field. The value of this field is ignored when executing this instruction.
+/-	Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<imm>	Is the immediate offset used for forming the address, a multiple of 4 in the range 0-1020, defaulting to 0 and encoded in the "imm8" field, as <imm>/4.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];

    // System register read from DBGDTRRXint.
    MemA[address,4] = AArch32.SysRegRead(cp, ThisInstr());
    if wback then R[n] = offset_addr;
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

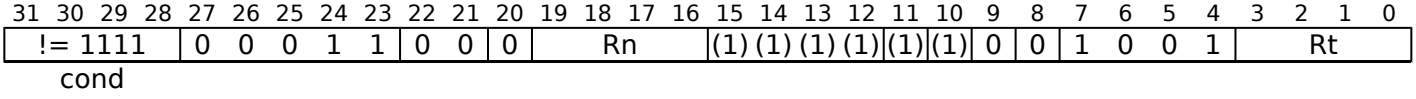
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STL

Store-Release Word stores a word from a register to memory. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release*.
For more information about support for shared memory see *Synchronization and semaphores*. For information about memory accesses see *Memory accesses*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

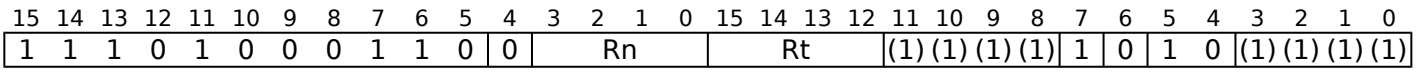
A1



A1

```
STL{<c>}{<q>} <Rt>, [<Rn>]

t = UInt(Rt); n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

T1



T1

```
STL{<c>}{<q>} <Rt>, [<Rn>]

t = UInt(Rt); n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rt>	Is the general-purpose register to be transferred, encoded in the "Rt" field.
<Rn>	Is the general-purpose base register, encoded in the "Rn" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    MemO[address, 4] = R[t];
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

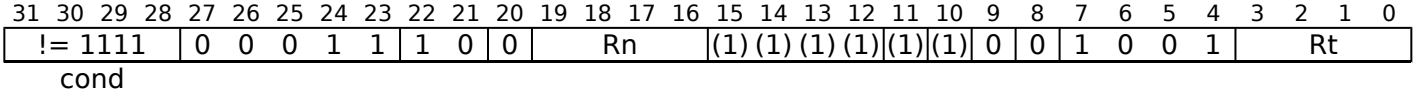
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STLB

Store-Release Byte stores a byte from a register to memory. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release*.
For more information about support for shared memory see *Synchronization and semaphores*. For information about memory accesses see *Memory accesses*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

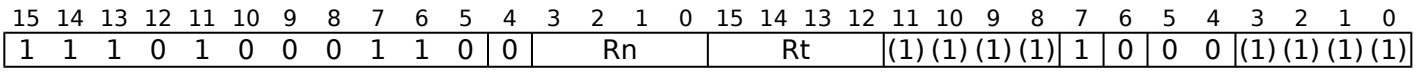
A1



A1

```
STLB{<c>}{<q>} <Rt>, [<Rn>]

t = UInt(Rt); n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

T1



T1

```
STLB{<c>}{<q>} <Rt>, [<Rn>]

t = UInt(Rt); n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rt>	Is the general-purpose register to be transferred, encoded in the "Rt" field.
<Rn>	Is the general-purpose base register, encoded in the "Rn" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    MemO[address, 1] = R[t]<7:0>;
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

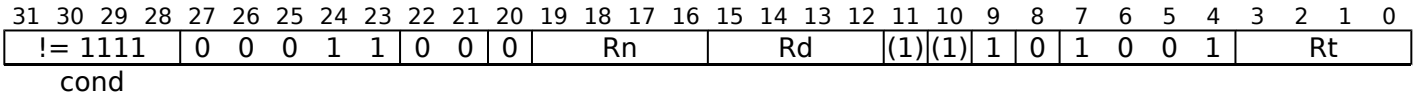
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STLEX

Store-Release Exclusive Word stores a word from a register to memory if the executing PE has exclusive access to the memory at that address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed.
The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release*.
For more information about support for shared memory see *Synchronization and semaphores*. For information about memory accesses see *Memory accesses*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



A1

```
STLEX{<c>}{<q>} <Rd>, <Rt>, [<Rn>]

d = UInt(Rd);  t = UInt(Rt);  n = UInt(Rn);
if d == 15 || t == 15 || n == 15 then UNPREDICTABLE;
if d == n || d == t then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

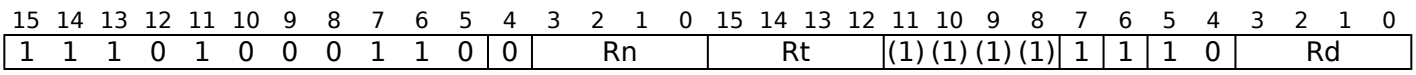
If `d == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction executes but the value stored is UNKNOWN.

If `d == n`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs the store to an UNKNOWN address.

T1



T1

```
STLEX{<c>}{<q>} <Rd>, <Rt>, [<Rn>]

d = UInt(Rd);  t = UInt(Rt);  n = UInt(Rn);
if d == 15 || t == 15 || n == 15 then UNPREDICTABLE;
if d == n || d == t then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction executes but the value stored is UNKNOWN.

If `d == n`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.

- The instruction performs the store to an UNKNOWN address.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<Rd> Is the destination general-purpose register into which the status result of the store exclusive is written, encoded in the "Rd" field. The value returned is:

0
If the operation updates memory.

1
If the operation fails to update memory.

<Rt> Is the general-purpose register to be transferred, encoded in the "Rt" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Rd> is not updated.

A non word-aligned memory address causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If AArch32.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch32.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    if AArch32.ExclusiveMonitorsPass(address,4) then
        MemO[address, 4] = R[t];
        R[d] = ZeroExtend('0');
    else
        R[d] = ZeroExtend('1');
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

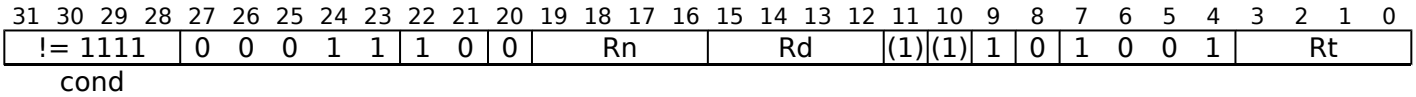
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STLEXB

Store-Release Exclusive Byte stores a byte from a register to memory if the executing PE has exclusive access to the memory at that address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release*.
For more information about support for shared memory see *Synchronization and semaphores*. For information about memory accesses see *Memory accesses*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



A1

```
STLEXB{<c>}{<q>} <Rd>, <Rt>, [<Rn>]

d = UInt(Rd);  t = UInt(Rt);  n = UInt(Rn);
if d == 15 || t == 15 || n == 15 then UNPREDICTABLE;
if d == n || d == t then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

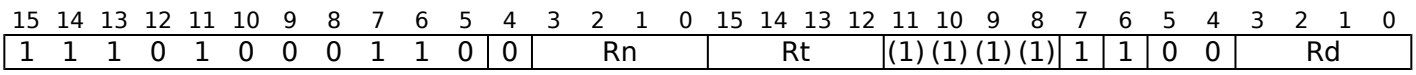
If `d == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If `d == n`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs the store to an UNKNOWN address.

T1



T1

```
STLEXB{<c>}{<q>} <Rd>, <Rt>, [<Rn>]

d = UInt(Rd);  t = UInt(Rt);  n = UInt(Rn);
if d == 15 || t == 15 || n == 15 then UNPREDICTABLE;
if d == n || d == t then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If `d == n`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

- The instruction performs the store to an UNKNOWN address.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<Rd> Is the destination general-purpose register into which the status result of the store exclusive is written, encoded in the "Rd" field. The value returned is:

0
If the operation updates memory.

1
If the operation fails to update memory.

<Rt> Is the general-purpose register to be transferred, encoded in the "Rt" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

Aborts

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Rd> is not updated.

If AArch32.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    if AArch32.ExclusiveMonitorsPass(address,1) then
        MemO[address, 1] = R[t]<7:0>;
        R[d] = ZeroExtend('0');
    else
        R[d] = ZeroExtend('1');
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STLEXD

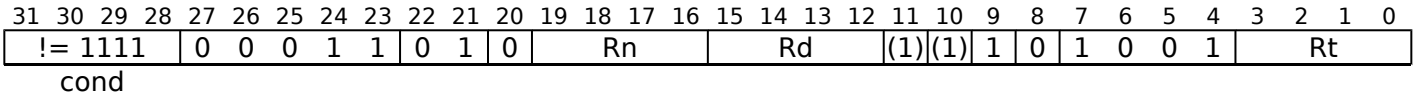
Store-Release Exclusive Doubleword stores a doubleword from two registers to memory if the executing PE has exclusive access to the memory at that address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed.

The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release*.

For more information about support for shared memory see *Synchronization and semaphores*. For information about memory accesses see *Memory accesses*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



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.

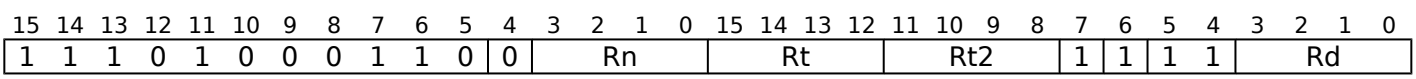
If `Rt<0> == '1'`, then one of the following behaviors must occur:

- 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



T1

```
STLEXD{<c>}{<q>} <Rd>, <Rt>, <Rt2>, [<Rn>]
```

```
d = UInt(Rd);  t = UInt(Rt);  t2 = UInt(Rt2);  n = UInt(Rn);
if d == 15 || t == 15 || t2 == 15 || n == 15 then UNPREDICTABLE;
if d == n || d == t || d == t2 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction executes but the value stored is UNKNOWN.

If `d == n`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs the store to an UNKNOWN address.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<Rd> Is the destination general-purpose register into which the status result of the store exclusive is written, encoded in the "Rd" field. The value returned is:

0
If the operation updates memory.

1
If the operation fails to update memory.

<Rt> For encoding A1: is the first general-purpose register to be transferred, encoded in the "Rt" field. <Rt> must be even-numbered and not R14.

For encoding T1: is the first general-purpose register to be transferred, encoded in the "Rt" field.

<Rt2> For encoding A1: is the second general-purpose register to be transferred. <Rt2> must be <R(t+1)>.

For encoding T1: is the second general-purpose register to be transferred, encoded in the "Rt2" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Rd> is not updated.

A non word-aligned memory address causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If AArch32.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch32.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    // Create doubleword to store such that R[t] will be stored at address and R[t2] at address+4.
    value = if BigEndian(AccType_ORDERED) then R[t]:R[t2] else R[t2]:R[t];
    if AArch32.ExclusiveMonitorsPass(address, 8) then
        MemO[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_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

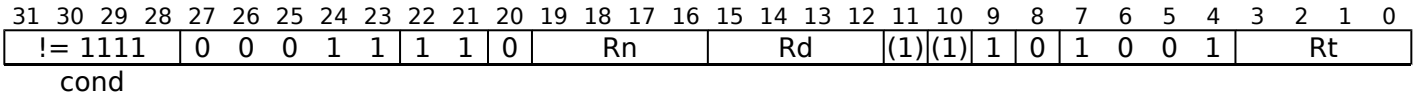
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STLEXH

Store-Release Exclusive Halfword stores a halfword from a register to memory if the executing PE has exclusive access to the memory at that address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed.
The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release*.
For more information about support for shared memory see *Synchronization and semaphores*. For information about memory accesses see *Memory accesses*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



A1

```
STLEXH{<c>}{<q>} <Rd>, <Rt>, [<Rn>]

d = UInt(Rd);  t = UInt(Rt);  n = UInt(Rn);
if d == 15 || t == 15 || n == 15 then UNPREDICTABLE;
if d == n || d == t then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

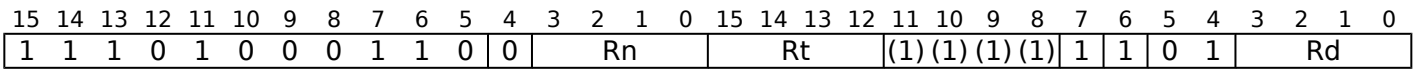
If `d == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction executes but the value stored is UNKNOWN.

If `d == n`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs the store to an UNKNOWN address.

T1



T1

```
STLEXH{<c>}{<q>} <Rd>, <Rt>, [<Rn>]

d = UInt(Rd);  t = UInt(Rt);  n = UInt(Rn);
if d == 15 || t == 15 || n == 15 then UNPREDICTABLE;
if d == n || d == t then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction executes but the value stored is UNKNOWN.

If `d == n`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.

- The instruction executes as NOP.
- The instruction performs the store to an UNKNOWN address.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<Rd> Is the destination general-purpose register into which the status result of the store exclusive is written, encoded in the "Rd" field. The value returned is:

0
If the operation updates memory.

1
If the operation fails to update memory.

<Rt> Is the general-purpose register to be transferred, encoded in the "Rt" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated
- <Rd> is not updated.

A non word-aligned memory address causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If AArch32.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch32.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    if AArch32.ExclusiveMonitorsPass(address,2) then
        MemO[address, 2] = R[t]<15:0>;
        R[d] = ZeroExtend('0');
    else
        R[d] = ZeroExtend('1');
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

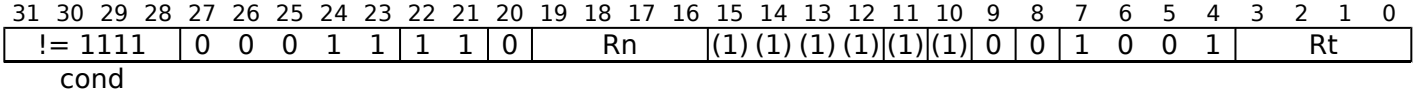
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STLH

Store-Release Halfword stores a halfword from a register to memory. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release*.
For more information about support for shared memory see *Synchronization and semaphores*. For information about memory accesses see *Memory accesses*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

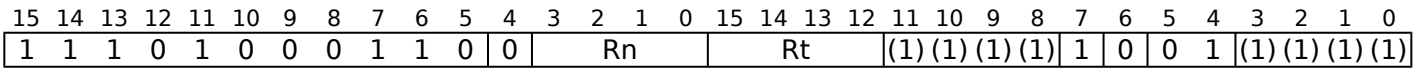
A1



A1

```
STLH{<c>}{<q>} <Rt>, [<Rn>]

t = UInt(Rt); n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

T1



T1

```
STLH{<c>}{<q>} <Rt>, [<Rn>]

t = UInt(Rt); n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rt>	Is the general-purpose register to be transferred, encoded in the "Rt" field.
<Rn>	Is the general-purpose base register, encoded in the "Rn" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    MemO[address, 2] = R[t]<15:0>;
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STM, STMIA, STMEA

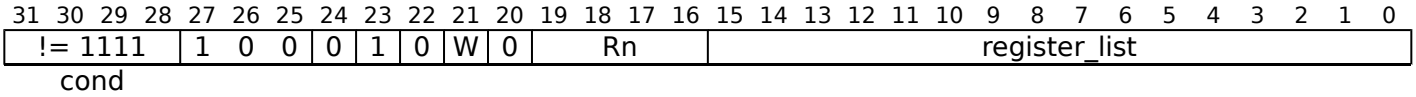
Store Multiple (Increment After, Empty Ascending) stores multiple registers to consecutive memory locations using an address from a base register. The consecutive memory locations start at this address, and the address just above the last of those locations can optionally be written back to the base register.

The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. See also *Encoding of lists of general-purpose registers and the PC*.

Armv8.2 permits the deprecation of some Store Multiple ordering behaviors in AArch32 state, for more information see *FEAT_LSMAOC*. For details of related system instructions see *STM (User registers)*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2) .

A1



A1

```
STM{IA}{<c>}{<q>} <Rn>{!}, <registers> // (Preferred syntax)

STMEA{<c>}{<q>} <Rn>{!}, <registers> // (Alternate syntax, Empty Ascending stack)
```

```
n = UInt(Rn);  registers = register_list;  wback = (W == '1');
if n == 15 || BitCount(registers) < 1 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

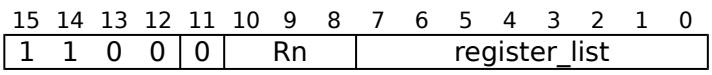
If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction operates as an `STM` with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers stored.

If `n == 15 && wback`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes without writeback of the base address.
- The instruction executes with writeback to the PC. The instruction is handled as described in *Using R15*.

T1



T1

```
STM{IA}{<c>}{<q>} <Rn>!, <registers> // (Preferred syntax)

STMEA{<c>}{<q>} <Rn>!, <registers> // (Alternate syntax, Empty Ascending stack)
```

```
n = UInt(Rn);  registers = '00000000':register_list;  wback = TRUE;
if BitCount(registers) < 1 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

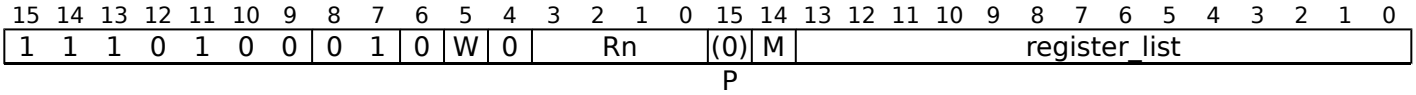
- The instruction is UNDEFINED.

- The instruction executes as `NOP`.
- The instruction operates as an `STM` with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers stored.

If `n == 15 && wback`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes without writeback of the base address.
- The instruction executes with writeback to the PC. The instruction is handled as described in *Using R15*.

T2



T2

```
STM{IA}{<c>}.W <Rn>{!}, <registers> // (Preferred syntax, if <Rn>, '!' and <registers> can be represented

STMEA{<c>}.W <Rn>{!}, <registers> // (Alternate syntax, Empty Ascending stack, if <Rn>, '!' and <register

STM{IA}{<c>}{<q>} <Rn>{!}, <registers> // (Preferred syntax)

STMEA{<c>}{<q>} <Rn>{!}, <registers> // (Alternate syntax, Empty Ascending stack)
```

```
n = UInt(Rn);  registers = P:M:register_list;  wback = (W == '1');
if n == 15 || BitCount(registers) < 2 then UNPREDICTABLE;
if wback && registers<n> == '1' then UNPREDICTABLE;
if registers<13> == '1' then UNPREDICTABLE;
if registers<15> == '1' then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction operates as an `STM` with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers stored.

If `BitCount(registers) == 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes as described, with no change to its behavior and no additional side effects.
- The instruction operates as an `STM` with the same addressing mode but targeting an unspecified set of registers. These registers might include R15.

If `wback && registers<n> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction executes but the value stored for the base register is UNKNOWN.

If `registers<13> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction performs all of the stores using the specified addressing mode but the value of R13 is UNKNOWN.

If `registers<15> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.

- The instruction executes as NOP.
- The store instruction performs all of the stores using the specified addressing mode but the value of R15 is UNKNOWN.

If `n == 15 && wback`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes without writeback of the base address.
- The instruction executes with writeback to the PC. The instruction is handled as described in *Using R15*.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

IA	Is an optional suffix for the Increment After form.
<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rn>	Is the general-purpose base register, encoded in the "Rn" field.
!	The address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the "W" field as 1, otherwise this field defaults to 0.
<registers>	For encoding A1: is a list of one or more registers to be stored, separated by commas and surrounded by { and }. The PC can be in the list. However, Arm deprecates the use of instructions that include the PC in the list. If base register writeback is specified, and the base register is not the lowest-numbered register in the list, such an instruction stores an UNKNOWN value for the base register. For encoding T1: is a list of one or more registers to be stored, separated by commas and surrounded by { and }. The registers in the list must be in the range R0-R7, encoded in the "register_list" field. If the base register is not the lowest-numbered register in the list, such an instruction stores an UNKNOWN value for the base register. For encoding T2: is a list of one or more registers to be stored, separated by commas and surrounded by { and }. The registers in the list must be in the range R0-R12, encoded in the "register_list" field, and can optionally contain the LR. If the LR is in the list, the "M" field is set to 1, otherwise it defaults to 0.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    for i = 0 to 14
        if registers<i> == '1' then
            if i == n && wback && i != LowestSetBit(registers) then
                MemS[address,4] = bits(32) UNKNOWN;  // Only possible for encodings T1 and A1
            else
                MemS[address,4] = R[i];
            address = address + 4;
    if registers<15> == '1' then  // Only possible for encoding A1
        MemS[address,4] = PCStoreValue();
    if wback then R[n] = R[n] + 4*BitCount(registers);
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

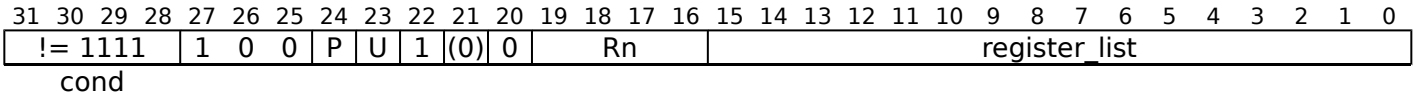
STM (User registers)

In an EL1 mode other than System mode, Store Multiple (User registers) stores multiple User mode registers to consecutive memory locations using an address from a base register. The PE reads the base register value normally, using the current mode to determine the correct Banked version of the register. This instruction cannot writeback to the base register.

Store Multiple (User registers) is UNDEFINED in Hyp mode, and CONSTRAINED UNPREDICTABLE in User or System modes.

Armv8.2 permits the deprecation of some Store Multiple ordering behaviors in AArch32 state, for more information see *FEAT_LSMAOC*.

A1



A1

```
STM{<amode>}{<c>}{<q>} <Rn>, <registers>^
```

```
n = UInt(Rn);  registers = register_list;  increment = (U == '1');  wordhigher = (P == U);
if n == 15 || BitCount(registers) < 1 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction operates as an `STM` with the same addressing mode but targeting an unspecified set of registers. These registers might include R15.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<amode> is one of:

DA
Decrement After. The consecutive memory addresses end at the address in the base register. Encoded as P = 0, U = 0.

ED
Empty Descending. For this instruction, a synonym for DA.

DB
Decrement Before. The consecutive memory addresses end one word below the address in the base register. Encoded as P = 1, U = 0.

FD
Full Descending. For this instruction, a synonym for DB.

IA
Increment After. The consecutive memory addresses start at the address in the base register. This is the default. Encoded as P = 0, U = 1.

EA
Empty Ascending. For this instruction, a synonym for IA.

IB
Increment Before. The consecutive memory addresses start one word above the address in the base register. Encoded as P = 1, U = 1.

FA
Full Ascending. For this instruction, a synonym for IB.

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

<registers> Is a list of one or more registers, separated by commas and surrounded by { and }. It specifies the set of registers to be stored by the STM instruction. The registers are stored with the lowest-numbered register to the lowest memory address, through to the highest-numbered register to the highest memory address. See also *Encoding of lists of general-purpose registers and the PC*.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if PSTATE.EL == EL2 then
        UNDEFINED;
    elsif PSTATE.M IN {M32_User,M32_System} then
        UNPREDICTABLE;
    else
        length = 4*BitCount(registers);
        address = if increment then R[n] else R[n]-length;
        if wordhigher then address = address+4;
        for i = 0 to 14
            if registers<i> == '1' then  // Store User mode register
                MemS[address,4] = Rmode[i, M32_User];
                address = address + 4;
        if registers<15> == '1' then
            MemS[address,4] = PCStoreValue();
```

CONSTRAINED UNPREDICTABLE behavior

If `PSTATE.M IN {M32_User,M32_System}`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction operates as an STM with the same addressing mode but targeting an unspecified set of registers. These registers might include R15.

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

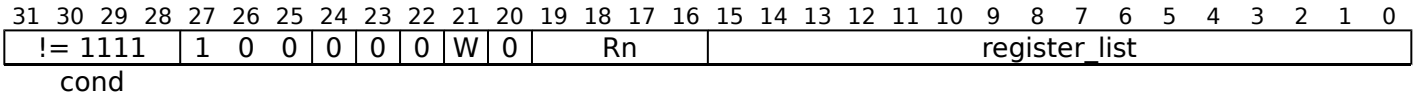
STMDA, STMED

Store Multiple Decrement After (Empty Descending) stores multiple registers to consecutive memory locations using an address from a base register. The consecutive memory locations end at this address, and the address just below the lowest of those locations can optionally be written back to the base register.

The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. See also *Encoding of lists of general-purpose registers and the PC*.

Armv8.2 permits the deprecation of some Store Multiple ordering behaviors in AArch32 state, for more information see *FEAT_LSMAOC*. For details of related system instructions see *STM (User registers)*.

A1



A1

```
STMDA{<c>}{<q>} <Rn>{!}, <registers> // (Preferred syntax)

STMED{<c>}{<q>} <Rn>{!}, <registers> // (Alternate syntax, Empty Descending stack)

n = UInt(Rn);  registers = register_list;  wback = (W == '1');
if n == 15 || BitCount(registers) < 1 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction targets an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers stored.

If `n == 15 && wback`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes without writeback of the base address.
- The instruction uses the addressing mode described in the equivalent immediate offset instruction.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rn>	Is the general-purpose base register, encoded in the "Rn" field.
!	The address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the "W" field as 1, otherwise this field defaults to 0.
<registers>	Is a list of one or more registers to be stored, separated by commas and surrounded by { and }. The PC can be in the list. However, Arm deprecates the use of instructions that include the PC in the list. If base register writeback is specified, and the base register is not the lowest-numbered register in the list, such an instruction stores an UNKNOWN value for the base register.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n] - 4*BitCount(registers) + 4;
    for i = 0 to 14
        if registers<i> == '1' then
            if i == n && wback && i != LowestSetBit(registers) then
                MemS[address,4] = bits(32) UNKNOWN;
            else
                MemS[address,4] = R[i];
            address = address + 4;
    if registers<15> == '1' then
        MemS[address,4] = PCStoreValue();
    if wback then R[n] = R[n] - 4*BitCount(registers);
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STMDB, STMFD

Store Multiple Decrement Before (Full Descending) stores multiple registers to consecutive memory locations using an address from a base register. The consecutive memory locations end just below this address, and the address of the first of those locations can optionally be written back to the base register.

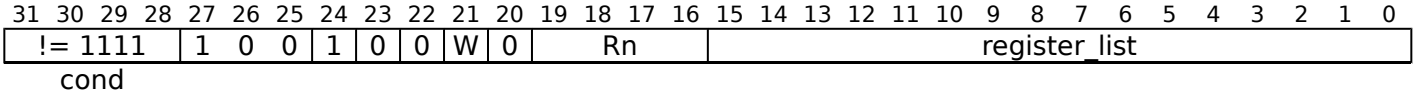
The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. See also *Encoding of lists of general-purpose registers and the PC*.

Armv8.2 permits the deprecation of some Store Multiple ordering behaviors in AArch32 state, for more information see *FEAT_LSMAOC*. For details of related system instructions see *STM (User registers)*.

This instruction is used by the alias PUSH (multiple registers).

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



A1

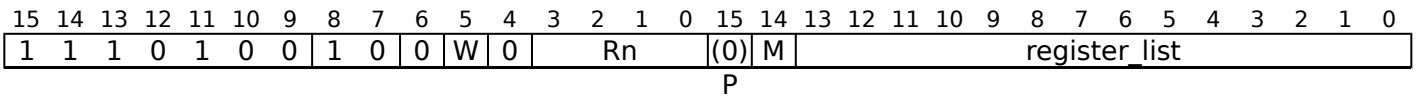
```
STMDB{<c>}{<q>} <Rn>{!}, <registers> // (Preferred syntax)

STMFD{<c>}{<q>} <Rn>{!}, <registers> // (Alternate syntax, Full Descending stack)

n = UInt(Rn);  registers = register_list;  wback = (W == '1');
if n == 15 || BitCount(registers) < 1 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction operates as an `STM` with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers stored.

T1



T1

```
STMDB{<c>}{<q>} <Rn>{!}, <registers> // (Preferred syntax)

STMFD{<c>}{<q>} <Rn>{!}, <registers> // (Alternate syntax, Full Descending stack)

n = UInt(Rn);  registers = P:M:register_list;  wback = (W == '1');
if n == 15 || BitCount(registers) < 2 then UNPREDICTABLE;
if wback && registers<n> == '1' then UNPREDICTABLE;
if registers<13> == '1' then UNPREDICTABLE;
if registers<15> == '1' then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.

- The instruction executes as `NOP`.
- The instruction operates as an `STM` with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers stored.

If `wback && registers<n> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction executes but the value stored for the base register is UNKNOWN.

If `BitCount(registers) == 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes as described, with no change to its behavior and no additional side effects.
- The instruction operates as an `STM` with the same addressing mode but targeting an unspecified set of registers. These registers might include R15.

If `registers<13> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes as described, with no change to its behavior and no additional side effects.
- The store instruction performs all of the stores using the specified addressing mode but the value of R13 is UNKNOWN.

If `registers<15> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction performs all of the stores using the specified addressing mode but the value of R15 is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rn>	Is the general-purpose base register, encoded in the "Rn" field.
!	The address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the "W" field as 1, otherwise this field defaults to 0.
<registers>	For encoding A1: is a list of one or more registers to be stored, separated by commas and surrounded by { and }. The PC can be in the list. However, Arm deprecates the use of instructions that include the PC in the list. If base register writeback is specified, and the base register is not the lowest-numbered register in the list, such an instruction stores an UNKNOWN value for the base register. For encoding T1: is a list of one or more registers to be stored, separated by commas and surrounded by { and }. The registers in the list must be in the range R0-R12, encoded in the "register_list" field, and can optionally contain the LR. If the LR is in the list, the "M" field is set to 1, otherwise it defaults to 0.

Alias Conditions

Alias	Of variant	Is preferred when
PUSH (multiple registers)	T1	`W == '1' && Rn == '1101' && BitCount(M:register_list) > 1`
PUSH (multiple registers)	A1	`W == '1' && Rn == '1101' && BitCount(register_list) > 1`

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n] - 4*BitCount(registers);
    for i = 0 to 14
        if registers<i> == '1' then
            if i == n && wback && i != LowestSetBit(registers) then
                MemS[address,4] = bits(32) UNKNOWN;  // Only possible for encoding A1
            else
                MemS[address,4] = R[i];
            address = address + 4;
    if registers<15> == '1' then  // Only possible for encoding A1
        MemS[address,4] = PCStoreValue();
    if wback then R[n] = R[n] - 4*BitCount(registers);
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

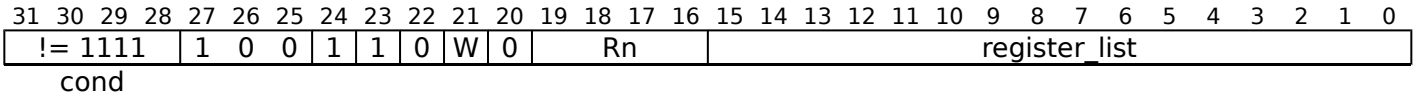
STMIB, STMFA

Store Multiple Increment Before (Full Ascending) stores multiple registers to consecutive memory locations using an address from a base register. The consecutive memory locations start just above this address, and the address of the last of those locations can optionally be written back to the base register.

The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. See also *Encoding of lists of general-purpose registers and the PC*.

Armv8.2 permits the deprecation of some Store Multiple ordering behaviors in AArch32 state, for more information see *FEAT_LSMAOC*. For details of related system instructions see *STM (User registers)*.

A1



A1

```
STMIB{<c>}{<q>} <Rn>{!}, <registers> // (Preferred syntax)

STMFA{<c>}{<q>} <Rn>{!}, <registers> // (Alternate syntax, Full Ascending stack)

n = UInt(Rn);  registers = register_list;  wback = (W == '1');
if n == 15 || BitCount(registers) < 1 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction operates as an `STM` with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers stored.

If `n == 15 && wback`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes without writeback of the base address.
- The instruction uses the addressing mode described in the equivalent immediate offset instruction.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rn>	Is the general-purpose base register, encoded in the "Rn" field.
!	The address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the "W" field as 1, otherwise this field defaults to 0.
<registers>	Is a list of one or more registers to be stored, separated by commas and surrounded by { and }. The PC can be in the list. However, Arm deprecates the use of instructions that include the PC in the list. If base register writeback is specified, and the base register is not the lowest-numbered register in the list, such an instruction stores an UNKNOWN value for the base register.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n] + 4;
    for i = 0 to 14
        if registers<i> == '1' then
            if i == n && wback && i != LowestSetBit(registers) then
                MemS[address,4] = bits(32) UNKNOWN;
            else
                MemS[address,4] = R[i];
            address = address + 4;
    if registers<15> == '1' then
        MemS[address,4] = PCStoreValue();
    if wback then R[n] = R[n] + 4*BitCount(registers);
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

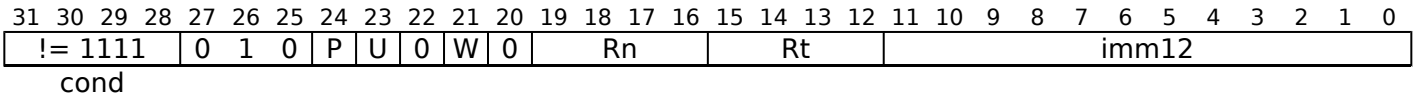
STR (immediate)

Store Register (immediate) calculates an address from a base register value and an immediate offset, and stores a word from a register to memory. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see *Memory accesses*.

This instruction is used by the alias PUSH (single register).

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 , T2 , T3 and T4) .

A1



Offset (P == 1 && W == 0)

```
STR{<c>}{<q>} <Rt>, [<Rn> {, #{+/-}<imm>}]
```

Post-indexed (P == 0 && W == 0)

```
STR{<c>}{<q>} <Rt>, [<Rn>], #{+/-}<imm>
```

Pre-indexed (P == 1 && W == 1)

```
STR{<c>}{<q>} <Rt>, [<Rn>, #{+/-}<imm>]!
```

```
if P == '0' && W == '1' then SEE "STRT";
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
if wback && (n == 15 || n == t) then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

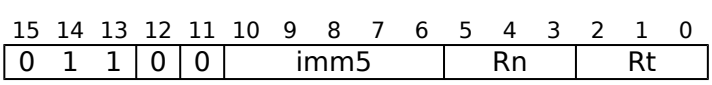
If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If `wback && n == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes without writeback of the base address.
- The instruction uses the addressing mode described in the equivalent immediate offset instruction.

T1

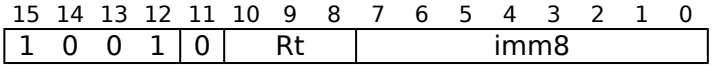


T1

```
STR{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```

```
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm5:'00', 32);
index = TRUE;  add = TRUE;  wback = FALSE;
```

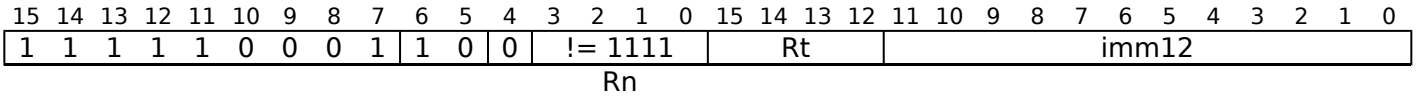
T2



T2

```
STR{<c>}{<q>} <Rt>, [SP{, #{+}<imm>}]

t = UInt(Rt);  n = 13;  imm32 = ZeroExtend(imm8:'00', 32);
index = TRUE;  add = TRUE;  wback = FALSE;
```

T3



T3

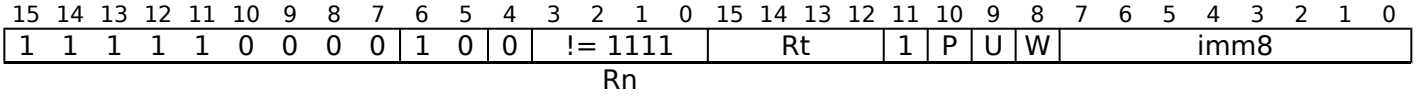
```
STR{<c>}.W <Rt>, [<Rn> {, #{+}<imm>}] // (<Rt>, <Rn>, <imm> can be represented in T1 or T2)

STR{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]

if Rn == '1111' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
index = TRUE;  add = TRUE;  wback = FALSE;
if t == 15 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `t == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

T4



Offset (P == 1 && U == 0 && W == 0)

```
STR{<c>}{<q>} <Rt>, [<Rn> {, #-<imm>}]
```

Post-indexed (P == 0 && W == 1)

```
STR{<c>}{<q>} <Rt>, [<Rn>], #{+/-}<imm>
```

Pre-indexed (P == 1 && W == 1)

```
STR{<c>}{<q>} <Rt>, [<Rn>, #{+/-}<imm>]!

if P == '1' && U == '1' && W == '0' then SEE "STRT";
if Rn == '1111' || (P == '0' && W == '0') then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
if t == 15 || (wback && n == t) then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction executes but the value stored is UNKNOWN.

If `t == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<Rt> For encoding A1: is the general-purpose register to be transferred, encoded in the "Rt" field. The PC can be used, but this is deprecated.

For encoding T1, T2, T3 and T4: is the general-purpose register to be transferred, encoded in the "Rt" field.

<Rn> For encoding A1: is the general-purpose base register, encoded in the "Rn" field. The PC can be used in the offset variant, but this is deprecated.

For encoding T1, T3 and T4: is the general-purpose base register, encoded in the "Rn" field.

+/- Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

\+ Specifies the offset is added to the base register.

<imm> For encoding A1: is the 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 if omitted, and encoded in the "imm12" field.

For encoding T1: is the optional positive unsigned immediate byte offset, a multiple of 4, in the range 0 to 124, defaulting to 0 and encoded in the "imm5" field as <imm>/4.

For encoding T2: is the optional positive unsigned immediate byte offset, a multiple of 4, in the range 0 to 1020, defaulting to 0 and encoded in the "imm8" field as <imm>/4.

For encoding T3: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

For encoding T4: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm8" field.

Alias Conditions

Alias	Of variant	Is preferred when
PUSH (single register)	A1 (pre-indexed)	`P == '1' && U == '0' && W == '1' && Rn == '1101' && imm12 == '000000000100'`
PUSH (single register)	T4 (pre-indexed)	`Rn == '1101' && P == '1' && U == '0' && W == '1' && imm8 == '00000100'`

Operation

```
if CurrentInstrSet() == InstrSet_A32 then
    if ConditionPassed() then
        EncodingSpecificOperations();
        offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
        address = if index then offset_addr else R[n];
        MemU[address,4] = if t == 15 then PCStoreValue() else R[t];
        if wback then R[n] = offset_addr;
else
    if ConditionPassed() then
        EncodingSpecificOperations();
        offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
        address = if index then offset_addr else R[n];
        MemU[address,4] = R[t];
        if wback then R[n] = offset_addr;
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

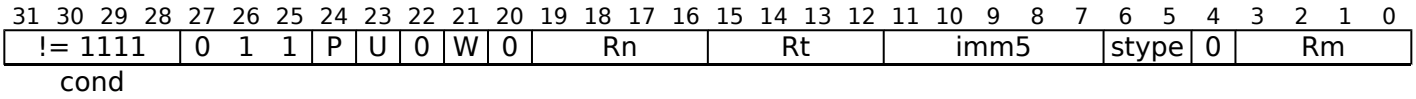
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STR (register)

Store Register (register) calculates an address from a base register value and an offset register value, stores a word from a register to memory. The offset register value can optionally be shifted. For information about memory accesses see *Memory accesses*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2) .

A1



Offset (P == 1 && W == 0)

```
STR{<c>}{<q>} <Rt>, [<Rn>, {+/-}<Rm>{, <shift>}]
```

Post-indexed (P == 0 && W == 0)

```
STR{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>{, <shift>}
```

Pre-indexed (P == 1 && W == 1)

```
STR{<c>}{<q>} <Rt>, [<Rn>, {+/-}<Rm>{, <shift>}]!
```

```
if P == '0' && W == '1' then SEE "STRT";
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm5);
if m == 15 then UNPREDICTABLE;
if wback && (n == 15 || n == t) then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

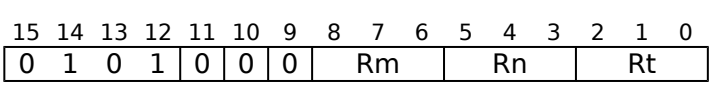
If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If `wback && n == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes without writeback of the base address.
- The instruction uses the addressing mode described in the equivalent immediate offset instruction.

T1

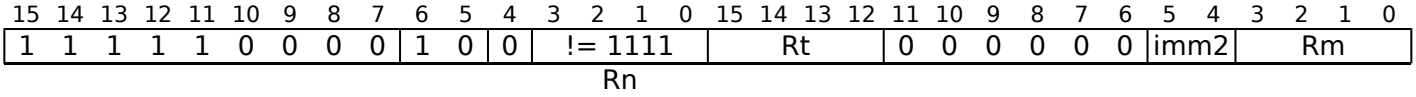


T1

```
STR{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>]
```

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
```

T2



T2

```
STR{<c>}.W <Rt>, [<Rn>, {+}<Rm>] // (<Rt>, <Rn>, <Rm> can be represented in T1)

STR{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>{, LSL #<imm>}]
```

```
if Rn == '1111' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, UInt(imm2));
if t == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

CONSTRAINED UNPREDICTABLE behavior

If `t == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<Rt> For encoding A1: is the general-purpose register to be transferred, encoded in the "Rt" field. The PC can be used, but this is deprecated.

For encoding T1 and T2: is the general-purpose register to be transferred, encoded in the "Rt" field.

<Rn> For encoding A1: is the general-purpose base register, encoded in the "Rn" field. The PC can be used in the offset variant, but this is deprecated.

For encoding T1 and T2: is the general-purpose base register, encoded in the "Rn" field.

+/- Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

\+ Specifies the index register is added to the base register.

<Rm> Is the general-purpose index register, encoded in the "Rm" field.

<shift> The shift to apply to the value read from <Rm>. If absent, no shift is applied. Otherwise, see *Shifts applied to a register*.

<imm> If present, the size of the left shift to apply to the value from <Rm>, in the range 1-3. <imm> is encoded in imm2. If absent, no shift is specified and imm2 is encoded as 0b00.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset = Shift(R[m], shift_t, shift_n, PSTATE.C);
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if index then offset_addr else R[n];
    if t == 15 then  // Only possible for encoding A1
        data = PCStoreValue();
    else
        data = R[t];
    MemU[address,4] = data;
    if wback then R[n] = offset_addr;
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

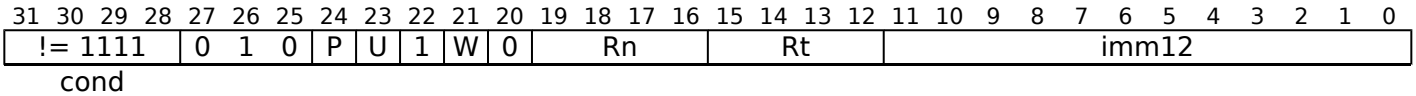
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STRB (immediate)

Store Register Byte (immediate) calculates an address from a base register value and an immediate offset, and stores a byte from a register to memory. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see *Memory accesses*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 , T2 and T3) .

A1



Offset (P == 1 && W == 0)

```
STRB{<c>}{<q>} <Rt>, [<Rn> {, #{+/-}<imm>}]
```

Post-indexed (P == 0 && W == 0)

```
STRB{<c>}{<q>} <Rt>, [<Rn>], #{+/-}<imm>
```

Pre-indexed (P == 1 && W == 1)

```
STRB{<c>}{<q>} <Rt>, [<Rn>, #{+/-}<imm>]!
```

```
if P == '0' && W == '1' then SEE "STRBT";
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
if t == 15 then UNPREDICTABLE;
if wback && (n == 15 || n == t) then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `t == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

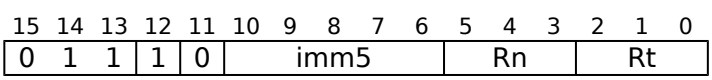
If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If `wback && n == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes without writeback of the base address.
- The instruction uses the addressing mode described in the equivalent immediate offset instruction.

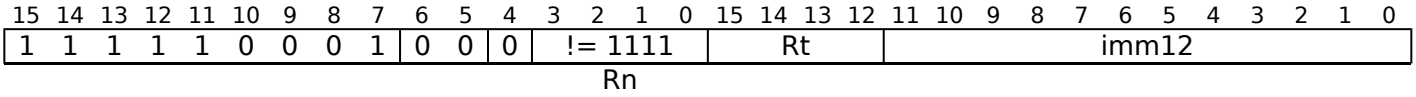
T1



T1

```
STRB{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]

t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm5, 32);
index = TRUE;  add = TRUE;  wback = FALSE;
```

T2



T2

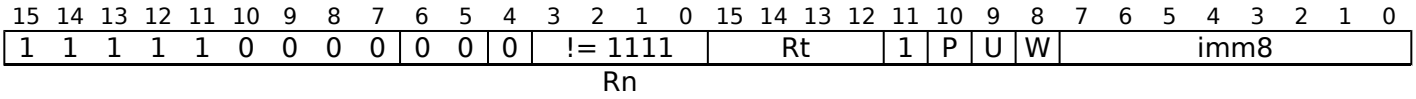
```
STRB{<c>}.W <Rt>, [<Rn> {, #{+}<imm>}] // (<Rt>, <Rn>, <imm> can be represented in T1)

STRB{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]

if Rn == '1111' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
index = TRUE;  add = TRUE;  wback = FALSE;
if t == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

CONSTRAINED UNPREDICTABLE behavior

If `t == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

T3



Offset (P == 1 && U == 0 && W == 0)

```
STRB{<c>}{<q>} <Rt>, [<Rn> {, #-<imm>}]
```

Post-indexed (P == 0 && W == 1)

```
STRB{<c>}{<q>} <Rt>, [<Rn>], #{+/-}<imm>
```

Pre-indexed (P == 1 && W == 1)

```
STRB{<c>}{<q>} <Rt>, [<Rn>, #{+/-}<imm>]!

if P == '1' && U == '1' && W == '0' then SEE "STRBT";
if Rn == '1111' || (P == '0' && W == '0') then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
if t == 15 || (wback && n == t) then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

CONSTRAINED UNPREDICTABLE behavior

If `t == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction executes but the value stored is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<Rt> Is the general-purpose register to be transferred, encoded in the "Rt" field.

<Rn> For encoding A1: is the general-purpose base register, encoded in the "Rn" field. The PC can be used in the offset variant, but this is deprecated.

For encoding T1, T2 and T3: is the general-purpose base register, encoded in the "Rn" field.

+/- Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

\+ Specifies the offset is added to the base register.

<imm> For encoding A1: is the 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 if omitted, and encoded in the "imm12" field.

For encoding T1: is an optional 5-bit unsigned immediate byte offset, in the range 0 to 31, defaulting to 0 and encoded in the "imm5" field.

For encoding T2: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

For encoding T3: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm8" field.

Operation

```
if CurrentInstrSet() == InstrSet_A32 then
    if ConditionPassed() then
        EncodingSpecificOperations();
        offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
        address = if index then offset_addr else R[n];
        MemU[address,1] = R[t]<7:0>;
        if wback then R[n] = offset_addr;
else
    if ConditionPassed() then
        EncodingSpecificOperations();
        offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
        address = if index then offset_addr else R[n];
        MemU[address,1] = R[t]<7:0>;
        if wback then R[n] = offset_addr;
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

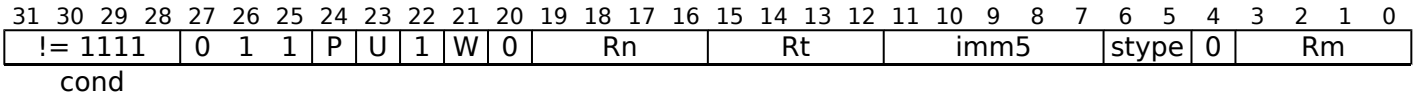
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STRB (register)

Store Register Byte (register) calculates an address from a base register value and an offset register value, and stores a byte from a register to memory. The offset register value can optionally be shifted. For information about memory accesses see *Memory accesses*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2) .

A1



Offset (P == 1 && W == 0)

```
STRB{<c>}{<q>} <Rt>, [<Rn>, {+/-}<Rm>{, <shift>}]
```

Post-indexed (P == 0 && W == 0)

```
STRB{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>{, <shift>}
```

Pre-indexed (P == 1 && W == 1)

```
STRB{<c>}{<q>} <Rt>, [<Rn>, {+/-}<Rm>{, <shift>}]!
```

```
if P == '0' && W == '1' then SEE "STRBT";
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm5);
if t == 15 || m == 15 then UNPREDICTABLE;
if wback && (n == 15 || n == t) then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `t == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

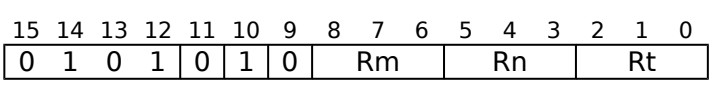
If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If `wback && n == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes without writeback of the base address.
- The instruction uses the addressing mode described in the equivalent immediate offset instruction.

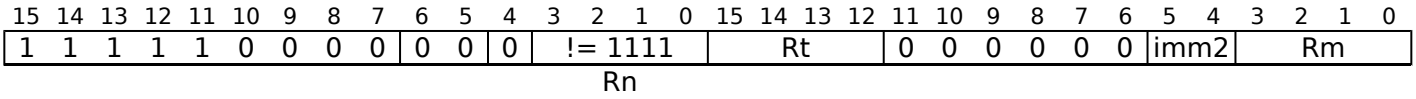
T1



T1

```
STRB{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>]

t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
```

T2



T2

```
STRB{<c>}.W <Rt>, [<Rn>, {+}<Rm>] // (<Rt>, <Rn>, <Rm> can be represented in T1)

STRB{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>{, LSL #<imm>}]

if Rn == '1111' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, UInt(imm2));
if t == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

CONSTRAINED UNPREDICTABLE behavior

If `t == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rt>	Is the general-purpose register to be transferred, encoded in the "Rt" field.
<Rn>	For encoding A1: is the general-purpose base register, encoded in the "Rn" field. The PC can be used in the offset variant, but this is deprecated. For encoding T1 and T2: is the general-purpose base register, encoded in the "Rn" field.
+/-	Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

+	Specifies the index register is added to the base register.
<Rm>	Is the general-purpose index register, encoded in the "Rm" field.
<shift>	The shift to apply to the value read from <Rm>. If absent, no shift is applied. Otherwise, see *Shifts applied to a register*.
<imm>	If present, the size of the left shift to apply to the value from <Rm>, in the range 1-3. <imm> is encoded in imm2. If absent, no shift is specified and imm2 is encoded as 0b00.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset = Shift(R[m], shift_t, shift_n, PSTATE.C);
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if index then offset_addr else R[n];
    MemU[address,1] = R[t]<7:0>;
    if wback then R[n] = offset_addr;
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STRBT

Store Register Byte Unprivileged stores a byte from a register to memory. For information about memory accesses see *Memory accesses*.

The memory access is restricted as if the PE were running in User mode. This makes no difference if the PE is actually running in User mode.

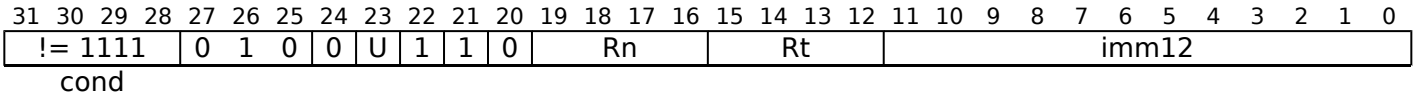
STRBT is UNPREDICTABLE in Hyp mode.

The T32 instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an immediate offset, and leaves the base register unchanged.

The A32 instruction uses a post-indexed addressing mode, that uses a base register value as the address for the memory access, and calculates a new address from a base register value and an offset and writes it back to the base register. The offset can be an immediate value or an optionally-shifted register value.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1) .

A1



A1

```
STRBT{<c>}{<q>} <Rt>, [<Rn>] {, #{+/-}<imm>}

t = UInt(Rt);  n = UInt(Rn);  postindex = TRUE;  add = (U == '1');
register_form = FALSE;  imm32 = ZeroExtend(imm12, 32);
if t == 15 || n == 15 || n == t then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `t == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

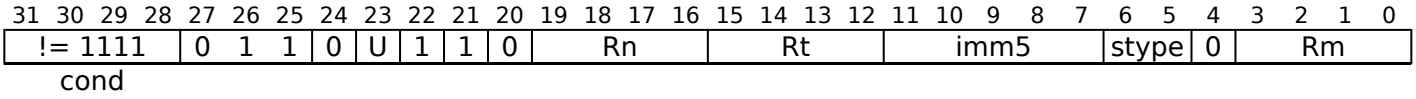
If `n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction executes but the value stored is UNKNOWN.

If `n == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction uses post-indexed addressing with the base register as PC. This is handled as described in *Using R15*.
- The instruction is treated as if bit[24] == 1 and bit[21] == 0. The instruction uses immediate offset addressing with the base register as PC, without writeback.

A2



A2

```
STRBT{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>{, <shift>}
```

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);  postindex = TRUE;  add = (U == '1');
register_form = TRUE;  (shift_t, shift_n) = DecodeImmShift(stype, imm5);
if t == 15 || n == 15 || n == t || m == 15 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `t == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

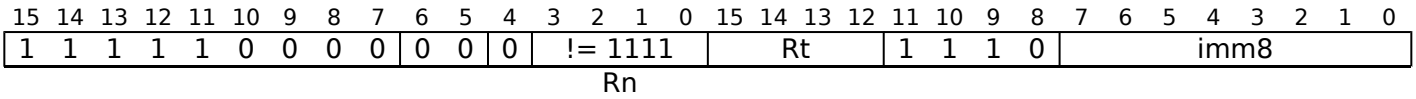
If `n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction executes but the value stored is UNKNOWN.

If `n == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction uses post-indexed addressing with the base register as PC. This is handled as described in *Using R15*.
- The instruction is treated as if bit[24] == 1 and bit[21] == 0. The instruction uses immediate offset addressing with the base register as PC, without writeback.

T1



T1

```
STRBT{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```

```
if Rn == '1111' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  postindex = FALSE;  add = TRUE;
register_form = FALSE;  imm32 = ZeroExtend(imm8, 32);
if t == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

CONSTRAINED UNPREDICTABLE behavior

If `t == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rt>	For encoding A1: is the general-purpose register to be transferred, encoded in the "Rt" field. The PC can be used, but this is deprecated.

For encoding A2 and T1: is the general-purpose register to be transferred, encoded in the "Rt" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

+/- For encoding A1: specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

For encoding A2: specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<Rm> Is the general-purpose index register, encoded in the "Rm" field.

<shift> The shift to apply to the value read from <Rm>. If absent, no shift is applied. Otherwise, see *Shifts applied to a register*.

+ Specifies the offset is added to the base register.

<imm> For encoding A1: is the 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 if omitted, and encoded in the "imm12" field.

For encoding T1: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field.

Operation

```
if ConditionPassed() then
    if PSTATE.EL == EL2 then UNPREDICTABLE;               // Hyp mode
    EncodingSpecificOperations();
    offset = if register_form then Shift(R[m], shift_t, shift_n, PSTATE.C) else imm32;
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if postindex then R[n] else offset_addr;
    MemU_unpriv[address,1] = R[t]<7:0>;
    if postindex then R[n] = offset_addr;
```

CONSTRAINED UNPREDICTABLE behavior

If `PSTATE.EL == EL2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes as `STRB` (immediate).

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

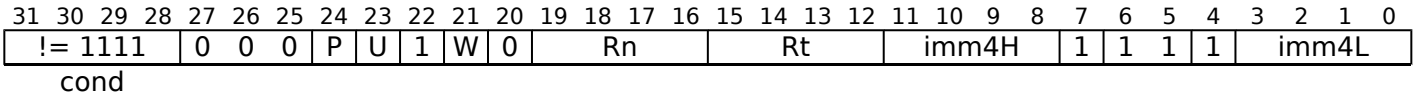
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STRD (immediate)

Store Register Dual (immediate) calculates an address from a base register value and an immediate offset, and stores two words from two registers to memory. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see *Memory accesses*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



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.

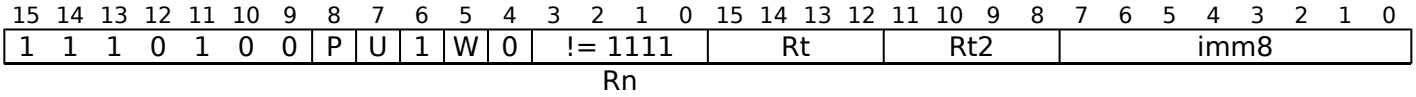
If `Rt<0> == '1'`, then one of the following behaviors must occur:

- 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



Offset (P == 1 && W == 0)

```
STRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn> {, #{+/-}<imm>}]
```

Post-indexed (P == 0 && W == 1)

```
STRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>], #{+/-}<imm>
```

Pre-indexed (P == 1 && W == 1)

```
STRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>, #{+/-}<imm>]!
```

```
if P == '0' && W == '0' then SEE "Related encodings";
t = UInt(Rt);  t2 = UInt(Rt2);  n = UInt(Rn);  imm32 = ZeroExtend(imm8:'00', 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
if wback && (n == t || n == t2) then UNPREDICTABLE;
if n == 15 || t == 15 || t2 == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

CONSTRAINED UNPREDICTABLE behavior

If `t == 15 || t2 == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

If `wback && (n == t || n == t2)`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction executes but the value stored is UNKNOWN.

If `wback && n == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes without writeback of the base address.
- The instruction uses the addressing mode described in the equivalent immediate offset instruction.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Related encodings: *Load/store dual, load/store exclusive, table branch*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rt>	For encoding A1: is the first general-purpose register to be transferred, encoded in the "Rt" field. This register must be even-numbered and not R14. For encoding T1: is the first general-purpose register to be transferred, encoded in the "Rt" field.

<Rt2> For encoding A1: is the second general-purpose register to be transferred. This register must be <R(t+1)>.

For encoding T1: is the second general-purpose register to be transferred, encoded in the "Rt2" field.

<Rn> For encoding A1: is the general-purpose base register, encoded in the "Rn" field. The PC can be used in the offset variant, but this is deprecated.

For encoding T1: is the general-purpose base register, encoded in the "Rn" field.

+/- Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<imm> For encoding A1: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm4H:imm4L" field.

For encoding T1: is the unsigned immediate byte offset, a multiple of 4, in the range 0 to 1020, defaulting to 0 if omitted, and encoded in the "imm8" field as <imm>/4.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    if address == Align(address, 8) then
        bits(64) data;
        if BigEndian(AccType_ATOMIC) then
            data<63:32> = R[t];
            data<31:0> = R[t2];
        else
            data<31:0> = R[t];
            data<63:32> = R[t2];
        MemA[address,8] = data;
    else
        MemA[address,4] = R[t];
        MemA[address+4,4] = R[t2];
    if wback then R[n] = offset_addr;
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

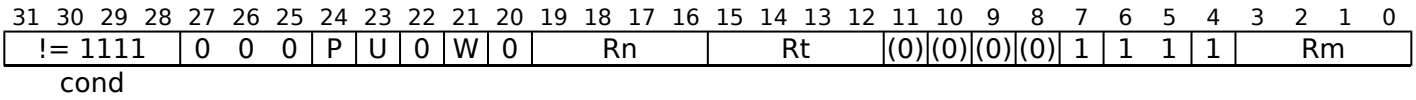
Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STRD (register)

Store Register Dual (register) calculates an address from a base register value and a register offset, and stores two words from two registers to memory. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see *Memory accesses*.

A1



Offset (P == 1 && W == 0)

```
STRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>, {+/-}<Rm>]
```

Post-indexed (P == 0 && W == 0)

```
STRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>], {+/-}<Rm>
```

Pre-indexed (P == 1 && W == 1)

```
STRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>, {+/-}<Rm>]!
```

```
if Rt<0> == '1' then UNPREDICTABLE;
t = UInt(Rt);  t2 = t+1;  n = UInt(Rn);  m = UInt(Rm);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
if P == '0' && W == '1' then UNPREDICTABLE;
if t2 == 15 || m == 15 then UNPREDICTABLE;
if wback && (n == 15 || n == t || n == t2) then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `t == 15 || t2 == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

If `wback && (n == t || n == t2)`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If `wback && n == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes without writeback of the base address.
- The instruction uses the addressing mode described in the equivalent immediate offset instruction.

If `Rt<0> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: t<0> = '0'.
- The instruction executes with the additional decode: t2 = t.
- The instruction executes as described, with no change to its behavior and no additional side-effects. This does not apply when Rt == '1111'.

If `P == '0' && W == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: P = '1'; W = '0'.
- The instruction executes with the additional decode: P = '1'; W = '1'.
- The instruction executes with the additional decode: P = '0'; W = '0'.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<Rt> Is the first general-purpose register to be transferred, encoded in the "Rt" field. This register must be even-numbered and not R14.

<Rt2> Is the second general-purpose register to be transferred. This register must be <R(t+1)>.

<Rn> Is the general-purpose base register, encoded in the "Rn" field. The PC can be used in the offset variant, but this is deprecated.

+/- Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<Rm> Is the general-purpose index register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + R[m]) else (R[n] - R[m]);
    address = if index then offset_addr else R[n];
    if address == Align(address, 8) then
        bits(64) data;
        if BigEndian(AccType_ATOMIC) then
            data<63:32> = R[t];
            data<31:0> = R[t2];
        else
            data<31:0> = R[t];
            data<63:32> = R[t2];
        MemA[address,8] = data;
    else
        MemA[address,4] = R[t];
        MemA[address+4,4] = R[t2];
    if wback then R[n] = offset_addr;
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

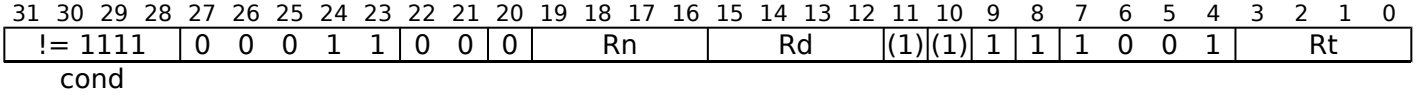
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STREX

Store Register Exclusive calculates an address from a base register value and an immediate offset, stores a word from a register to the calculated address if the PE has exclusive access to the memory at that address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed.
For more information about support for shared memory see *Synchronization and semaphores*. For information about memory accesses see *Memory accesses*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



A1

```
STREX{<c>}{<q>} <Rd>, <Rt>, [<Rn> {, {#}<imm>}]

d = UInt(Rd);  t = UInt(Rt);  n = UInt(Rn);  imm32 = Zeros(32); // Zero offset
if d == 15 || t == 15 || n == 15 then UNPREDICTABLE;
if d == n || d == t then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

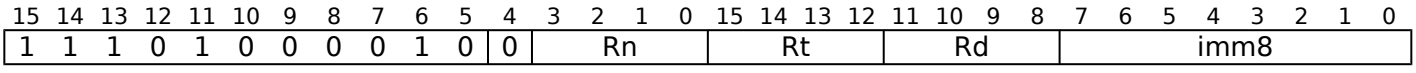
If `d == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction executes but the value stored is UNKNOWN.

If `d == n`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs the store to an UNKNOWN address.

T1



T1

```
STREX{<c>}{<q>} <Rd>, <Rt>, [<Rn> {, #<imm>}]

d = UInt(Rd);  t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm8:'00', 32);
if d == 15 || t == 15 || n == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
if d == n || d == t then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction executes but the value stored is UNKNOWN.

If `d == n`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.

- The instruction performs the store to an UNKNOWN address.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<Rd> Is the destination general-purpose register into which the status result of the store exclusive is written, encoded in the "Rd" field. The value returned is:

0
If the operation updates memory.

1
If the operation fails to update memory.

<Rt> Is the general-purpose register to be transferred, encoded in the "Rt" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

<imm> For encoding A1: the immediate offset added to the value of <Rn> to calculate the address. <imm> can only be 0 or omitted.

For encoding T1: the immediate offset added to the value of <Rn> to calculate the address. <imm> can be omitted, meaning an offset of 0. Values are multiples of 4 in the range 0-1020.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Rd> is not updated.

A non word-aligned memory address causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If AArch32.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch32.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n] + imm32;
    if AArch32.ExclusiveMonitorsPass(address,4) then
        MemA[address,4] = R[t];
        R[d] = ZeroExtend('0');
    else
        R[d] = ZeroExtend('1');
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

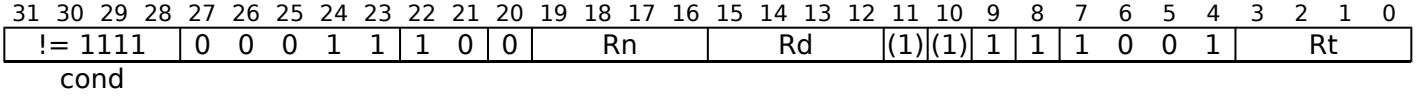
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STREXB

Store Register Exclusive Byte derives an address from a base register value, stores a byte from a register to the derived address if the executing PE has exclusive access to the memory at that address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed.
For more information about support for shared memory see *Synchronization and semaphores*. For information about memory accesses see *Memory accesses*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



A1

```
STREXB{<c>}{<q>} <Rd>, <Rt>, [<Rn>]

d = UInt(Rd);  t = UInt(Rt);  n = UInt(Rn);
if d == 15 || t == 15 || n == 15 then UNPREDICTABLE;
if d == n || d == t then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

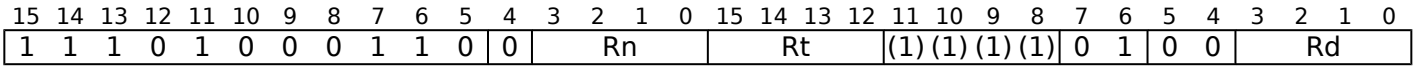
If `d == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction executes but the value stored is UNKNOWN.

If `d == n`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs the store to an UNKNOWN address.

T1



T1

```
STREXB{<c>}{<q>} <Rd>, <Rt>, [<Rn>]

d = UInt(Rd);  t = UInt(Rt);  n = UInt(Rn);
if d == 15 || t == 15 || n == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
if d == n || d == t then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction executes but the value stored is UNKNOWN.

If `d == n`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.

- The instruction performs the store to an UNKNOWN address.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<Rd> Is the destination general-purpose register into which the status result of the store exclusive is written, encoded in the "Rd" field. The value returned is:

0
If the operation updates memory.

1
If the operation fails to update memory.

<Rt> Is the general-purpose register to be transferred, encoded in the "Rt" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

Aborts

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Rd> is not updated.

If AArch32.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    if AArch32.ExclusiveMonitorsPass(address,1) then
        MemA[address,1] = R[t]<7:0>;
        R[d] = ZeroExtend('0');
    else
        R[d] = ZeroExtend('1');
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

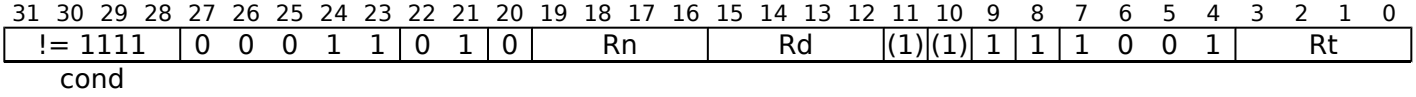
STREXD

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



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.

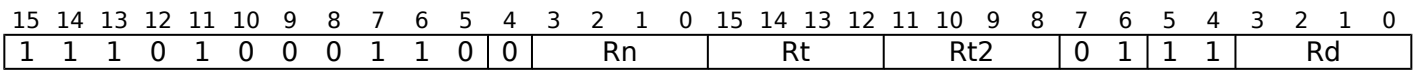
If `Rt<0> == '1'`, then one of the following behaviors must occur:

- 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



T1

```
STREXD{<c>}{<q>} <Rd>, <Rt>, <Rt2>, [<Rn>]
```

```
d = UInt(Rd);  t = UInt(Rt);  t2 = UInt(Rt2);  n = UInt(Rn);
if d == 15 || t == 15 || t2 == 15 || n == 15 then UNPREDICTABLE;
// Armv8-A removes UNPREDICTABLE for R13
if d == n || d == t || d == t2 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction executes but the value stored is UNKNOWN.

If `d == n`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs the store to an UNKNOWN address.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<Rd> Is the destination general-purpose register into which the status result of the store exclusive is written, encoded in the "Rd" field. The value returned is:

0
If the operation updates memory.

1
If the operation fails to update memory.

<Rd> must not be the same as <Rn>, <Rt>, or <Rt2>.

<Rt> For encoding A1: is the first general-purpose register to be transferred, encoded in the "Rt" field. <Rt> must be even-numbered and not R14.

For encoding T1: is the first general-purpose register to be transferred, encoded in the "Rt" field.

<Rt2> For encoding A1: is the second general-purpose register to be transferred. <Rt2> must be <R(t+1)>.

For encoding T1: is the second general-purpose register to be transferred, encoded in the "Rt2" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Rd> is not updated.

A non doubleword-aligned memory address causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If AArch32.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch32.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    // Create doubleword to store such that R[t] will be stored at address and R[t2] at address+4.
    value = if BigEndian(AccType_ATOMIC) then R[t]:R[t2] else R[t2]:R[t];
    if AArch32.ExclusiveMonitorsPass(address,8) then
        MemA[address,8] = value;  R[d] = ZeroExtend('0');
    else
        R[d] = ZeroExtend('1');
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

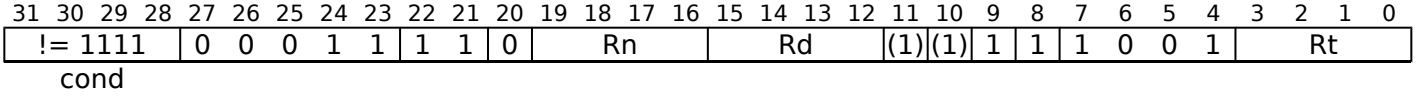
STREXH

Store Register Exclusive Halfword derives an address from a base register value, stores a halfword from a register to the derived address if the executing PE has exclusive access to the memory at that address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed.

For more information about support for shared memory see *Synchronization and semaphores*. For information about memory accesses see *Memory accesses*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



A1

```
STREXH{<c>}{<q>} <Rd>, <Rt>, [<Rn>]

d = UInt(Rd);  t = UInt(Rt);  n = UInt(Rn);
if d == 15 || t == 15 || n == 15 then UNPREDICTABLE;
if d == n || d == t then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

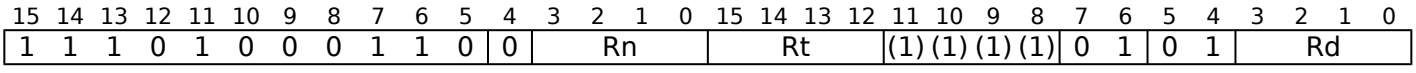
If `d == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction executes but the value stored is UNKNOWN.

If `d == n`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs the store to an UNKNOWN address.

T1



T1

```
STREXH{<c>}{<q>} <Rd>, <Rt>, [<Rn>]

d = UInt(Rd);  t = UInt(Rt);  n = UInt(Rn);
if d == 15 || t == 15 || n == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
if d == n || d == t then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction executes but the value stored is UNKNOWN.

If `d == n`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.

- The instruction performs the store to an UNKNOWN address.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<Rd> Is the destination general-purpose register into which the status result of the store exclusive is written, encoded in the "Rd" field. The value returned is:

0
If the operation updates memory.

1
If the operation fails to update memory.

<Rt> Is the general-purpose register to be transferred, encoded in the "Rt" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Rd> is not updated.

A non halfword-aligned memory address causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If AArch32.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch32.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    if AArch32.ExclusiveMonitorsPass(address,2) then
        MemA[address,2] = R[t]<15:0>;
        R[d] = ZeroExtend('0');
    else
        R[d] = ZeroExtend('1');
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

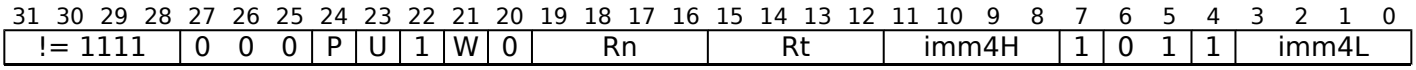
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STRH (immediate)

Store Register Halfword (immediate) calculates an address from a base register value and an immediate offset, and stores a halfword from a register to memory. It can use offset, post-indexed, or pre-indexed addressing. For information about memory accesses see *Memory accesses*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 , T2 and T3) .

A1



Offset (P == 1 && W == 0)

```
STRH{<c>}{<q>} <Rt>, [<Rn> {, #{+/-}<imm>}]
```

Post-indexed (P == 0 && W == 0)

```
STRH{<c>}{<q>} <Rt>, [<Rn>], #{+/-}<imm>
```

Pre-indexed (P == 1 && W == 1)

```
STRH{<c>}{<q>} <Rt>, [<Rn>, #{+/-}<imm>]!
```

```
if P == '0' && W == '1' then SEE "STRHT";
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm4H:imm4L, 32);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
if t == 15 then UNPREDICTABLE;
if wback && (n == 15 || n == t) then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `t == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

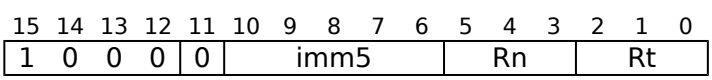
If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If `wback && n == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes without writeback of the base address.
- The instruction uses the addressing mode described in the equivalent immediate offset instruction.

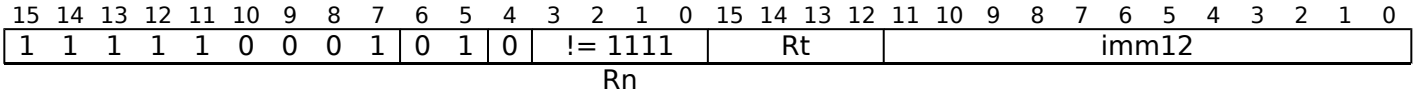
T1



T1

```
STRH{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]

t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm5:'0', 32);
index = TRUE;  add = TRUE;  wback = FALSE;
```

T2



T2

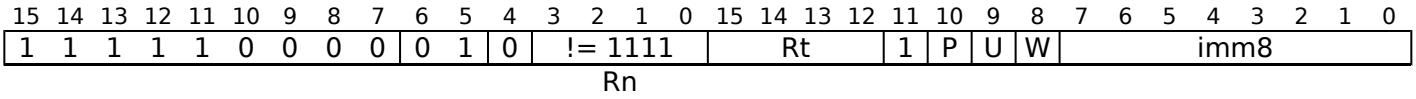
```
STRH{<c>}.W <Rt>, [<Rn> {, #{+}<imm>}] // (<Rt>, <Rn>, <imm> can be represented in T1)

STRH{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]

if Rn == '1111' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
index = TRUE;  add = TRUE;  wback = FALSE;
if t == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

CONSTRAINED UNPREDICTABLE behavior

If `t == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

T3



Offset (P == 1 && U == 0 && W == 0)

```
STRH{<c>}{<q>} <Rt>, [<Rn> {, #-<imm>}]
```

Post-indexed (P == 0 && W == 1)

```
STRH{<c>}{<q>} <Rt>, [<Rn>], #{+/-}<imm>
```

Pre-indexed (P == 1 && W == 1)

```
STRH{<c>}{<q>} <Rt>, [<Rn>, #{+/-}<imm>]!

if P == '1' && U == '1' && W == '0' then SEE "STRHT";
if Rn == '1111' || (P == '0' && W == '0') then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
if t == 15 || (wback && n == t) then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

CONSTRAINED UNPREDICTABLE behavior

If `t == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction executes but the value stored is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<Rt> Is the general-purpose register to be transferred, encoded in the "Rt" field.

<Rn> For encoding A1: is the general-purpose base register, encoded in the "Rn" field. The PC can be used in the offset variant, but this is deprecated.

For encoding A1, T1, T2, T3: is the general-purpose base register, encoded in the "Rn" field.

+/- Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

\+ Specifies the offset is added to the base register.

<imm> For encoding A1: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm4H:imm4L" field.

For encoding T1: is the optional positive unsigned immediate byte offset, a multiple of 2, in the range 0 to 62, defaulting to 0 and encoded in the "imm5" field as <imm>/2.

For encoding T2: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

For encoding T3: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm8" field.

Operation

```
if CurrentInstrSet() == InstrSet_A32 then
    if ConditionPassed() then
        EncodingSpecificOperations();
        offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
        address = if index then offset_addr else R[n];
        MemU[address,2] = R[t]<15:0>;
        if wback then R[n] = offset_addr;
else
    if ConditionPassed() then
        EncodingSpecificOperations();
        offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
        address = if index then offset_addr else R[n];
        MemU[address,2] = R[t]<15:0>;
        if wback then R[n] = offset_addr;
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

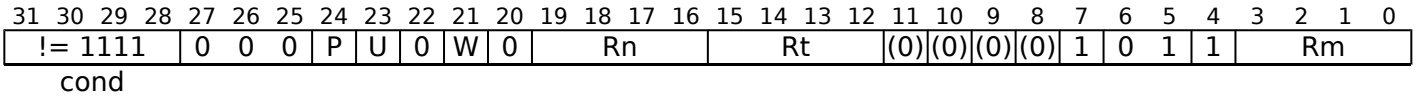
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STRH (register)

Store Register Halfword (register) calculates an address from a base register value and an offset register value, and stores a halfword from a register to memory. The offset register value can be shifted left by 0, 1, 2, or 3 bits. For information about memory accesses see *Memory accesses*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2) .

A1



Offset (P == 1 && W == 0)

```
STRH{<c>}{<q>} <Rt>, [<Rn>, {+/-}<Rm>]
```

Post-indexed (P == 0 && W == 0)

```
STRH{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>
```

Pre-indexed (P == 1 && W == 1)

```
STRH{<c>}{<q>} <Rt>, [<Rn>, {+/-}<Rm>]!
```

```
if P == '0' && W == '1' then SEE "STRHT";
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
(shift_t, shift_n) = (SRType_LSL, 0);
if t == 15 || m == 15 then UNPREDICTABLE;
if wback && (n == 15 || n == t) then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `t == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

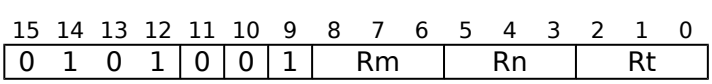
If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If `wback && n == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes without writeback of the base address.
- The instruction uses the addressing mode described in the equivalent immediate offset instruction.

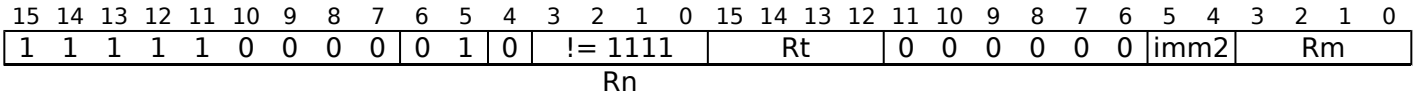
T1



T1

```
STRH{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>]

t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
```

T2



T2

```
STRH{<c>}.W <Rt>, [<Rn>, {+}<Rm>] // (<Rt>, <Rn>, <Rm> can be represented in T1)

STRH{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>{, LSL #<imm>}]

if Rn == '1111' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, UInt(imm2));
if t == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

CONSTRAINED UNPREDICTABLE behavior

If `t == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rt>	Is the general-purpose register to be transferred, encoded in the "Rt" field.
<Rn>	For encoding A1: is the general-purpose base register, encoded in the "Rn" field. The PC can be used in the offset variant, but this is deprecated. For encoding T1 and T2: is the general-purpose base register, encoded in the "Rn" field.
+/-	Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

+	Specifies the index register is added to the base register.
<Rm>	Is the general-purpose index register, encoded in the "Rm" field.
<imm>	If present, the size of the left shift to apply to the value from <Rm>, in the range 1-3. <imm> is encoded in imm2. If absent, no shift is specified and imm2 is encoded as 0b00.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset = Shift(R[m], shift_t, shift_n, PSTATE.C);
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if index then offset_addr else R[n];
    MemU[address,2] = R[t]<15:0>;
    if wback then R[n] = offset_addr;
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STRHT

Store Register Halfword Unprivileged stores a halfword from a register to memory. For information about memory accesses see *Memory accesses*.

The memory access is restricted as if the PE were running in User mode. This makes no difference if the PE is actually running in User mode.

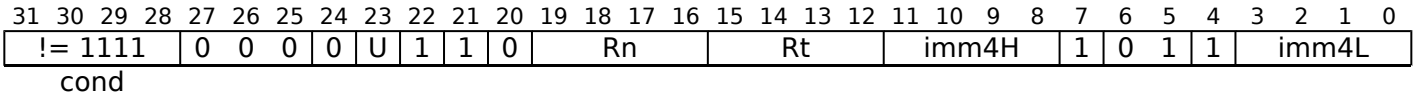
STRHT is UNPREDICTABLE in Hyp mode.

The T32 instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an immediate offset, and leaves the base register unchanged.

The A32 instruction uses a post-indexed addressing mode, that uses a base register value as the address for the memory access, and calculates a new address from a base register value and an offset and writes it back to the base register. The offset can be an immediate value or a register value.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1) .

A1



A1

```
STRHT{<c>}{<q>} <Rt>, [<Rn>] {, #{+/-}<imm>}

t = UInt(Rt);  n = UInt(Rn);  postindex = TRUE;  add = (U == '1');
register_form = FALSE;  imm32 = ZeroExtend(imm4H:imm4L, 32);
if t == 15 || n == 15 || n == t then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `t == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

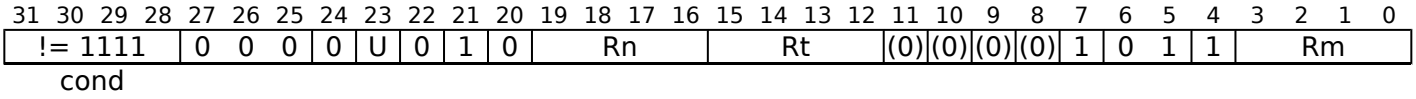
If `n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction executes but the value stored is UNKNOWN.

If `n == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction uses post-indexed addressing with the base register as PC. This is handled as described in *Using R15*.
- The instruction is treated as if bit[24] == 1 and bit[21] == 0. The instruction uses immediate offset addressing with the base register as PC, without writeback.

A2



A2

```
STRHT{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>
```

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);  postindex = TRUE;  add = (U == '1');
register_form = TRUE;
if t == 15 || n == 15 || n == t || m == 15 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `t == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

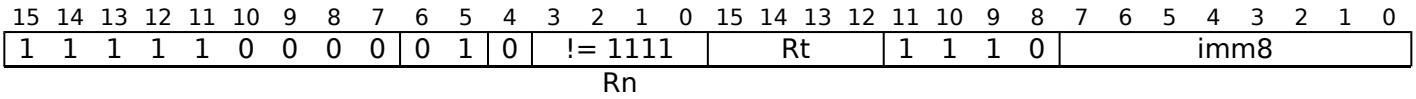
If `n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction executes but the value stored is UNKNOWN.

If `n == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction uses post-indexed addressing with the base register as PC. This is handled as described in *Using R15*.
- The instruction is treated as if bit[24] == 1 and bit[21] == 0. The instruction uses immediate offset addressing with the base register as PC, without writeback.

T1



T1

```
STRHT{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```

```
if Rn == '1111' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  postindex = FALSE;  add = TRUE;
register_form = FALSE;  imm32 = ZeroExtend(imm8, 32);
if t == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

CONSTRAINED UNPREDICTABLE behavior

If `t == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rt>	Is the general-purpose register to be transferred, encoded in the "Rt" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

+/- For encoding A1: specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

For encoding A2: specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<Rm> Is the general-purpose index register, encoded in the "Rm" field.

\+ Specifies the offset is added to the base register.

<imm> For encoding A1: is the 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 if omitted, and encoded in the "imm4H:imm4L" field.

For encoding T1: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field.

Operation

```
if ConditionPassed() then
    if PSTATE.EL == EL2 then UNPREDICTABLE;               // Hyp mode
    EncodingSpecificOperations();
    offset = if register_form then R[m] else imm32;
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if postindex then R[n] else offset_addr;
    MemU_unpriv[address,2] = R[t]<15:0>;
    if postindex then R[n] = offset_addr;
```

CONSTRAINED UNPREDICTABLE behavior

If `PSTATE.EL == EL2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes as `STRH` (immediate).

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STRT

Store Register Unprivileged stores a word from a register to memory. For information about memory accesses see *Memory accesses*.

The memory access is restricted as if the PE were running in User mode. This makes no difference if the PE is actually running in User mode.

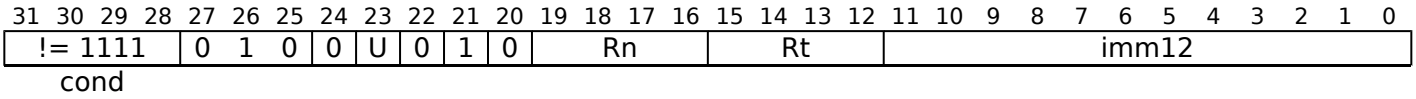
STRT is UNPREDICTABLE in Hyp mode.

The T32 instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an immediate offset, and leaves the base register unchanged.

The A32 instruction uses a post-indexed addressing mode, that uses a base register value as the address for the memory access, and calculates a new address from a base register value and an offset and writes it back to the base register. The offset can be an immediate value or an optionally-shifted register value.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1) .

A1



A1

```
STRT{<c>}{<q>} <Rt>, [<Rn>] {, #{+/-}<imm>}

t = UInt(Rt);  n = UInt(Rn);  postindex = TRUE;  add = (U == '1');
register_form = FALSE;  imm32 = ZeroExtend(imm12, 32);
if n == 15 || n == t then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

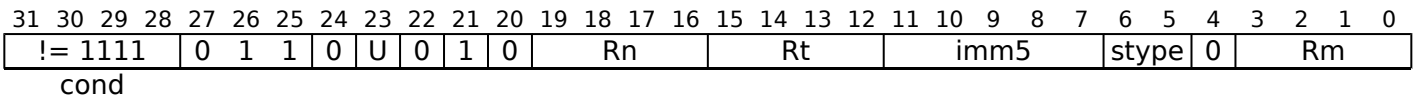
If `n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If `n == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction uses post-indexed addressing with the base register as PC. This is handled as described in *Using R15*.
- The instruction is treated as if bit[24] == 1 and bit[21] == 0. The instruction uses immediate offset addressing with the base register as PC, without writeback.

A2



A2

```
STRT{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>{, <shift>}

t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);  postindex = TRUE;  add = (U == '1');
register_form = TRUE;  (shift_t, shift_n) = DecodeImmShift(stype, imm5);
if n == 15 || n == t || m == 15 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

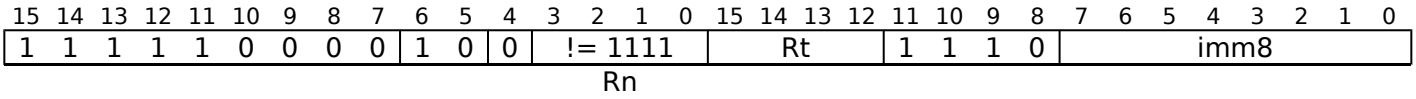
If `n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction executes but the value stored is UNKNOWN.

If `n == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction uses post-indexed addressing with the base register as PC. This is handled as described in *Using R15*.
- The instruction is treated as if bit[24] == 1 and bit[21] == 0. The instruction uses immediate offset addressing with the base register as PC, without writeback.

T1



T1

```
STRT{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```

```
if Rn == '1111' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  postindex = FALSE;  add = TRUE;
register_form = FALSE;  imm32 = ZeroExtend(imm8, 32);
if t == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

CONSTRAINED UNPREDICTABLE behavior

If `t == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<Rt> For encoding A1 and A2: is the general-purpose register to be transferred, encoded in the "Rt" field. The PC can be used, but this is deprecated.

For encoding T1: is the general-purpose register to be transferred, encoded in the "Rt" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

+/- For encoding A1: specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

For encoding A2: specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<Rm> Is the general-purpose index register, encoded in the "Rm" field.

<shift> The shift to apply to the value read from <Rm>. If absent, no shift is applied. Otherwise, see *Shifts applied to a register*.

\+ Specifies the offset is added to the base register.

<imm> For encoding A1: is the 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 if omitted, and encoded in the "imm12" field.

For encoding T1: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field.

Operation

```
if ConditionPassed() then
    if PSTATE.EL == EL2 then UNPREDICTABLE;               // Hyp mode
    EncodingSpecificOperations();
    offset = if register_form then Shift(R[m], shift_t, shift_n, PSTATE.C) else imm32;
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if postindex then R[n] else offset_addr;
    if t == 15 then  // Only possible for encodings A1 and A2
        data = PCStoreValue();
    else
        data = R[t];
    MemU_unpriv[address,4] = data;
    if postindex then R[n] = offset_addr;
```

CONSTRAINED UNPREDICTABLE behavior

If `PSTATE.EL == EL2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes as `STR` (immediate).

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SUB (immediate, from PC)

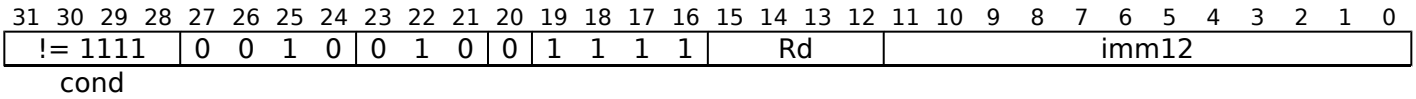
Subtract from PC subtracts an immediate value from the Align(PC, 4) value to form a PC-relative address, and writes the result to the destination register. Arm recommends that, where possible, software avoids using this alias

This is an alias of ADR. This means:

- The encodings in this description are named to match the encodings of ADR.
- The description of ADR gives the operational pseudocode for this instruction.

It has encodings from the following instruction sets: A32 (A2) and T32 (T2) .

A2



A2

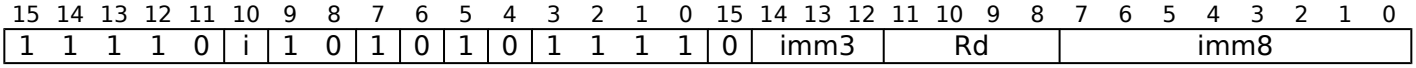
```
SUB{<c>}{<q>} <Rd>, PC, #<const>
```

is equivalent to

```
ADR{<c>}{<q>} <Rd>, <label>
```

and is the preferred disassembly when `imm12 == '000000000000'`.

T2



T2

```
SUB{<c>}{<q>} <Rd>, PC, #<imm12>
```

is equivalent to

```
ADR{<c>}{<q>} <Rd>, <label>
```

and is the preferred disassembly when `i:imm3:imm8 == '000000000000'`.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rd>	For encoding A2: is the general-purpose destination register, encoded in the "Rd" field. If the PC is used, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*. For encoding T2: is the general-purpose destination register, encoded in the "Rd" field.
<label>	For encoding A2: the label of an instruction or literal data item whose address is to be loaded into <Rd>. The assembler calculates the required value of the offset from the Align(PC, 4) value of the ADR instruction to this label. If the offset is zero or positive, encoding A1 is used, with imm32 equal to the offset. If the offset is negative, encoding A2 is used, with imm32 equal to the size of the offset. That is, the use of encoding A2 indicates that the required offset is minus the value of imm32. Permitted values of the size of the offset are any of the constants described in *Modified immediate constants in A32 instructions*.

For encoding T2: the label of an instruction or literal data item whose address is to be loaded into <Rd>. The assembler calculates the required value of the offset from the Align(PC, 4) value of the `ADR` instruction to this label.

If the offset is zero or positive, encoding T3 is used, with imm32 equal to the offset.

If the offset is negative, encoding T2 is used, with imm32 equal to the size of the offset. That is, the use of encoding T2 indicates that the required offset is minus the value of imm32.

Permitted values of the size of the offset are 0-4095.

<imm12> Is a 12-bit unsigned immediate, in the range 0 to 4095, encoded in the "i:imm3:imm8" field.

<const> An immediate value. See *Modified immediate constants in A32 instructions* for the range of values.

Operation

The description of ADR gives the operational pseudocode for this instruction.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SUB, SUBS (immediate)

Subtract (immediate) subtracts an immediate value from a register value, and writes the result to the destination register.

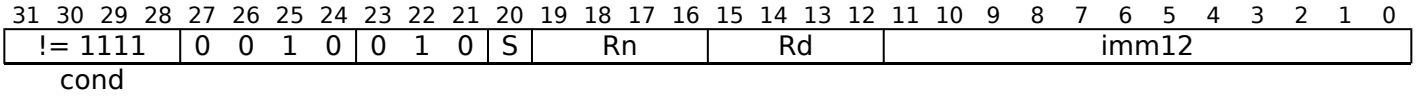
If the destination register is not the PC, the SUBS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. If the destination register is the PC:

- The SUB variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*.
- The SUBS variant of the instruction performs an exception return without the use of the stack. In this case:
  - The PE branches to the address written to the PC, and restores *PSTATE* from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state*.
  - The instruction is UNDEFINED in Hyp mode, except for encoding T5 with <imm8> set to zero, which is the encoding for the ERET instruction, see *ERET*.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 , T2 , T3 , T4 and T5) .

A1



SUB (S == 0 && Rn != 11x1)

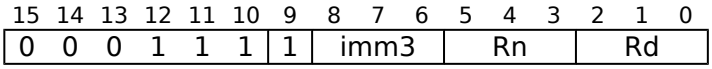
```
SUB{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

SUBS (S == 1 && Rn != 1101)

```
SUBS{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

```
if Rn == '1111' && S == '0' then SEE "ADR";
if Rn == '1101' then SEE "SUB (SP minus immediate)";
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = A32ExpandImm(imm12);
```

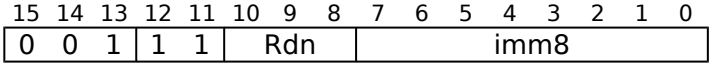
T1



T1

```
SUB<c>{<q>} <Rd>, <Rn>, #<imm3> // (Inside IT block)

SUBS{<q>} <Rd>, <Rn>, #<imm3> // (Outside IT block)

d = UInt(Rd);  n = UInt(Rn);  setflags = !InITBlock();  imm32 = ZeroExtend(imm3, 32);
```

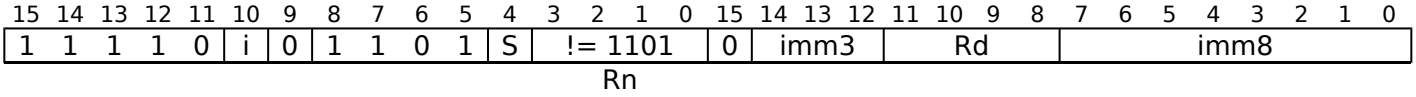
T2



T2

```
SUB<c>{<q>} <Rdn>, #<imm8> // (Inside IT block, and <Rdn>, <imm8> can be represented in T1)

SUB<c>{<q>} {<Rdn>,} <Rdn>, #<imm8> // (Inside IT block, and <Rdn>, <imm8> cannot be represented in T1)

SUBS{<q>} <Rdn>, #<imm8> // (Outside IT block, and <Rdn>, <imm8> can be represented in T1)

SUBS{<q>} {<Rdn>,} <Rdn>, #<imm8> // (Outside IT block, and <Rdn>, <imm8> cannot be represented in T1)

d = UInt(Rdn);  n = UInt(Rdn);  setflags = !InITBlock();  imm32 = ZeroExtend(imm8, 32);
```

T3



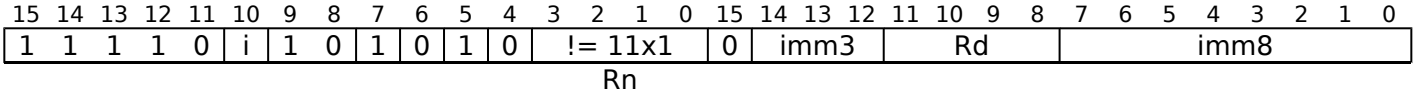
SUB (S == 0)

```
SUB<c>.W {<Rd>,} <Rn>, #<const> // (Inside IT block, and <Rd>, <Rn>, <const> can be represented in T1 or

SUB{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

SUBS (S == 1 && Rd != 1111)

```
SUBS.W {<Rd>,} <Rn>, #<const> // (Outside IT block, and <Rd>, <Rn>, <const> can be represented in T1 or T

SUBS{<c>}{<q>} {<Rd>,} <Rn>, #<const>

if Rd == '1111' && S == '1' then SEE "CMP (immediate)";
if Rn == '1101' then SEE "SUB (SP minus immediate)";
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = T32ExpandImm(i:imm3:imm8);
if (d == 15 && !setflags) || n == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

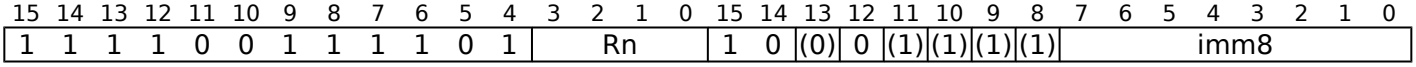
T4



T4

```
SUB{<c>}{<q>} {<Rd>,} <Rn>, #<imm12> // (<imm12> cannot be represented in T1, T2, or T3)

SUBW{<c>}{<q>} {<Rd>,} <Rn>, #<imm12> // (<imm12> can be represented in T1, T2, or T3)

if Rn == '1111' then SEE "ADR";
if Rn == '1101' then SEE "SUB (SP minus immediate)";
d = UInt(Rd);  n = UInt(Rn);  setflags = FALSE;  imm32 = ZeroExtend(i:imm3:imm8, 32);
if d == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

T5



T5 (!(Rn == 1110 && imm8 == 00000000))

```
SUBS{<c>}{<q>} PC, LR, #<imm8>
```

```
if Rn == '1110' && IsZero(imm8) then SEE "ERET";
d = 15;  n = UInt(Rn);  setflags = TRUE;  imm32 = ZeroExtend(imm8, 32);
if n != 14 then UNPREDICTABLE;
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *SUBS PC. LR and related instructions (A32)* and *SUBS PC, LR and related instructions (T32)*.

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<Rdn> Is the general-purpose source and destination register, encoded in the "Rdn" field.

<imm8> For encoding T2: is a 8-bit unsigned immediate, in the range 0 to 255, encoded in the "imm8" field.

For encoding T5: is a 8-bit unsigned immediate, in the range 0 to 255, encoded in the "imm8" field. If <Rn> is the LR, and zero is used, see *ERET*.

<Rd> For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. If the PC is used:

- For the SUB variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*.
- For the SUBS variant, the instruction performs an exception return, that restores *PSTATE* from SPSR_<current_mode>. Arm deprecates use of this instruction unless <Rn> is the LR.

For encoding T1, T3 and T4: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.

<Rn> For encoding A1 and T4: is the general-purpose source register, encoded in the "Rn" field. If the SP is used, see *SUB (SP minus immediate)*. If the PC is used, see *ADR*.

For encoding T1: is the general-purpose source register, encoded in the "Rn" field.

For encoding T3: is the general-purpose source register, encoded in the "Rn" field. If the SP is used, see *SUB (SP minus immediate)*.

<imm3> Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "imm3" field.

<imm12> Is a 12-bit unsigned immediate, in the range 0 to 4095, encoded in the "i:imm3:imm8" field.

<const> For encoding A1: an immediate value. See *Modified immediate constants in A32 instructions* for the range of values.

For encoding T3: an immediate value. See *Modified immediate constants in T32 instructions* for the range of values.

In the T32 instruction set, MOVS{<c>}{<q>} PC, LR is a pseudo-instruction for SUBS{<c>}{<q>} PC, LR, #0.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, nzcv) = AddWithCarry(R[n], NOT(imm32), '1');
    if d == 15 then
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.<N,Z,C,V> = nzcv;
```

Operational information

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SUB, SUBS (register)

Subtract (register) subtracts an optionally-shifted register value from a register value, and writes the result to the destination register.

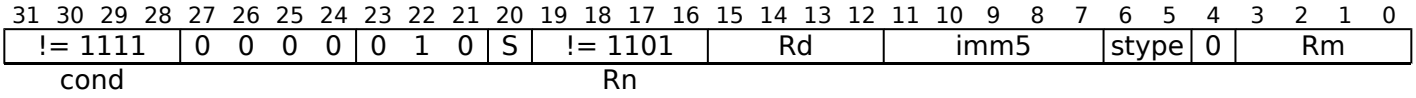
If the destination register is not the PC, the SUBS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. However, when the destination register is the PC:

- The SUB variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*.
- The SUBS variant of the instruction performs an exception return without the use of the stack. Arm deprecates use of this instruction. However, in this case:
  - The PE branches to the address written to the PC, and restores *PSTATE* from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state*.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2) .

A1



SUB, rotate right with extend (S == 0 && imm5 == 00000 && stype == 11)

```
SUB{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

SUB, shift or rotate by value (S == 0 && !(imm5 == 00000 && stype == 11))

```
SUB{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

SUBS, rotate right with extend (S == 1 && imm5 == 00000 && stype == 11)

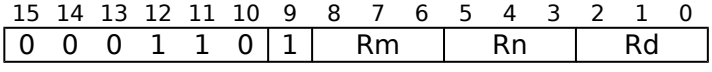
```
SUBS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

SUBS, shift or rotate by value (S == 1 && !(imm5 == 00000 && stype == 11))

```
SUBS{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

```
if Rn == '1101' then SEE "SUB (SP minus register)";
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm5);
```

T1

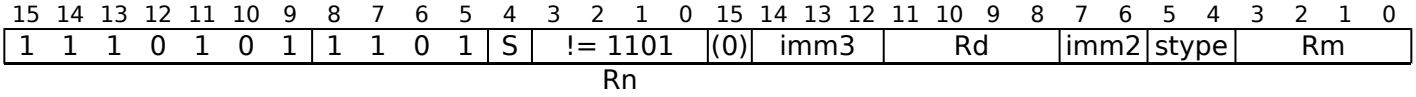


T1

```
SUB<c>{<q>} <Rd>, <Rn>, <Rm> // (Inside IT block)

SUBS{<q>} {<Rd>,} <Rn>, <Rm> // (Outside IT block)
```

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);
```

T2



SUB, rotate right with extend (S == 0 && imm3 == 000 && imm2 == 00 && stype == 11)

```
SUB{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

SUB, shift or rotate by value (S == 0 && !(imm3 == 000 && imm2 == 00 && stype == 11))

```
SUB<c>.W {<Rd>,} <Rn>, <Rm> // (Inside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1)

SUB{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

SUBS, rotate right with extend (S == 1 && imm3 == 000 && Rd != 1111 && imm2 == 00 && stype == 11)

```
SUBS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

SUBS, shift or rotate by value (S == 1 && !(imm3 == 000 && imm2 == 00 && stype == 11) && Rd != 1111)

```
SUBS.W {<Rd>,} <Rn>, <Rm> // (Outside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1)

SUBS{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

```
if Rd == '1111' && S == '1' then SEE "CMP (register)";
if Rn == '1101' then SEE "SUB (SP minus register)";
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm3:imm2);
if (d == 15 && !setflags) || n == 15 || m == 15 then UNPREDICTABLE;
// Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<Rd> For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the SUB variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*.
- For the SUBS variant, the instruction performs an exception return, that restores *PSTATE* from SPSR_<current_mode>.

For encoding T1 and T2: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.

<Rn> For encoding A1: is the first general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated. If the SP is used, see *SUB (SP minus register)*.

For encoding T1: is the first general-purpose source register, encoded in the "Rn" field.

For encoding T2: is the first general-purpose source register, encoded in the "Rn" field. If the SP is used, see *SUB (SP minus register)*.

<Rm> For encoding A1: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.

For encoding T1 and T2: is the second general-purpose source register, encoded in the "Rm" field.

<shift> Is the type of shift to be applied to the second source register, encoded in “stype”:

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<amount> For encoding A1: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm5" field as <amount> modulo 32.

For encoding T2: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the "imm3:imm2" field as <amount> modulo 32.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (result, nzcv) = AddWithCarry(R[n], NOT(shifted), '1');
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.<N,Z,C,V> = nzcv;
```

Operational information

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

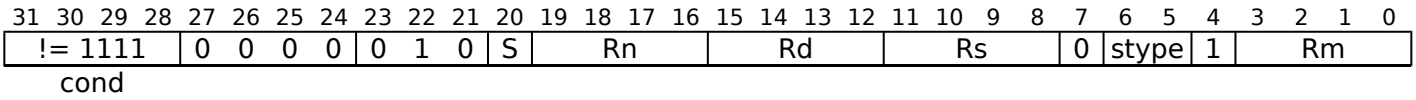
Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SUB, SUBS (register-shifted register)

Subtract (register-shifted register) subtracts a register-shifted register value from a register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

A1



Flag setting (S == 1)

```
SUBS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, <shift> <Rs>
```

Not flag setting (S == 0)

```
SUB{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, <shift> <Rs>
```

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  s = UInt(Rs);
setflags = (S == '1');  shift_t = DecodeRegShift(stype);
if d == 15 || n == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<Rd> Is the general-purpose destination register, encoded in the "Rd" field.

<Rn> Is the first general-purpose source register, encoded in the "Rn" field.

<Rm> Is the second general-purpose source register, encoded in the "Rm" field.

<shift> Is the type of shift to be applied to the second source register, encoded in “stype”:

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<Rs> Is the third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[s]<7:0>);
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (result, nzcv) = AddWithCarry(R[n], NOT(shifted), '1');
    R[d] = result;
    if setflags then
        PSTATE.<N,Z,C,V> = nzcv;
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SUB, SUBS (SP minus immediate)

Subtract from SP (immediate) subtracts an immediate value from the SP value, and writes the result to the destination register.

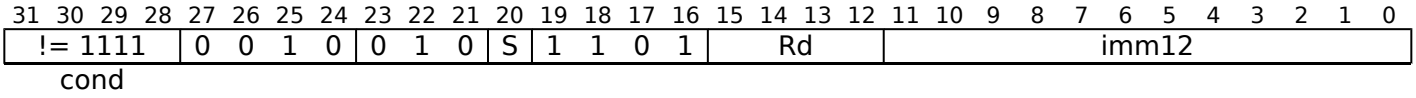
If the destination register is not the PC, the SUBS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. If the destination register is the PC:

- The SUB variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*.
- The SUBS variant of the instruction performs an exception return without the use of the stack. Arm deprecates use of this instruction. However, in this case:
  - The PE branches to the address written to the PC, and restores *PSTATE* from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state*.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 , T2 and T3) .

A1



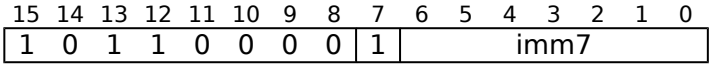
SUB (S == 0)

```
SUB{<c>}{<q>} {<Rd>,} SP, #<const>
```

SUBS (S == 1)

```
SUBS{<c>}{<q>} {<Rd>,} SP, #<const>

d = UInt(Rd);  setflags = (S == '1');  imm32 = A32ExpandImm(imm12);
```

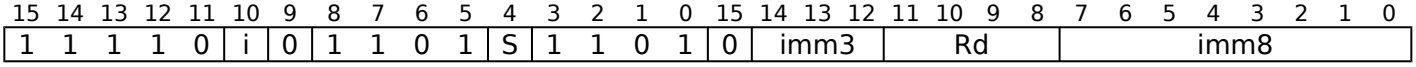
T1



T1

```
SUB{<c>}{<q>} {SP,} SP, #<imm7>

d = 13;  setflags = FALSE;  imm32 = ZeroExtend(imm7:'00', 32);
```

T2



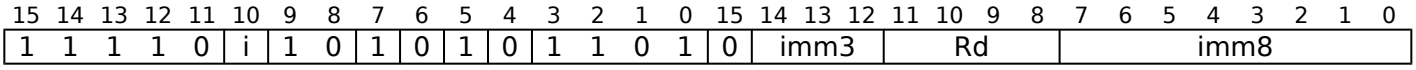
SUB (S == 0)

```
SUB{<c>}.W {<Rd>,} SP, #<const> // (<Rd>, <const> can be represented in T1)

SUB{<c>}{<q>} {<Rd>,} SP, #<const>
```

SUBS (S == 1 && Rd != 1111)

```
SUBS{<c>}{<q>} {<Rd>,} SP, #<const>

if Rd == '1111' && S == '1' then SEE "CMP (immediate)";
d = UInt(Rd);  setflags = (S == '1');  imm32 = T32ExpandImm(i:imm3:imm8);
if d == 15 && !setflags then UNPREDICTABLE;
```

T3



T3

```
SUB{<c>}{<q>} {<Rd>,} SP, #<imm12> // (<imm12> cannot be represented in T1, T2, or T3)

SUBW{<c>}{<q>} {<Rd>,} SP, #<imm12> // (<imm12> can be represented in T1, T2, or T3)

d = UInt(Rd);  setflags = FALSE;  imm32 = ZeroExtend(i:imm3:imm8, 32);
if d == 15 then UNPREDICTABLE;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
SP,	Is the stack pointer.
<imm7>	Is the unsigned immediate, a multiple of 4, in the range 0 to 508, encoded in the "imm7" field as <imm7>/4.
<Rd>	For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the SP. If the PC is used: • For the SUB variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*. • For the SUBS variant, the instruction performs an exception return, that restores *PSTATE* from SPSR_<current_mode>. Arm deprecates use of this instruction unless <Rn> is the LR. For encoding T2 and T3: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the SP.
<imm12>	Is a 12-bit unsigned immediate, in the range 0 to 4095, encoded in the "i:imm3:imm8" field.
<const>	For encoding A1: an immediate value. See *Modified immediate constants in A32 instructions* for the range of values. For encoding T2: an immediate value. See *Modified immediate constants in T32 instructions* for the range of values.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, nzcv) = AddWithCarry(SP, NOT(imm32), '1');
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.<N,Z,C,V> = nzcv;
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SUB, SUBS (SP minus register)

Subtract from SP (register) subtracts an optionally-shifted register value from the SP value, and writes the result to the destination register.

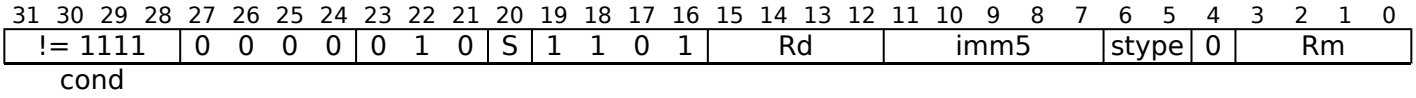
If the destination register is not the PC, the SUBS variant of the instruction updates the condition flags based on the result.

The field descriptions for <Rd> identify the encodings where the PC is permitted as the destination register. If the destination register is the PC:

- The SUB variant of the instruction is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*.
- The SUBS variant of the instruction performs an exception return without the use of the stack. Arm deprecates use of this instruction. However, in this case:
  - The PE branches to the address written to the PC, and restores *PSTATE* from SPSR_<current_mode>.
  - The PE checks SPSR_<current_mode> for an illegal return event. See *Illegal return events from AArch32 state*.
  - The instruction is UNDEFINED in Hyp mode.
  - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



SUB, rotate right with extend (S == 0 && imm5 == 00000 && stype == 11)

```
SUB{<c>}{<q>} {<Rd>,} SP, <Rm> , RRX
```

SUB, shift or rotate by value (S == 0 && !(imm5 == 00000 && stype == 11))

```
SUB{<c>}{<q>} {<Rd>,} SP, <Rm> {, <shift> #<amount>}
```

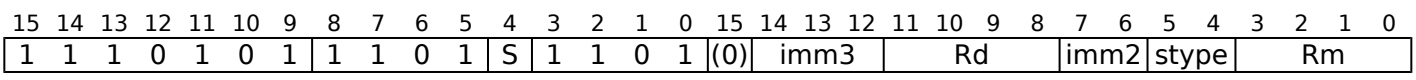
SUBS, rotate right with extend (S == 1 && imm5 == 00000 && stype == 11)

```
SUBS{<c>}{<q>} {<Rd>,} SP, <Rm> , RRX
```

SUBS, shift or rotate by value (S == 1 && !(imm5 == 00000 && stype == 11))

```
SUBS{<c>}{<q>} {<Rd>,} SP, <Rm> {, <shift> #<amount>}

d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm5);
```

T1



SUB, rotate right with extend (S == 0 && imm3 == 000 && imm2 == 00 && stype == 11)

```
SUB{<c>}{<q>} {<Rd>,} SP, <Rm>, RRX
```

SUB, shift or rotate by value (S == 0 && !(imm3 == 000 && imm2 == 00 && stype == 11))

```
SUB{<c>}.W {<Rd>,} SP, <Rm> // (<Rd>, <Rm> can be represented in T1 or T2)

SUB{<c>}{<q>} {<Rd>,} SP, <Rm> {, <shift> #<amount>}
```

SUBS, rotate right with extend (S == 1 && imm3 == 000 && Rd != 1111 && imm2 == 00 && stype == 11)

```
SUBS{<c>}{<q>} {<Rd>,} SP, <Rm>, RRX
```

SUBS, shift or rotate by value (S == 1 && !(imm3 == 000 && imm2 == 00 && stype == 11) && Rd != 1111)

```
SUBS{<c>}{<q>} {<Rd>,} SP, <Rm> {, <shift> #<amount>}
```

```
if Rd == '1111' && S == '1' then SEE "CMP (register)";
d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm3:imm2);
if (d == 15 && !setflags) || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<Rd> For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the SP. Arm deprecates using the PC as the destination register, but if the PC is used:

- For the SUB variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*.
- For the SUBS variant, the instruction performs an exception return, that restores *PSTATE* from SPSR_<current_mode>.

For encoding T1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the SP.

<Rm> For encoding A1: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.

For encoding T1: is the second general-purpose source register, encoded in the "Rm" field.

<shift> Is the type of shift to be applied to the second source register, encoded in “stype”:

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<amount> For encoding A1: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm5" field as <amount> modulo 32.

For encoding T1: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the "imm3:imm2" field as <amount> modulo 32.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (result, nzcv) = AddWithCarry(SP, NOT(shifted), '1');
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.<N,Z,C,V> = nzcv;
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SVC

Supervisor Call causes a Supervisor Call exception. For more information, see *Supervisor Call (SVC) exception*.

SVC was previously called SWI, Software Interrupt, and this name is still found in some documentation.

Software can use this instruction as a call to an operating system to provide a service.

In the following cases, the Supervisor Call exception generated by the SVC instruction is taken to Hyp mode:

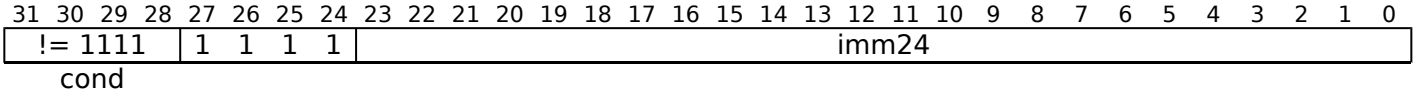
- If the SVC is executed in Hyp mode.
- If *HCR*.TGE is set to 1, and the SVC is executed in Non-secure User mode. For more information, see *Supervisor Call exception, when HCR.TGE is set to 1*

In these cases, the *HSR, Hyp Syndrome Register* identifies that the exception entry was caused by a Supervisor Call exception, EC value 0x11, see *Use of the HSR*. The immediate field in the *HSR*:

- If the SVC is unconditional:
  - For the T32 instruction, is the zero-extended value of the imm8 field.
  - For the A32 instruction, is the least-significant 16 bits the imm24 field.
- If the SVC is conditional, is UNKNOWN.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

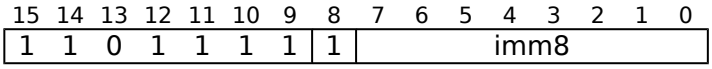


A1

```
SVC{<c>}{<q>} {#}<imm>
```

```
imm32 = ZeroExtend(imm24, 32);
```

T1



T1

```
SVC{<c>}{<q>} {#}<imm>
```

```
imm32 = ZeroExtend(imm8, 32);
```

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<imm>	For encoding A1: is a 24-bit unsigned immediate, in the range 0 to 16777215, encoded in the "imm24" field. This value is for assembly and disassembly only. SVC handlers in some systems interpret imm24 in software, for example to determine the required service. For encoding T1: is a 8-bit unsigned immediate, in the range 0 to 255, encoded in the "imm8" field. This value is for assembly and disassembly only. SVC handlers in some systems interpret imm8 in software, for example to determine the required service.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    AArch32.CheckForSVCTrap(imm32<15:0>);
    AArch32.CallSupervisor(imm32<15:0>);
```

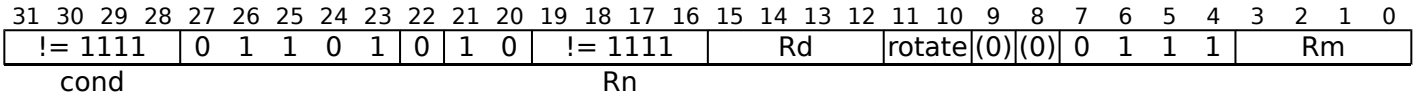
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SXTAB

Signed Extend and Add Byte extracts an 8-bit value from a register, sign-extends it to 32 bits, adds the result to the value in another register, and writes the final result to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit value.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

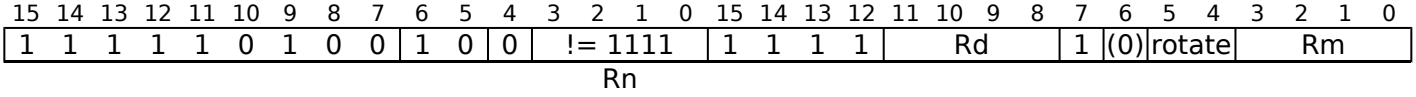


A1

```
SXTAB{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, ROR #<amount>}
```

```
if Rn == '1111' then SEE "SXTB";
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE;
```

T1



T1

```
SXTAB{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, ROR #<amount>}
```

```
if Rn == '1111' then SEE "SXTB";
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<Rd> Is the general-purpose destination register, encoded in the "Rd" field.

<Rn> Is the first general-purpose source register, encoded in the "Rn" field.

<Rm> Is the second general-purpose source register, encoded in the "Rm" field.

<amount> Is the rotate amount, encoded in “rotate”:

rotate	<amount>
00	(omitted)
01	8
10	16
11	24

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d] = R[n] + SignExtend(rotated<7:0>, 32);
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

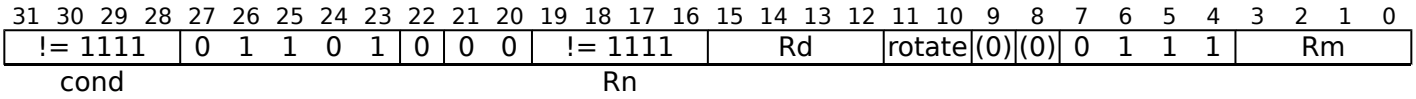
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SXTAB16

Signed Extend and Add Byte 16 extracts two 8-bit values from a register, sign-extends them to 16 bits each, adds the results to two 16-bit values from another register, and writes the final results to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit values.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

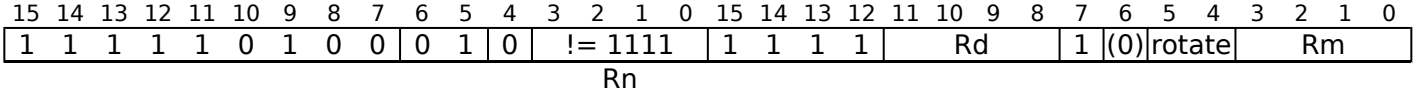


A1

```
SXTAB16{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, ROR #<amount>}
```

```
if Rn == '1111' then SEE "SXTB16";
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE;
```

T1



T1

```
SXTAB16{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, ROR #<amount>}
```

```
if Rn == '1111' then SEE "SXTB16";
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<Rd> Is the general-purpose destination register, encoded in the "Rd" field.

<Rn> Is the first general-purpose source register, encoded in the "Rn" field.

<Rm> Is the second general-purpose source register, encoded in the "Rm" field.

<amount> Is the rotate amount, encoded in “rotate”:

rotate	<amount>
00	(omitted)
01	8
10	16
11	24

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d]<15:0>  = R[n]<15:0> + SignExtend(rotated<7:0>, 16);
    R[d]<31:16> = R[n]<31:16> + SignExtend(rotated<23:16>, 16);
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

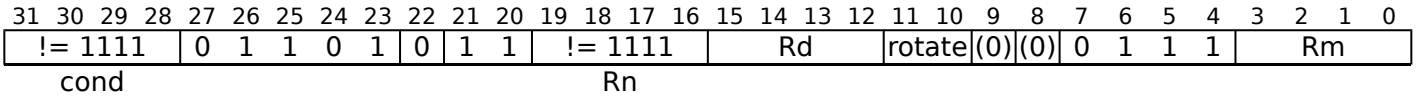
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SXTAH

Signed Extend and Add Halfword extracts a 16-bit value from a register, sign-extends it to 32 bits, adds the result to a value from another register, and writes the final result to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 16-bit value.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

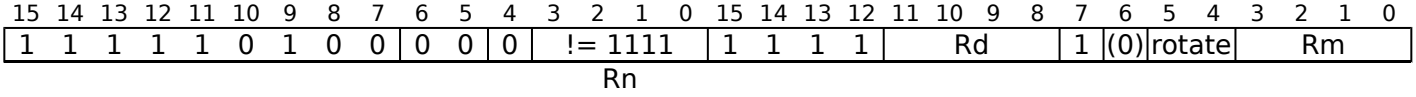


A1

```
SXTAH{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, ROR #<amount>}
```

```
if Rn == '1111' then SEE "SXTH";
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE;
```

T1



T1

```
SXTAH{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, ROR #<amount>}
```

```
if Rn == '1111' then SEE "SXTH";
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<Rd> Is the general-purpose destination register, encoded in the "Rd" field.

<Rn> Is the first general-purpose source register, encoded in the "Rn" field.

<Rm> Is the second general-purpose source register, encoded in the "Rm" field.

<amount> Is the rotate amount, encoded in “rotate”:

rotate	<amount>
00	(omitted)
01	8
10	16
11	24

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d] = R[n] + SignExtend(rotated<15:0>, 32);
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

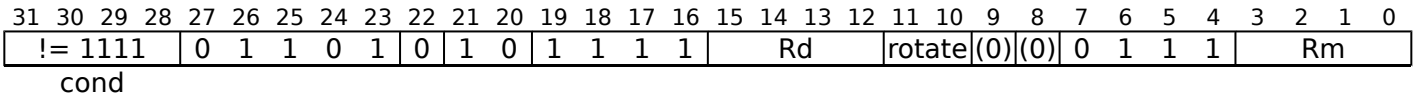
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SXTB

Signed Extend Byte extracts an 8-bit value from a register, sign-extends it to 32 bits, and writes the result to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit value.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2) .

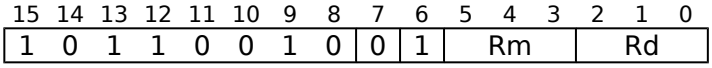
A1



A1

```
SXTB{<c>}{<q>} {<Rd>,} <Rm> {, ROR #<amount>}

d = UInt(Rd);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE;
```

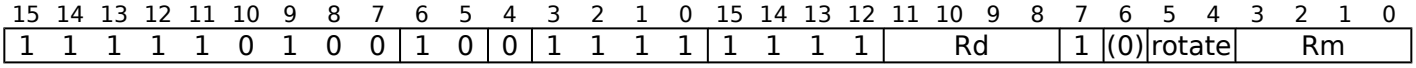
T1



T1

```
SXTB{<c>}{<q>} {<Rd>,} <Rm>

d = UInt(Rd);  m = UInt(Rm);  rotation = 0;
```

T2



T2

```
SXTB{<c>}.W {<Rd>,} <Rm> // (<Rd>, <Rm> can be represented in T1)

SXTB{<c>}{<q>} {<Rd>,} <Rm> {, ROR #<amount>}

d = UInt(Rd);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rm>	Is the general-purpose source register, encoded in the "Rm" field.
<amount>	Is the rotate amount, encoded in “rotate”:

rotate	<amount>
00	(omitted)
01	8
10	16
11	24

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d] = SignExtend(rotated<7:0>, 32);
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

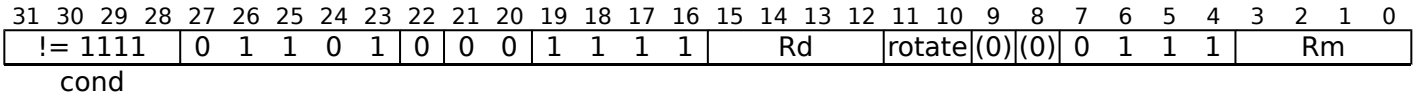
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SXTB16

Signed Extend Byte 16 extracts two 8-bit values from a register, sign-extends them to 16 bits each, and writes the results to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit values.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

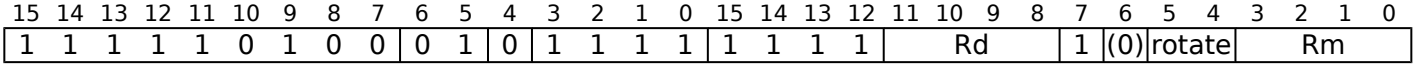
A1



A1

```
SXTB16{<c>}{<q>} {<Rd>,} <Rm> {, ROR #<amount>}

d = UInt(Rd);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE;
```

T1



T1

```
SXTB16{<c>}{<q>} {<Rd>,} <Rm> {, ROR #<amount>}

d = UInt(Rd);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<Rd> Is the general-purpose destination register, encoded in the "Rd" field.

<Rm> Is the general-purpose source register, encoded in the "Rm" field.

<amount> Is the rotate amount, encoded in “rotate”:

rotate	<amount>
00	(omitted)
01	8
10	16
11	24

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d]<15:0>  = SignExtend(rotated<7:0>, 16);
    R[d]<31:16> = SignExtend(rotated<23:16>, 16);
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

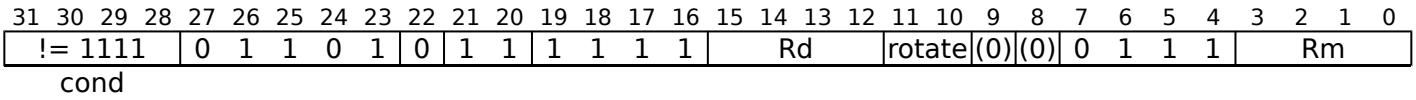
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SXTH

Signed Extend Halfword extracts a 16-bit value from a register, sign-extends it to 32 bits, and writes the result to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 16-bit value.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2) .

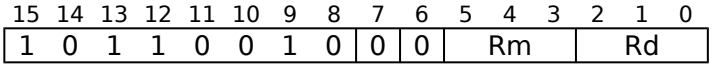
A1



A1

```
SXTH{<c>}{<q>} {<Rd>,} <Rm> {, ROR #<amount>}

d = UInt(Rd);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE;
```

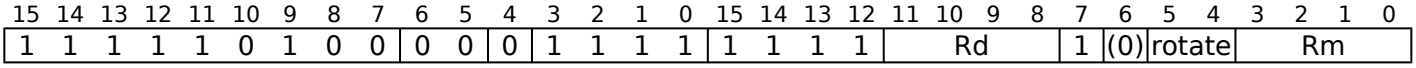
T1



T1

```
SXTH{<c>}{<q>} {<Rd>,} <Rm>

d = UInt(Rd);  m = UInt(Rm);  rotation = 0;
```

T2



T2

```
SXTH{<c>}.W {<Rd>,} <Rm> // (<Rd>, <Rm> can be represented in T1)

SXTH{<c>}{<q>} {<Rd>,} <Rm> {, ROR #<amount>}

d = UInt(Rd);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rm>	Is the general-purpose source register, encoded in the "Rm" field.
<amount>	Is the rotate amount, encoded in “rotate”:

rotate	<amount>
00	(omitted)
01	8
10	16
11	24

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d] = SignExtend(rotated<15:0>, 32);
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

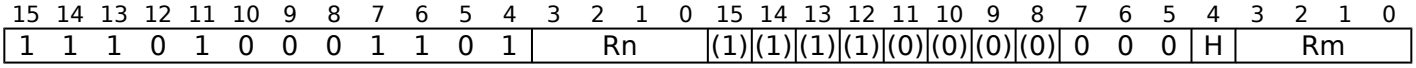
Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

TBB, TBH

Table Branch Byte or Halfword causes a PC-relative forward branch using a table of single byte or halfword offsets. A base register provides a pointer to the table, and a second register supplies an index into the table. The branch length is twice the value returned from the table.

T1



Byte (H == 0)

```
TBB{<c>}{<q>} [<Rn>, <Rm>] // (Outside or last in IT block)
```

Halfword (H == 1)

```
TBH{<c>}{<q>} [<Rn>, <Rm>, LSL #1] // (Outside or last in IT block)
```

```
n = UInt(Rn);  m = UInt(Rm);  is_tbh = (H == '1');
if m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rn>	Is the general-purpose base register holding the address of the table of branch lengths, encoded in the "Rn" field. The PC can be used. If it is, the table immediately follows this instruction.
<Rm>	For the byte variant: is the general-purpose index register, encoded in the "Rm" field. This register contains an integer pointing to a single byte in the table. The offset in the table is the value of the index. For the halfword variant: is the general-purpose index register, encoded in the "Rm" field. This register contains an integer pointing to a halfword in the table. The offset in the table is twice the value of the index.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if is_tbh then
        halfwords = UInt(MemU[R[n]+LSL(R[m],1), 2]);
    else
        halfwords = UInt(MemU[R[n]+R[m], 1]);
    BranchWritePC(PC + 2*halfwords, BranchType_INDIR);
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

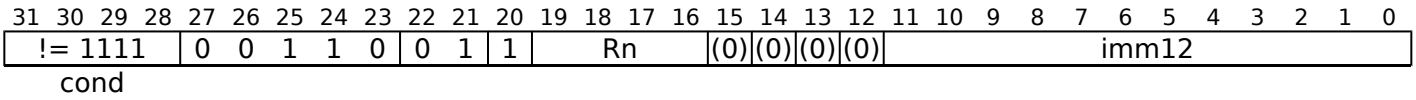
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

TEQ (immediate)

Test Equivalence (immediate) performs a bitwise exclusive OR operation on a register value and an immediate value. It updates the condition flags based on the result, and discards the result.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

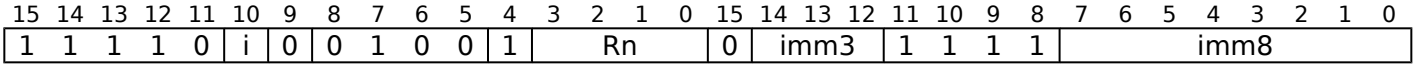


A1

```
TEQ{<c>}{<q>} <Rn>, #<const>
```

```
n = UInt(Rn);
(imm32, carry) = A32ExpandImm_C(imm12, PSTATE.C);
```

T1



T1

```
TEQ{<c>}{<q>} <Rn>, #<const>
```

```
n = UInt(Rn);
(imm32, carry) = T32ExpandImm_C(i:imm3:imm8, PSTATE.C);
if n == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rn>	For encoding A1: is the general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.
	For encoding T1: is the general-purpose source register, encoded in the "Rn" field.
<const>	For encoding A1: an immediate value. See *Modified immediate constants in A32 instructions* for the range of values.
	For encoding T1: an immediate value. See *Modified immediate constants in T32 instructions* for the range of values.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = R[n] EOR imm32;
    PSTATE.N = result<31>;
    PSTATE.Z = IsZeroBit(result);
    PSTATE.C = carry;
    // PSTATE.V unchanged
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

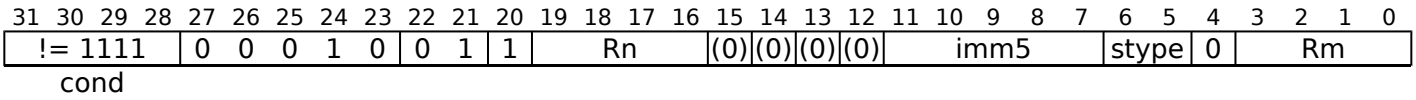
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

TEQ (register)

Test Equivalence (register) performs a bitwise exclusive OR operation on a register value and an optionally-shifted register value. It updates the condition flags based on the result, and discards the result.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



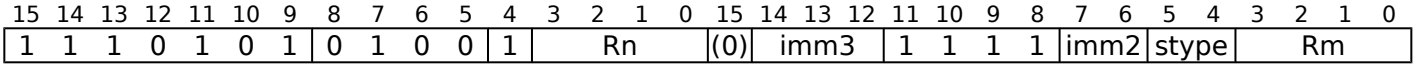
Rotate right with extend (imm5 == 00000 && stype == 11)

```
TEQ{<c>}{<q>} <Rn>, <Rm>, RRX
```

Shift or rotate by value (!(imm5 == 00000 && stype == 11))

```
TEQ{<c>}{<q>} <Rn>, <Rm> {, <shift> #<amount>}

n = UInt(Rn);  m = UInt(Rm);
(shift_t, shift_n) = DecodeImmShift(stype, imm5);
```

T1



Rotate right with extend (imm3 == 000 && imm2 == 00 && stype == 11)

```
TEQ{<c>}{<q>} <Rn>, <Rm>, RRX
```

Shift or rotate by value (!(imm3 == 000 && imm2 == 00 && stype == 11))

```
TEQ{<c>}{<q>} <Rn>, <Rm> {, <shift> #<amount>}

n = UInt(Rn);  m = UInt(Rm);
(shift_t, shift_n) = DecodeImmShift(stype, imm3:imm2);
if n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rn>	For encoding A1: is the first general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated. For encoding T1: is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	For encoding A1: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated. For encoding T1: is the second general-purpose source register, encoded in the "Rm" field.
<shift>	Is the type of shift to be applied to the second source register, encoded in “stype”:

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<amount> For encoding A1: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm5" field as <amount> modulo 32.

For encoding T1: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the "imm3:imm2" field as <amount> modulo 32.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
    result = R[n] EOR shifted;
    PSTATE.N = result<31>;
    PSTATE.Z = IsZeroBit(result);
    PSTATE.C = carry;
    // PSTATE.V unchanged
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

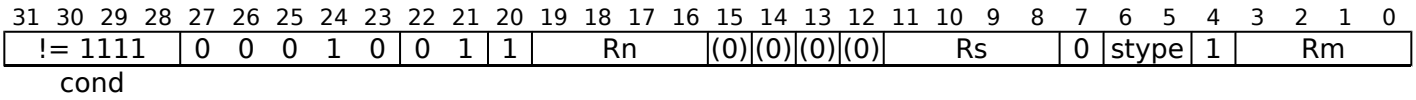
Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

TEQ (register-shifted register)

Test Equivalence (register-shifted register) performs a bitwise exclusive OR operation on a register value and a register-shifted register value. It updates the condition flags based on the result, and discards the result.

A1



A1

```
TEQ{<c>}{<q>} <Rn>, <Rm>, <type> <Rs>
```

```
n = UInt(Rn);  m = UInt(Rm);  s = UInt(Rs);
shift_t = DecodeRegShift(stype);
if n == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<Rn> Is the first general-purpose source register, encoded in the "Rn" field.

<Rm> Is the second general-purpose source register, encoded in the "Rm" field.

<type> Is the type of shift to be applied to the second source register, encoded in “stype”:

stype	<type>
00	LSL
01	LSR
10	ASR
11	ROR

<Rs> Is the third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[s]<7:0>);
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
    result = R[n] EOR shifted;
    PSTATE.N = result<31>;
    PSTATE.Z = IsZeroBit(result);
    PSTATE.C = carry;
    // PSTATE.V unchanged
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.

- The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

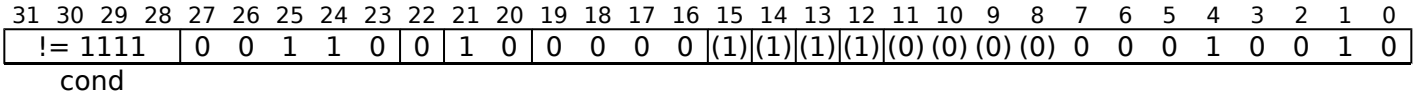
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

TSB CSYNC

Trace Synchronization Barrier. This instruction is a barrier that synchronizes the trace operations of instructions. If *FEAT_TRF* is not implemented, this instruction executes as a NOP.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1
(FEAT_TRF)



A1

```
TSB{<c>}{<q>} CSYNC
```

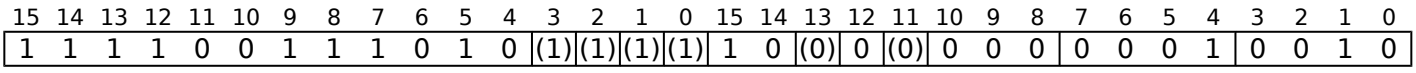
```
if !HaveSelfHostedTrace() then EndOfInstruction(); // Instruction executes as NOP
if cond != '1110' then UNPREDICTABLE;              // ESB must be encoded with AL condition
```

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
(FEAT_TRF)



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_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

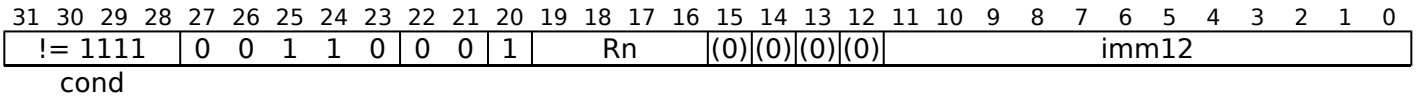
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

TST (immediate)

Test (immediate) performs a bitwise AND operation on a register value and an immediate value. It updates the condition flags based on the result, and discards the result.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

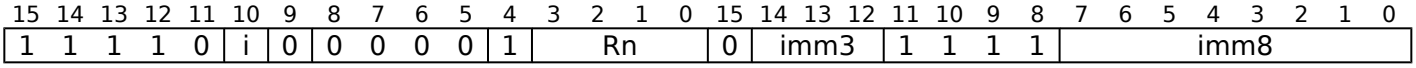


A1

```
TST{<c>}{<q>} <Rn>, #<const>
```

```
n = UInt(Rn);
(imm32, carry) = A32ExpandImm_C(imm12, PSTATE.C);
```

T1



T1

```
TST{<c>}{<q>} <Rn>, #<const>
```

```
n = UInt(Rn);
(imm32, carry) = T32ExpandImm_C(i:imm3:imm8, PSTATE.C);
if n == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rn>	For encoding A1: is the general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.
	For encoding T1: is the general-purpose source register, encoded in the "Rn" field.
<const>	For encoding A1: an immediate value. See *Modified immediate constants in A32 instructions* for the range of values.
	For encoding T1: an immediate value. See *Modified immediate constants in T32 instructions* for the range of values.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = R[n] AND imm32;
    PSTATE.N = result<31>;
    PSTATE.Z = IsZeroBit(result);
    PSTATE.C = carry;
    // PSTATE.V unchanged
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

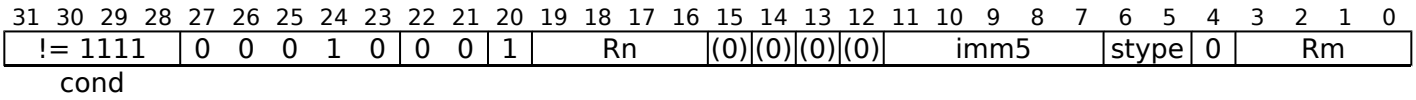
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

TST (register)

Test (register) performs a bitwise AND operation on a register value and an optionally-shifted register value. It updates the condition flags based on the result, and discards the result.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2) .

A1



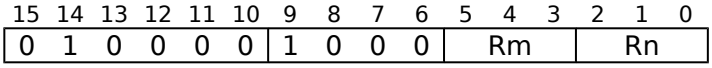
Rotate right with extend (imm5 == 00000 && stype == 11)

```
TST{<c>}{<q>} <Rn>, <Rm>, RRX
```

Shift or rotate by value (!(imm5 == 00000 && stype == 11))

```
TST{<c>}{<q>} <Rn>, <Rm> {, <shift> #<amount>}

n = UInt(Rn);  m = UInt(Rm);
(shift_t, shift_n) = DecodeImmShift(stype, imm5);
```

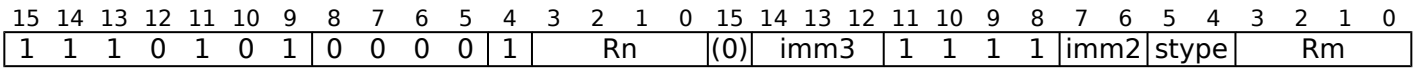
T1



T1

```
TST{<c>}{<q>} <Rn>, <Rm>

n = UInt(Rn);  m = UInt(Rm);
(shift_t, shift_n) = (SRType_LSL, 0);
```

T2



Rotate right with extend (imm3 == 000 && imm2 == 00 && stype == 11)

```
TST{<c>}{<q>} <Rn>, <Rm>, RRX
```

Shift or rotate by value (!(imm3 == 000 && imm2 == 00 && stype == 11))

```
TST{<c>}.W <Rn>, <Rm> // (<Rn>, <Rm> can be represented in T1)

TST{<c>}{<q>} <Rn>, <Rm> {, <shift> #<amount>}

n = UInt(Rn);  m = UInt(Rm);
(shift_t, shift_n) = DecodeImmShift(stype, imm3:imm2);
if n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<Rn> For encoding A1: is the first general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.

For encoding T1 and T2: is the first general-purpose source register, encoded in the "Rn" field.

<Rm> For encoding A1: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.

For encoding T1 and T2: is the second general-purpose source register, encoded in the "Rm" field.

<shift> Is the type of shift to be applied to the second source register, encoded in “stype”:

stype	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<amount> For encoding A1: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm5" field as <amount> modulo 32.

For encoding T2: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR), encoded in the "imm3:imm2" field as <amount> modulo 32.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
    result = R[n] AND shifted;
    PSTATE.N = result<31>;
    PSTATE.Z = IsZeroBit(result);
    PSTATE.C = carry;
    // PSTATE.V unchanged
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

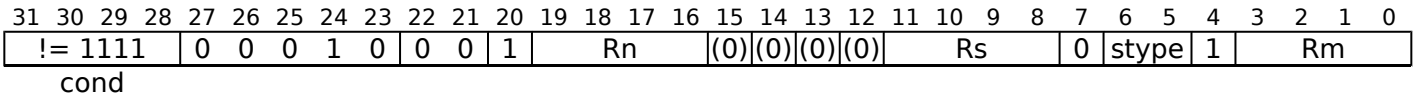
Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

TST (register-shifted register)

Test (register-shifted register) performs a bitwise AND operation on a register value and a register-shifted register value. It updates the condition flags based on the result, and discards the result.

A1



A1

```
TST{<c>}{<q>} <Rn>, <Rm>, <type> <Rs>
```

```
n = UInt(Rn);  m = UInt(Rm);  s = UInt(Rs);
shift_t = DecodeRegShift(stype);
if n == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<Rn> Is the first general-purpose source register, encoded in the "Rn" field.

<Rm> Is the second general-purpose source register, encoded in the "Rm" field.

<type> Is the type of shift to be applied to the second source register, encoded in “stype”:

stype	<type>
00	LSL
01	LSR
10	ASR
11	ROR

<Rs> Is the third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[s]<7:0>);
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
    result = R[n] AND shifted;
    PSTATE.N = result<31>;
    PSTATE.Z = IsZeroBit(result);
    PSTATE.C = carry;
    // PSTATE.V unchanged
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.

- The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

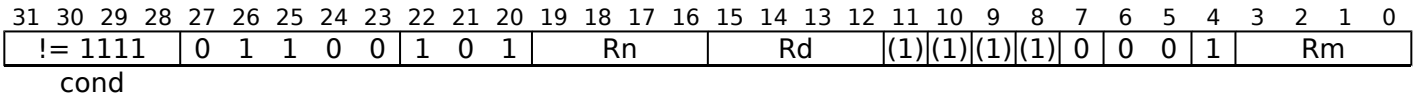
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UADD16

Unsigned Add 16 performs two 16-bit unsigned integer additions, and writes the results to the destination register. It sets *PSTATE*.GE according to the results of the additions.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

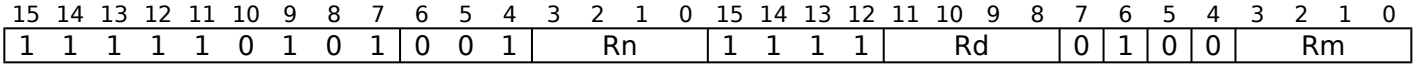
A1



A1

```
UADD16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1



T1

```
UADD16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = UInt(R[n]<15:0>) + UInt(R[m]<15:0>);
    sum2 = UInt(R[n]<31:16>) + UInt(R[m]<31:16>);
    R[d]<15:0>  = sum1<15:0>;
    R[d]<31:16> = sum2<15:0>;
    PSTATE.GE<1:0> = if sum1 >= 0x10000 then '11' else '00';
    PSTATE.GE<3:2> = if sum2 >= 0x10000 then '11' else '00';
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:

    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

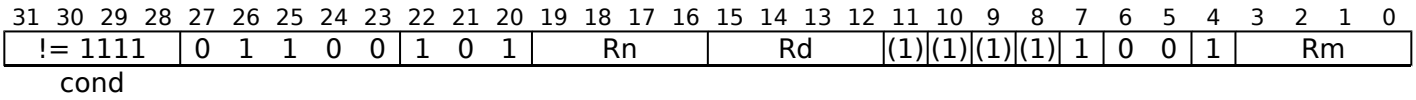
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UADD8

Unsigned Add 8 performs four unsigned 8-bit integer additions, and writes the results to the destination register. It sets *PSTATE*.GE according to the results of the additions.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

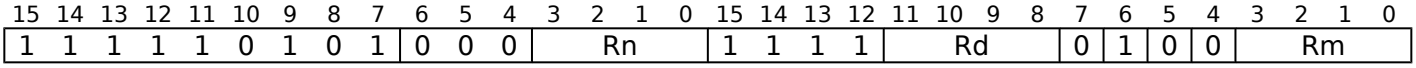
A1



A1

```
UADD8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1



T1

```
UADD8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m  == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = UInt(R[n]<7:0>) + UInt(R[m]<7:0>);
    sum2 = UInt(R[n]<15:8>) + UInt(R[m]<15:8>);
    sum3 = UInt(R[n]<23:16>) + UInt(R[m]<23:16>);
    sum4 = UInt(R[n]<31:24>) + UInt(R[m]<31:24>);
    R[d]<7:0>   = sum1<7:0>;
    R[d]<15:8>  = sum2<7:0>;
    R[d]<23:16> = sum3<7:0>;
    R[d]<31:24> = sum4<7:0>;
    PSTATE.GE<0>  = if sum1 >= 0x100 then '1' else '0';
    PSTATE.GE<1>  = if sum2 >= 0x100 then '1' else '0';
    PSTATE.GE<2>  = if sum3 >= 0x100 then '1' else '0';
    PSTATE.GE<3>  = if sum4 >= 0x100 then '1' else '0';
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

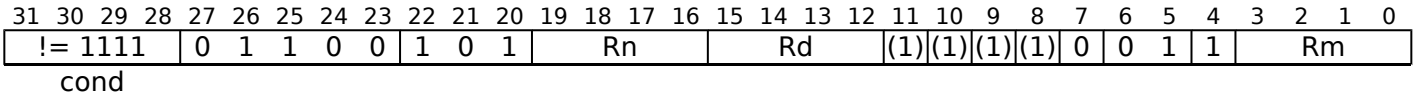
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UASX

Unsigned Add and Subtract with Exchange exchanges the two halfwords of the second operand, performs one unsigned 16-bit integer addition and one unsigned 16-bit subtraction, and writes the results to the destination register. It sets *PSTATE*.GE according to the results.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

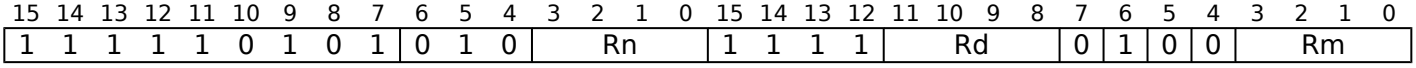
A1



A1

```
UASX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1



T1

```
UASX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff = UInt(R[n]<15:0>) - UInt(R[m]<31:16>);
    sum  = UInt(R[n]<31:16>) + UInt(R[m]<15:0>);
    R[d]<15:0>  = diff<15:0>;
    R[d]<31:16> = sum<15:0>;
    PSTATE.GE<1:0> = if diff >= 0 then '11' else '00';
    PSTATE.GE<3:2> = if sum  >= 0x10000 then '11' else '00';
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

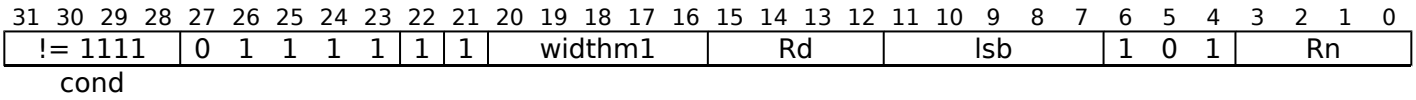
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UBFX

Unsigned Bit Field Extract extracts any number of adjacent bits at any position from a register, zero-extends them to 32 bits, and writes the result to the destination register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

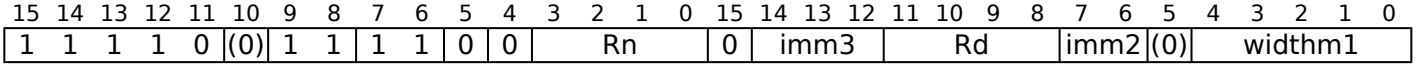
A1



A1

```
UBFX{<c>}{<q>} <Rd>, <Rn>, #<lsb>, #<width>

d = UInt(Rd);  n = UInt(Rn);
lsbit = UInt(lsb);  widthminus1 = UInt(widthm1);
if d == 15 || n == 15 then UNPREDICTABLE;
```

T1



T1

```
UBFX{<c>}{<q>} <Rd>, <Rn>, #<lsb>, #<width>

d = UInt(Rd);  n = UInt(Rn);
lsbit = UInt(imm3:imm2);  widthminus1 = UInt(widthm1);
if d == 15 || n == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the general-purpose source register, encoded in the "Rn" field.
<lsb>	For encoding A1: is the bit number of the least significant bit in the field, in the range 0 to 31, encoded in the "lsb" field. For encoding T1: is the bit number of the least significant bit in the field, in the range 0 to 31, encoded in the "imm3:imm2" field.
<width>	Is the width of the field, in the range 1 to 32-<lsb>, encoded in the "widthm1" field as <width>-1.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    msbit = lsbit + widthminus1;
    if msbit <= 31 then
        R[d] = ZeroExtend(R[n]<msbit:lsbit>, 32);
    else
        UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `msbit > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The value in the destination register is UNKNOWN.

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

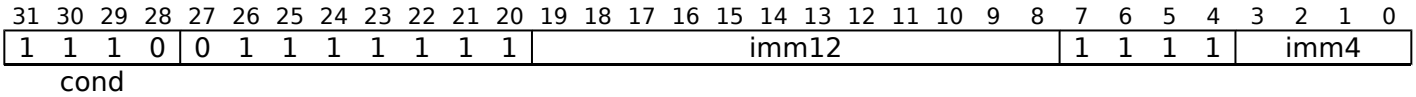
UDF

Permanently Undefined generates an Undefined Instruction exception.
The encodings for UDF used in this section are defined as permanently UNDEFINED in the Armv8-A architecture. However:

- With the T32 instruction set, Arm deprecates using the UDF instruction in an IT block.
- In the A32 instruction set, UDF is not conditional.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2) .

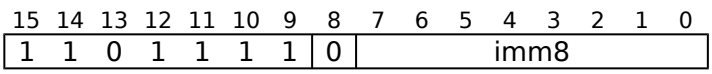
A1



A1

```
UDF{<c>}{<q>} {#}<imm>

imm32 = ZeroExtend(imm12:imm4, 32);
// imm32 is for assembly and disassembly only, and is ignored by hardware.
```

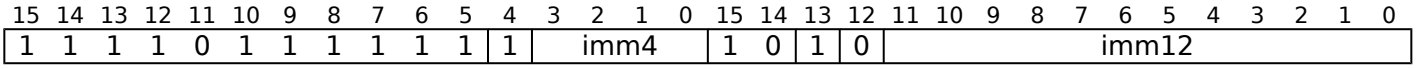
T1



T1

```
UDF{<c>}{<q>} {#}<imm>

imm32 = ZeroExtend(imm8, 32);
// imm32 is for assembly and disassembly only, and is ignored by hardware.
```

T2



T2

```
UDF{<c>}.W {#}<imm> // (<imm> can be represented in T1)

UDF{<c>}{<q>} {#}<imm>

imm32 = ZeroExtend(imm4:imm12, 32);
// imm32 is for assembly and disassembly only, and is ignored by hardware.
```

Assembler Symbols

<c>	For encoding A1: see *Standard assembler syntax fields*. <c> must be AL or omitted. For encoding T1 and T2: see *Standard assembler syntax fields*. Arm deprecates using any <c> value other than AL.
<q>	See *Standard assembler syntax fields*.
<imm>	For encoding A1: is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm12:imm4" field. The PE ignores the value of this constant.

For encoding T1: is a 8-bit unsigned immediate, in the range 0 to 255, encoded in the "imm8" field. The PE ignores the value of this constant.

For encoding T2: is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm4:imm12" field. The PE ignores the value of this constant.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    UNDEFINED;
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

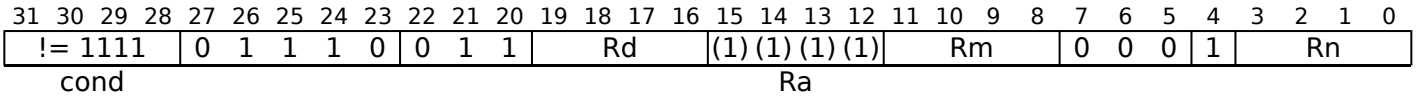
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UDIV

Unsigned Divide divides a 32-bit unsigned integer register value by a 32-bit unsigned integer register value, and writes the result to the destination register. The condition flags are not affected.
See *Divide instructions* for more information about this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



A1

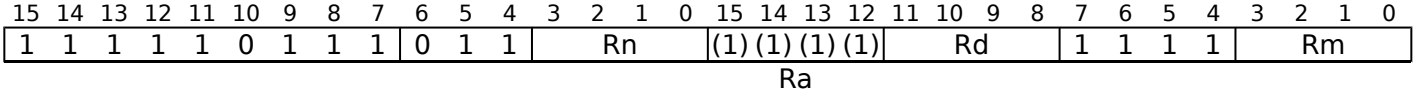
```
UDIV{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);
if d == 15 || n == 15 || m == 15 || a != 15 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `Ra != '1111'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes as described, with no change to its behavior and no additional side effects.
- The instruction performs a divide and the register specified by Ra becomes UNKNOWN.

T1



T1

```
UDIV{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);
if d == 15 || n == 15 || m == 15 || a != 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

CONSTRAINED UNPREDICTABLE behavior

If `Ra != '1111'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes as described, with no change to its behavior and no additional side effects.
- The instruction performs a divide and the register specified by Ra becomes UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.

<Rn> Is the first general-purpose source register holding the dividend, encoded in the "Rn" field.

<Rm> Is the second general-purpose source register holding the divisor, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if UInt(R[m]) == 0 then
        result = 0;
    else
        result = RoundTowardsZero(Real(UInt(R[n])) / Real(UInt(R[m])));
    R[d] = result<31:0>;
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

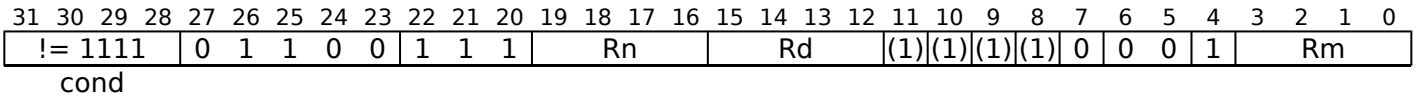
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UHADD16

Unsigned Halving Add 16 performs two unsigned 16-bit integer additions, halves the results, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

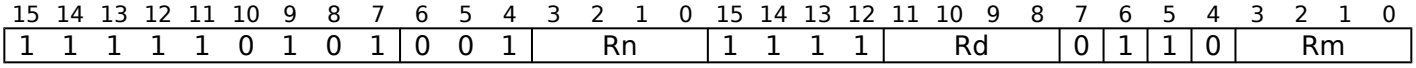
A1



A1

```
UHADD16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1



T1

```
UHADD16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = UInt(R[n]<15:0>) + UInt(R[m]<15:0>);
    sum2 = UInt(R[n]<31:16>) + UInt(R[m]<31:16>);
    R[d]<15:0>  = sum1<16:1>;
    R[d]<31:16> = sum2<16:1>;
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

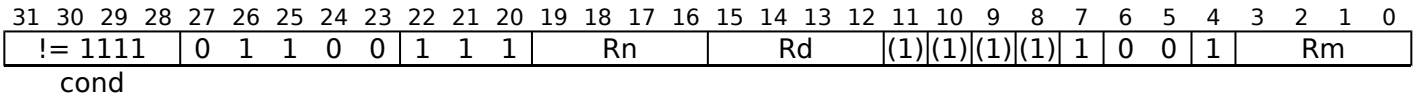
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UHADD8

Unsigned Halving Add 8 performs four unsigned 8-bit integer additions, halves the results, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

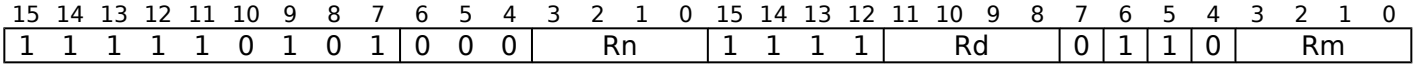
A1



A1

```
UHADD8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1



T1

```
UHADD8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = UInt(R[n]<7:0>) + UInt(R[m]<7:0>);
    sum2 = UInt(R[n]<15:8>) + UInt(R[m]<15:8>);
    sum3 = UInt(R[n]<23:16>) + UInt(R[m]<23:16>);
    sum4 = UInt(R[n]<31:24>) + UInt(R[m]<31:24>);
    R[d]<7:0>   = sum1<8:1>;
    R[d]<15:8>  = sum2<8:1>;
    R[d]<23:16> = sum3<8:1>;
    R[d]<31:24> = sum4<8:1>;
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

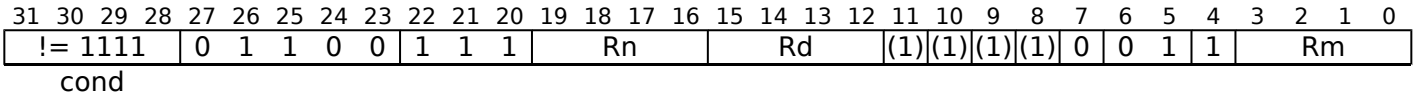
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UHASX

Unsigned Halving Add and Subtract with Exchange exchanges the two halfwords of the second operand, performs one unsigned 16-bit integer addition and one unsigned 16-bit subtraction, halves the results, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

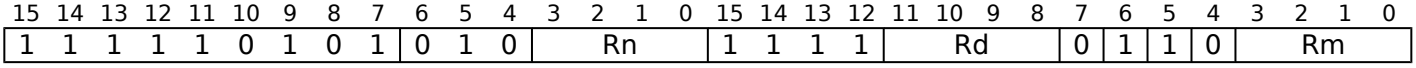
A1



A1

```
UHASX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1



T1

```
UHASX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff = UInt(R[n]<15:0>) - UInt(R[m]<31:16>);
    sum  = UInt(R[n]<31:16>) + UInt(R[m]<15:0>);
    R[d]<15:0>  = diff<16:1>;
    R[d]<31:16> = sum<16:1>;
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.

    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

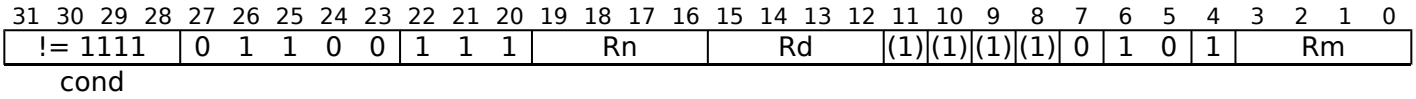
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UHSAX

Unsigned Halving Subtract and Add with Exchange exchanges the two halfwords of the second operand, performs one unsigned 16-bit integer subtraction and one unsigned 16-bit addition, halves the results, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

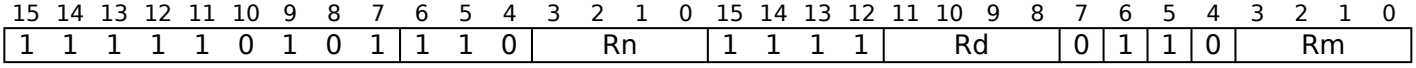
A1



A1

```
UHSAX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1



T1

```
UHSAX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum  = UInt(R[n]<15:0>) + UInt(R[m]<31:16>);
    diff = UInt(R[n]<31:16>) - UInt(R[m]<15:0>);
    R[d]<15:0>  = sum<16:1>;
    R[d]<31:16> = diff<16:1>;
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.

    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

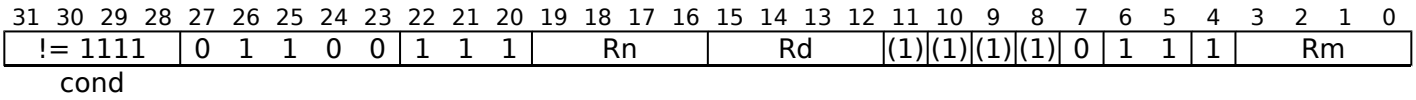
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UHSUB16

Unsigned Halving Subtract 16 performs two unsigned 16-bit integer subtractions, halves the results, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

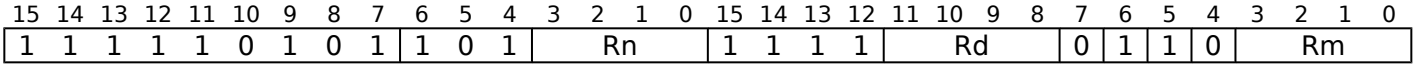
A1



A1

```
UHSUB16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1



T1

```
UHSUB16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = UInt(R[n]<15:0>) - UInt(R[m]<15:0>);
    diff2 = UInt(R[n]<31:16>) - UInt(R[m]<31:16>);
    R[d]<15:0>  = diff1<16:1>;
    R[d]<31:16> = diff2<16:1>;
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

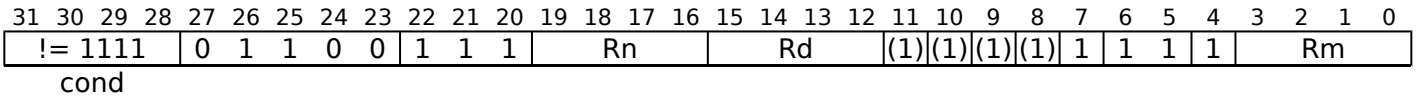
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UHSUB8

Unsigned Halving Subtract 8 performs four unsigned 8-bit integer subtractions, halves the results, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

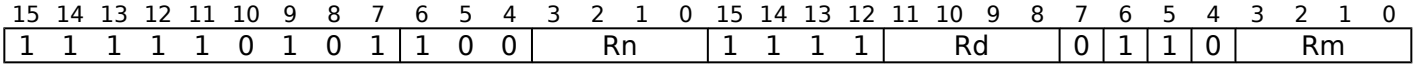
A1



A1

```
UHSUB8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1



T1

```
UHSUB8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = UInt(R[n]<7:0>) - UInt(R[m]<7:0>);
    diff2 = UInt(R[n]<15:8>) - UInt(R[m]<15:8>);
    diff3 = UInt(R[n]<23:16>) - UInt(R[m]<23:16>);
    diff4 = UInt(R[n]<31:24>) - UInt(R[m]<31:24>);
    R[d]<7:0>   = diff1<8:1>;
    R[d]<15:8>  = diff2<8:1>;
    R[d]<23:16> = diff3<8:1>;
    R[d]<31:24> = diff4<8:1>;
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

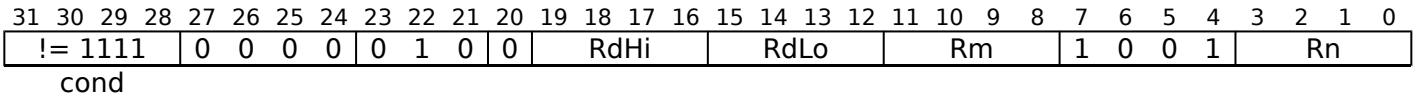
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UMAAL

Unsigned Multiply Accumulate Accumulate Long multiplies two unsigned 32-bit values to produce a 64-bit value, adds two unsigned 32-bit values, and writes the 64-bit result to two registers.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



A1

```
UMAAL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

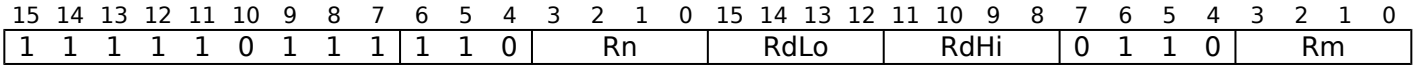
```
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
if dHi == dLo then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `dHi == dLo`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The value in the destination register is UNKNOWN.

T1



T1

```
UMAAL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

```
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
// Armv8-A removes UNPREDICTABLE for R13
if dHi == dLo then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `dHi == dLo`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The value in the destination register is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<RdLo>	Is the general-purpose source register holding the first addend and the destination register for the lower 32 bits of the result, encoded in the "RdLo" field.

<RdHi> Is the general-purpose source register holding the second addend and the destination register for the upper 32 bits of the result, encoded in the "RdHi" field.

<Rn> Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

<Rm> Is the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = UInt(R[n]) * UInt(R[m]) + UInt(R[dHi]) + UInt(R[dLo]);
    R[dHi] = result<63:32>;
    R[dLo] = result<31:0>;
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

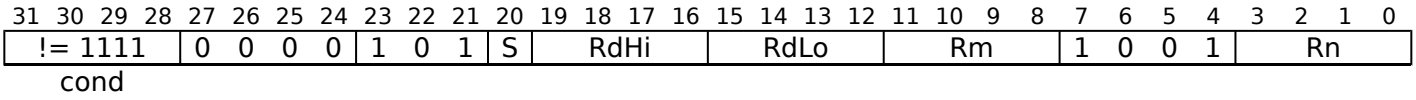
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UMLAL, UMLALS

Unsigned Multiply Accumulate Long multiplies two unsigned 32-bit values to produce a 64-bit value, and accumulates this with a 64-bit value.
In A32 instructions, the condition flags can optionally be updated based on the result. Use of this option adversely affects performance on many implementations.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



Flag setting (S == 1)

```
UMLALS{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

Not flag setting (S == 0)

```
UMLAL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

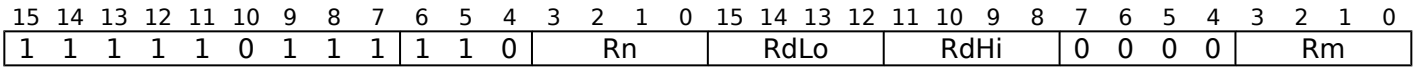
```
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
if dHi == dLo then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `dHi == dLo`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

T1



T1

```
UMLAL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

```
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);  setflags = FALSE;
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
// Armv8-A removes UNPREDICTABLE for R13
if dHi == dLo then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `dHi == dLo`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<RdLo>	Is the general-purpose source register holding the lower 32 bits of the addend, and the destination register for the lower 32 bits of the result, encoded in the "RdLo" field.
<RdHi>	Is the general-purpose source register holding the upper 32 bits of the addend, and the destination register for the upper 32 bits of the result, encoded in the "RdHi" field.
<Rn>	Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = UInt(R[n]) * UInt(R[m]) + UInt(R[dHi]:R[dLo]);
    R[dHi] = result<63:32>;
    R[dLo] = result<31:0>;
    if setflags then
        PSTATE.N = result<63>;
        PSTATE.Z = IsZeroBit(result<63:0>);
        // PSTATE.C, PSTATE.V unchanged
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

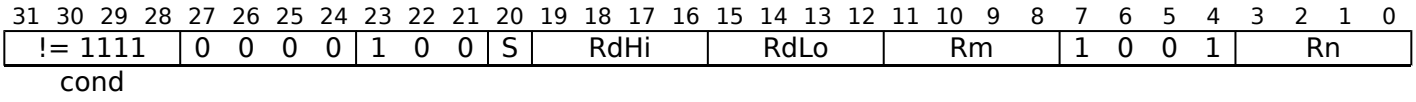
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UMULL, UMULLS

Unsigned Multiply Long multiplies two 32-bit unsigned values to produce a 64-bit result.
In A32 instructions, the condition flags can optionally be updated based on the result. Use of this option adversely affects performance on many implementations.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



Flag setting (S == 1)

```
UMULLS{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

Not flag setting (S == 0)

```
UMULL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

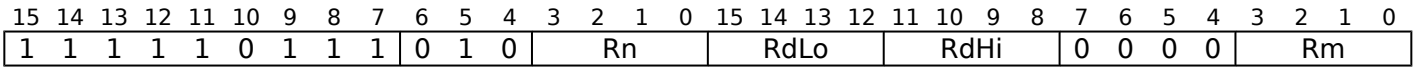
```
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
if dHi == dLo then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `dHi == dLo`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

T1



T1

```
UMULL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

```
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);  setflags = FALSE;
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
// Armv8-A removes UNPREDICTABLE for R13
if dHi == dLo then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `dHi == dLo`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<RdLo>	Is the general-purpose destination register for the lower 32 bits of the result, encoded in the "RdLo" field.
<RdHi>	Is the general-purpose destination register for the upper 32 bits of the result, encoded in the "RdHi" field.
<Rn>	Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = UInt(R[n]) * UInt(R[m]);
    R[dHi] = result<63:32>;
    R[dLo] = result<31:0>;
    if setflags then
        PSTATE.N = result<63>;
        PSTATE.Z = IsZeroBit(result<63:0>);
        // PSTATE.C, PSTATE.V unchanged
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

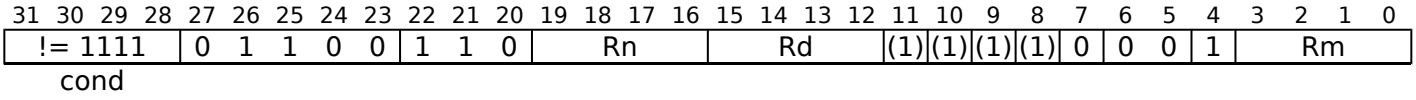
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UQADD16

Unsigned Saturating Add 16 performs two unsigned 16-bit integer additions, saturates the results to the 16-bit unsigned integer range 0 <= x <= 2^{16} - 1, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

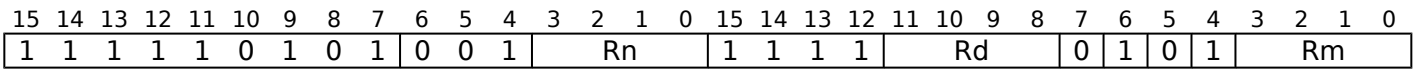
A1



A1

```
UQADD16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1



T1

```
UQADD16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = UInt(R[n]<15:0>) + UInt(R[m]<15:0>);
    sum2 = UInt(R[n]<31:16>) + UInt(R[m]<31:16>);
    R[d]<15:0>  = UnsignedSat(sum1, 16);
    R[d]<31:16> = UnsignedSat(sum2, 16);
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

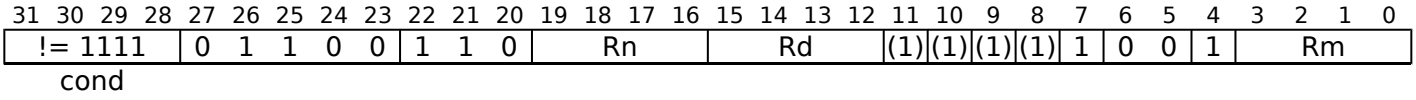
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UQADD8

Unsigned Saturating Add 8 performs four unsigned 8-bit integer additions, saturates the results to the 8-bit unsigned integer range 0 <= x <= 2^8 - 1, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

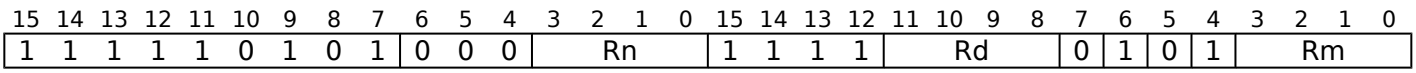
A1



A1

```
UQADD8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1



T1

```
UQADD8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = UInt(R[n]<7:0>) + UInt(R[m]<7:0>);
    sum2 = UInt(R[n]<15:8>) + UInt(R[m]<15:8>);
    sum3 = UInt(R[n]<23:16>) + UInt(R[m]<23:16>);
    sum4 = UInt(R[n]<31:24>) + UInt(R[m]<31:24>);
    R[d]<7:0>   = UnsignedSat(sum1, 8);
    R[d]<15:8>  = UnsignedSat(sum2, 8);
    R[d]<23:16> = UnsignedSat(sum3, 8);
    R[d]<31:24> = UnsignedSat(sum4, 8);
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

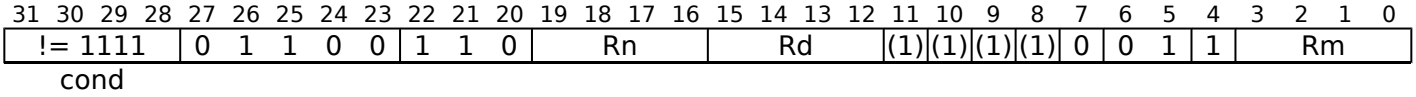
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UQASX

Unsigned Saturating Add and Subtract with Exchange exchanges the two halfwords of the second operand, performs one unsigned 16-bit integer addition and one unsigned 16-bit subtraction, saturates the results to the 16-bit unsigned integer range 0 <= x <= 2^{16} - 1, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

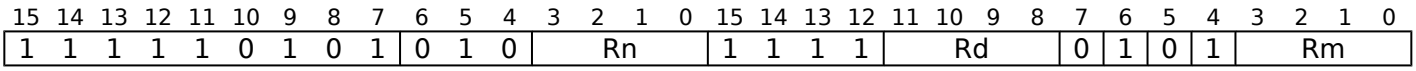


A1

```
UQASX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1



T1

```
UQASX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff = UInt(R[n]<15:0>) - UInt(R[m]<31:16>);
    sum  = UInt(R[n]<31:16>) + UInt(R[m]<15:0>);
    R[d]<15:0>  = UnsignedSat(diff, 16);
    R[d]<31:16> = UnsignedSat(sum, 16);
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

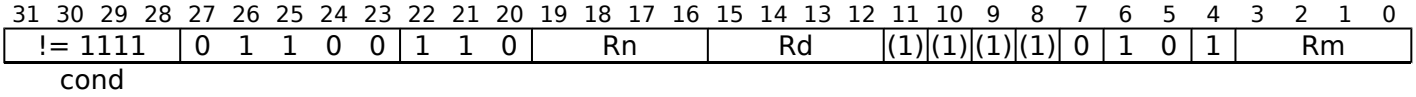
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UQSAX

Unsigned Saturating Subtract and Add with Exchange exchanges the two halfwords of the second operand, performs one unsigned 16-bit integer subtraction and one unsigned 16-bit addition, saturates the results to the 16-bit unsigned integer range 0 <= x <= 2^{16} - 1, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

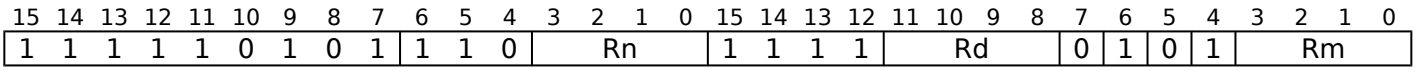


A1

```
UQSAX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1



T1

```
UQSAX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum  = UInt(R[n]<15:0>) + UInt(R[m]<31:16>);
    diff = UInt(R[n]<31:16>) - UInt(R[m]<15:0>);
    R[d]<15:0>  = UnsignedSat(sum, 16);
    R[d]<31:16> = UnsignedSat(diff, 16);
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

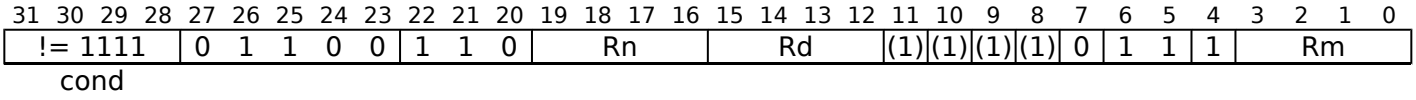
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UQSUB16

Unsigned Saturating Subtract 16 performs two unsigned 16-bit integer subtractions, saturates the results to the 16-bit unsigned integer range 0 <= x <= 2^{16} - 1, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

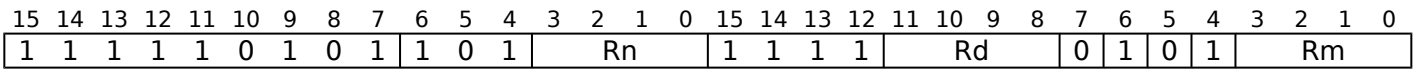
A1



A1

```
UQSUB16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1



T1

```
UQSUB16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = UInt(R[n]<15:0>) - UInt(R[m]<15:0>);
    diff2 = UInt(R[n]<31:16>) - UInt(R[m]<31:16>);
    R[d]<15:0>  = UnsignedSat(diff1, 16);
    R[d]<31:16> = UnsignedSat(diff2, 16);
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

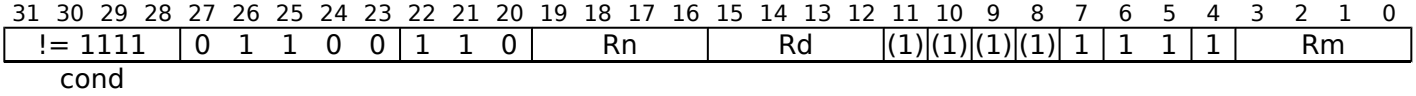
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UQSUB8

Unsigned Saturating Subtract 8 performs four unsigned 8-bit integer subtractions, saturates the results to the 8-bit unsigned integer range 0 <= x <= 2^8 - 1, and writes the results to the destination register.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

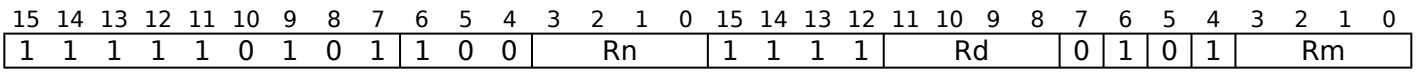
A1



A1

```
UQSUB8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1



T1

```
UQSUB8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = UInt(R[n]<7:0>) - UInt(R[m]<7:0>);
    diff2 = UInt(R[n]<15:8>) - UInt(R[m]<15:8>);
    diff3 = UInt(R[n]<23:16>) - UInt(R[m]<23:16>);
    diff4 = UInt(R[n]<31:24>) - UInt(R[m]<31:24>);
    R[d]<7:0>   = UnsignedSat(diff1, 8);
    R[d]<15:8>  = UnsignedSat(diff2, 8);
    R[d]<23:16> = UnsignedSat(diff3, 8);
    R[d]<31:24> = UnsignedSat(diff4, 8);
```

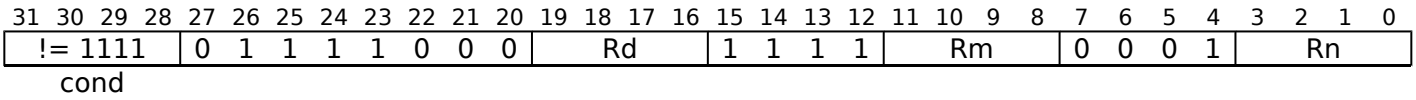

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

USAD8

Unsigned Sum of Absolute Differences performs four unsigned 8-bit subtractions, and adds the absolute values of the differences together.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

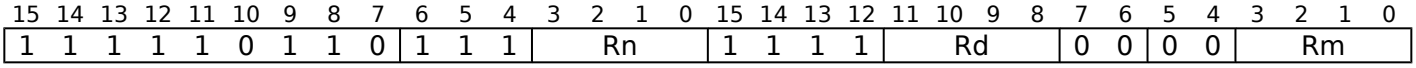
A1



A1

```
USAD8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1



T1

```
USAD8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    absdiff1 = Abs(UInt(R[n]<7:0>)   - UInt(R[m]<7:0>));
    absdiff2 = Abs(UInt(R[n]<15:8>)  - UInt(R[m]<15:8>));
    absdiff3 = Abs(UInt(R[n]<23:16>) - UInt(R[m]<23:16>));
    absdiff4 = Abs(UInt(R[n]<31:24>) - UInt(R[m]<31:24>));
    result = absdiff1 + absdiff2 + absdiff3 + absdiff4;
    R[d] = result<31:0>;
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

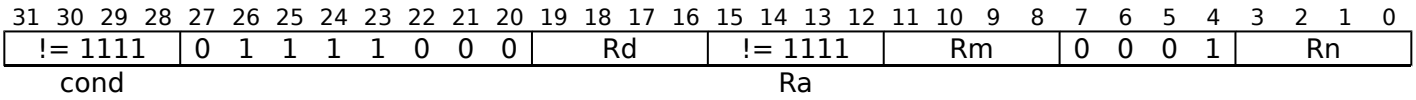
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

USADA8

Unsigned Sum of Absolute Differences and Accumulate performs four unsigned 8-bit subtractions, and adds the absolute values of the differences to a 32-bit accumulate operand.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

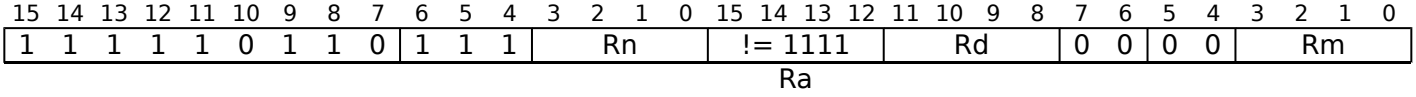


A1

```
USADA8{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

```
if Ra == '1111' then SEE "USAD8";
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1



T1

```
USADA8{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

```
if Ra == '1111' then SEE "USAD8";
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.
<Ra>	Is the third general-purpose source register holding the addend, encoded in the "Ra" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    absdiff1 = Abs(UInt(R[n]<7:0>)   - UInt(R[m]<7:0>));
    absdiff2 = Abs(UInt(R[n]<15:8>)  - UInt(R[m]<15:8>));
    absdiff3 = Abs(UInt(R[n]<23:16>) - UInt(R[m]<23:16>));
    absdiff4 = Abs(UInt(R[n]<31:24>) - UInt(R[m]<31:24>));
    result = UInt(R[a]) + absdiff1 + absdiff2 + absdiff3 + absdiff4;
    R[d] = result<31:0>;
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

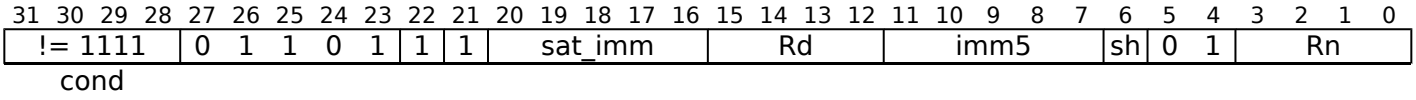
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

USAT

Unsigned Saturate saturates an optionally-shifted signed value to a selected unsigned range. This instruction sets *PSTATE*.Q to 1 if the operation saturates.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



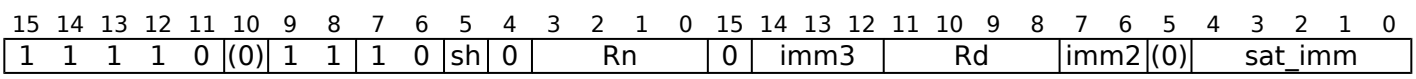
Arithmetic shift right (sh == 1)

```
USAT{<c>}{<q>} <Rd>, #<imm>, <Rn>, ASR #<amount>
```

Logical shift left (sh == 0)

```
USAT{<c>}{<q>} <Rd>, #<imm>, <Rn> {, LSL #<amount>}

d = UInt(Rd);  n = UInt(Rn);  saturate_to = UInt(sat_imm);
(shift_t, shift_n) = DecodeImmShift(sh:'0', imm5);
if d == 15 || n == 15 then UNPREDICTABLE;
```

T1



Arithmetic shift right (sh == 1 && !(imm3 == 000 && imm2 == 00))

```
USAT{<c>}{<q>} <Rd>, #<imm>, <Rn>, ASR #<amount>
```

Logical shift left (sh == 0)

```
USAT{<c>}{<q>} <Rd>, #<imm>, <Rn> {, LSL #<amount>}

if sh == '1' && (imm3:imm2) == '00000' then SEE "USAT16";
d = UInt(Rd);  n = UInt(Rn);  saturate_to = UInt(sat_imm);
(shift_t, shift_n) = DecodeImmShift(sh:'0', imm3:imm2);
if d == 15 || n == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<imm>	Is the bit position for saturation, in the range 0 to 31, encoded in the "sat_imm" field.
<Rn>	Is the general-purpose source register, encoded in the "Rn" field.
<amount>	For encoding A1: is the optional shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm5" field. For encoding A1: is the shift amount, in the range 1 to 32 encoded in the "imm5" field as <amount> modulo 32.

For encoding T1: is the optional shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm3:imm2" field.

For encoding T1: is the shift amount, in the range 1 to 31 encoded in the "imm3:imm2" field as <amount>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand = Shift(R[n], shift_t, shift_n, PSTATE.C);  // PSTATE.C ignored
    (result, sat) = UnsignedSatQ(SInt(operand), saturate_to);
    R[d] = ZeroExtend(result, 32);
    if sat then
        PSTATE.Q = '1';
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

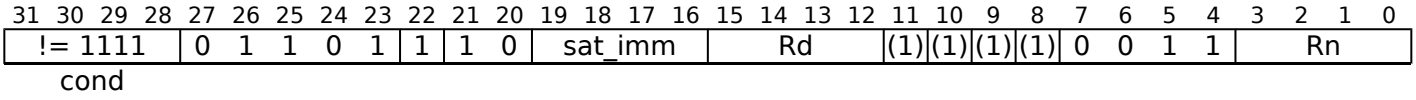
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

USAT16

Unsigned Saturate 16 saturates two signed 16-bit values to a selected unsigned range.
This instruction sets *PSTATE*.Q to 1 if the operation saturates.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

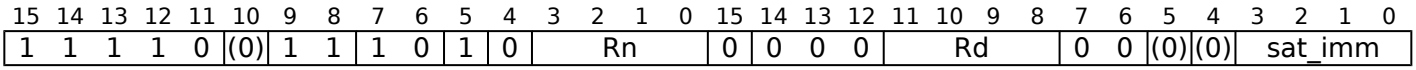


A1

```
USAT16{<c>}{<q>} <Rd>, #<imm>, <Rn>
```

```
d = UInt(Rd);  n = UInt(Rn);  saturate_to = UInt(sat_imm);
if d == 15 || n == 15 then UNPREDICTABLE;
```

T1



T1

```
USAT16{<c>}{<q>} <Rd>, #<imm>, <Rn>
```

```
d = UInt(Rd);  n = UInt(Rn);  saturate_to = UInt(sat_imm);
if d == 15 || n == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<imm>	Is the bit position for saturation, in the range 0 to 15, encoded in the "sat_imm" field.
<Rn>	Is the general-purpose source register, encoded in the "Rn" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result1, sat1) = UnsignedSatQ(SInt(R[n]<15:0>), saturate_to);
    (result2, sat2) = UnsignedSatQ(SInt(R[n]<31:16>), saturate_to);
    R[d]<15:0> = ZeroExtend(result1, 16);
    R[d]<31:16> = ZeroExtend(result2, 16);
    if sat1 || sat2 then
        PSTATE.Q = '1';
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

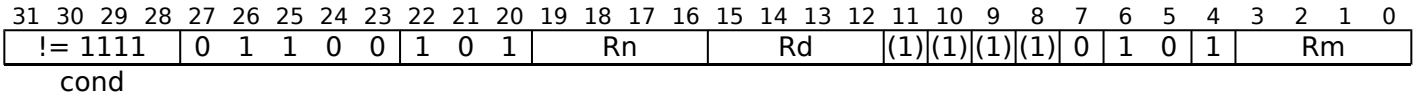
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

USAX

Unsigned Subtract and Add with Exchange exchanges the two halfwords of the second operand, performs one unsigned 16-bit integer subtraction and one unsigned 16-bit addition, and writes the results to the destination register. It sets *PSTATE*.GE according to the results.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

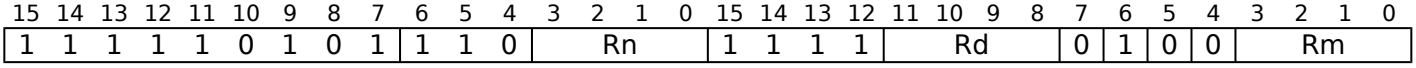
A1



A1

```
USAX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1



T1

```
USAX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum  = UInt(R[n]<15:0>) + UInt(R[m]<31:16>);
    diff = UInt(R[n]<31:16>) - UInt(R[m]<15:0>);
    R[d]<15:0>  = sum<15:0>;
    R[d]<31:16> = diff<15:0>;
    PSTATE.GE<1:0> = if sum  >= 0x10000 then '11' else '00';
    PSTATE.GE<3:2> = if diff >= 0 then '11' else '00';
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

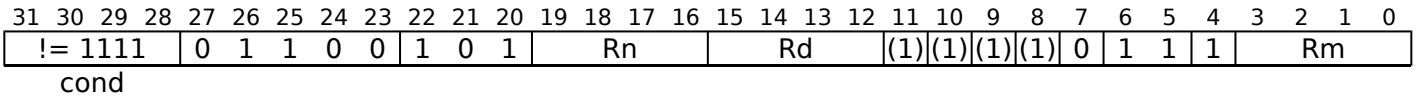
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

USUB16

Unsigned Subtract 16 performs two 16-bit unsigned integer subtractions, and writes the results to the destination register. It sets *PSTATE*.GE according to the results of the subtractions.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

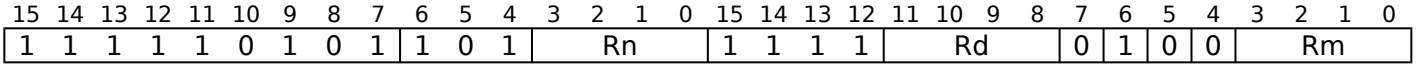
A1



A1

```
USUB16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1



T1

```
USUB16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = UInt(R[n]<15:0>) - UInt(R[m]<15:0>);
    diff2 = UInt(R[n]<31:16>) - UInt(R[m]<31:16>);
    R[d]<15:0>  = diff1<15:0>;
    R[d]<31:16> = diff2<15:0>;
    PSTATE.GE<1:0> = if diff1 >= 0 then '11' else '00';
    PSTATE.GE<3:2> = if diff2 >= 0 then '11' else '00';
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:

    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

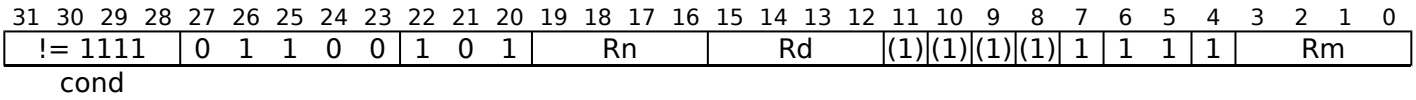
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

USUB8

Unsigned Subtract 8 performs four 8-bit unsigned integer subtractions, and writes the results to the destination register. It sets *PSTATE*.GE according to the results of the subtractions.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

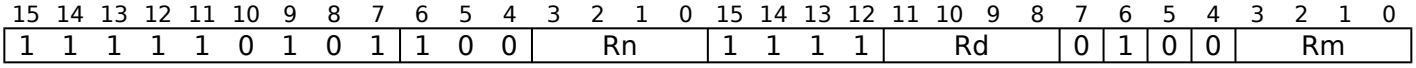
A1



A1

```
USUB8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1



T1

```
USUB8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = UInt(R[n]<7:0>) - UInt(R[m]<7:0>);
    diff2 = UInt(R[n]<15:8>) - UInt(R[m]<15:8>);
    diff3 = UInt(R[n]<23:16>) - UInt(R[m]<23:16>);
    diff4 = UInt(R[n]<31:24>) - UInt(R[m]<31:24>);
    R[d]<7:0>   = diff1<7:0>;
    R[d]<15:8>  = diff2<7:0>;
    R[d]<23:16> = diff3<7:0>;
    R[d]<31:24> = diff4<7:0>;
    PSTATE.GE<0>  = if diff1 >= 0 then '1' else '0';
    PSTATE.GE<1>  = if diff2 >= 0 then '1' else '0';
    PSTATE.GE<2>  = if diff3 >= 0 then '1' else '0';
    PSTATE.GE<3>  = if diff4 >= 0 then '1' else '0';
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

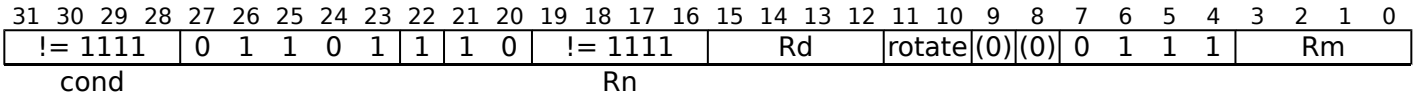
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UXTAB

Unsigned Extend and Add Byte extracts an 8-bit value from a register, zero-extends it to 32 bits, adds the result to the value in another register, and writes the final result to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit value.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

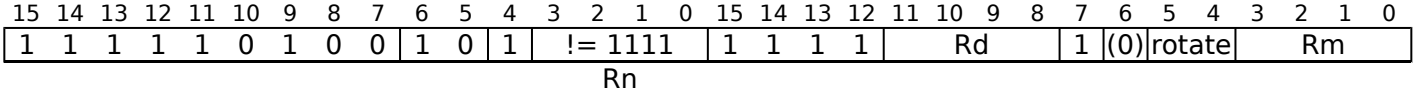


A1

```
UXTAB{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, ROR #<amount>}
```

```
if Rn == '1111' then SEE "UXTB";
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE;
```

T1



T1

```
UXTAB{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, ROR #<amount>}
```

```
if Rn == '1111' then SEE "UXTB";
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<Rd> Is the general-purpose destination register, encoded in the "Rd" field.

<Rn> Is the first general-purpose source register, encoded in the "Rn" field.

<Rm> Is the second general-purpose source register, encoded in the "Rm" field.

<amount> Is the rotate amount, encoded in “rotate”:

rotate	<amount>
00	(omitted)
01	8
10	16
11	24

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d] = R[n] + ZeroExtend(rotated<7:0>, 32);
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

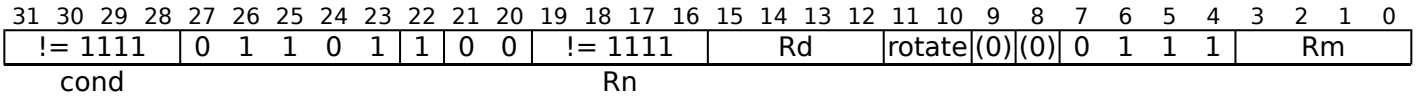
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UXTAB16

Unsigned Extend and Add Byte 16 extracts two 8-bit values from a register, zero-extends them to 16 bits each, adds the results to two 16-bit values from another register, and writes the final results to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit values.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

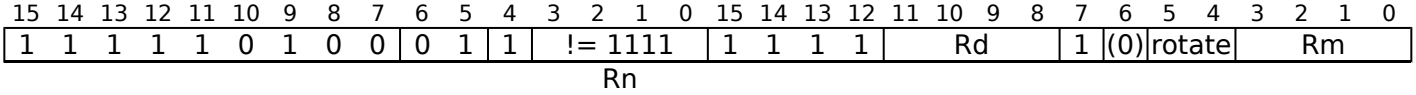


A1

```
UXTAB16{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, ROR #<amount>}
```

```
if Rn == '1111' then SEE "UXTB16";
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE;
```

T1



T1

```
UXTAB16{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, ROR #<amount>}
```

```
if Rn == '1111' then SEE "UXTB16";
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<Rd> Is the general-purpose destination register, encoded in the "Rd" field.

<Rn> Is the first general-purpose source register, encoded in the "Rn" field.

<Rm> Is the second general-purpose source register, encoded in the "Rm" field.

<amount> Is the rotate amount, encoded in “rotate”:

rotate	<amount>
00	(omitted)
01	8
10	16
11	24

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d]<15:0>  = R[n]<15:0> + ZeroExtend(rotated<7:0>, 16);
    R[d]<31:16> = R[n]<31:16> + ZeroExtend(rotated<23:16>, 16);
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

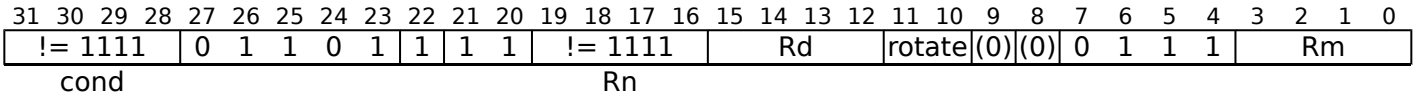
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UXTAH

Unsigned Extend and Add Halfword extracts a 16-bit value from a register, zero-extends it to 32 bits, adds the result to a value from another register, and writes the final result to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 16-bit value.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

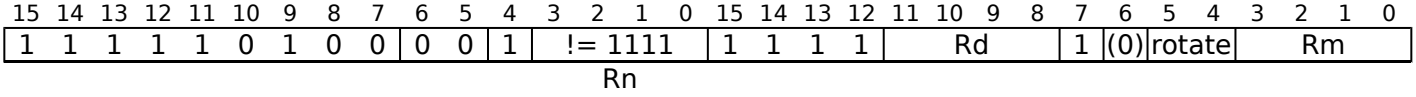


A1

```
UXTAH{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, ROR #<amount>}
```

```
if Rn == '1111' then SEE "UXTH";
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE;
```

T1



T1

```
UXTAH{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, ROR #<amount>}
```

```
if Rn == '1111' then SEE "UXTH";
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<Rd> Is the general-purpose destination register, encoded in the "Rd" field.

<Rn> Is the first general-purpose source register, encoded in the "Rn" field.

<Rm> Is the second general-purpose source register, encoded in the "Rm" field.

<amount> Is the rotate amount, encoded in “rotate”:

rotate	<amount>
00	(omitted)
01	8
10	16
11	24

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d] = R[n] + ZeroExtend(rotated<15:0>, 32);
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

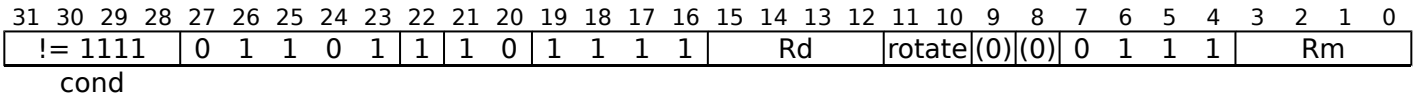
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UXTB

Unsigned Extend Byte extracts an 8-bit value from a register, zero-extends it to 32 bits, and writes the result to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit value.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2) .

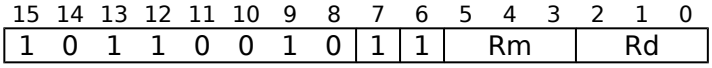
A1



A1

```
UXTB{<c>}{<q>} {<Rd>,} <Rm> {, ROR #<amount>}

d = UInt(Rd);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE;
```

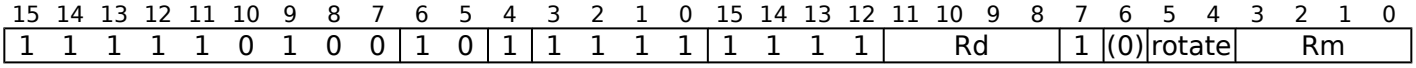
T1



T1

```
UXTB{<c>}{<q>} {<Rd>,} <Rm>

d = UInt(Rd);  m = UInt(Rm);  rotation = 0;
```

T2



T2

```
UXTB{<c>}.W {<Rd>,} <Rm> // (<Rd>, <Rm> can be represented in T1)

UXTB{<c>}{<q>} {<Rd>,} <Rm> {, ROR #<amount>}

d = UInt(Rd);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rm>	Is the general-purpose source register, encoded in the "Rm" field.
<amount>	Is the rotate amount, encoded in “rotate”:

rotate	<amount>
00	(omitted)
01	8
10	16
11	24

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d] = ZeroExtend(rotated<7:0>, 32);
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

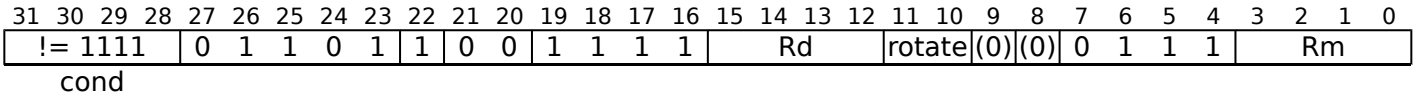
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UXTB16

Unsigned Extend Byte 16 extracts two 8-bit values from a register, zero-extends them to 16 bits each, and writes the results to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit values.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

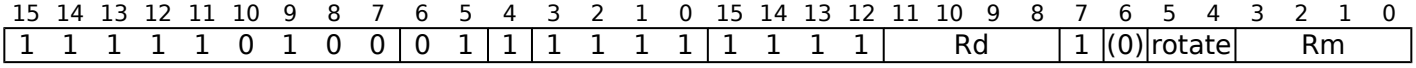
A1



A1

```
UXTB16{<c>}{<q>} {<Rd>,} <Rm> {, ROR #<amount>}

d = UInt(Rd);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE;
```

T1



T1

```
UXTB16{<c>}{<q>} {<Rd>,} <Rm> {, ROR #<amount>}

d = UInt(Rd);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<Rd> Is the general-purpose destination register, encoded in the "Rd" field.

<Rm> For encoding A1: is the general-purpose source register, encoded in the "Rm" field.

For encoding T1: is the second general-purpose source register, encoded in the "Rm" field.

<amount> Is the rotate amount, encoded in “rotate”:

rotate	<amount>
00	(omitted)
01	8
10	16
11	24

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d]<15:0>  = ZeroExtend(rotated<7:0>, 16);
    R[d]<31:16> = ZeroExtend(rotated<23:16>, 16);
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

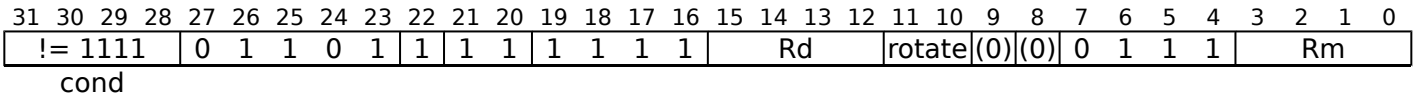
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UXTH

Unsigned Extend Halfword extracts a 16-bit value from a register, zero-extends it to 32 bits, and writes the result to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 16-bit value.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2) .

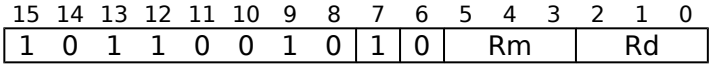
A1



A1

```
UXTH{<c>}{<q>} {<Rd>,} <Rm> {, ROR #<amount>}

d = UInt(Rd);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE;
```

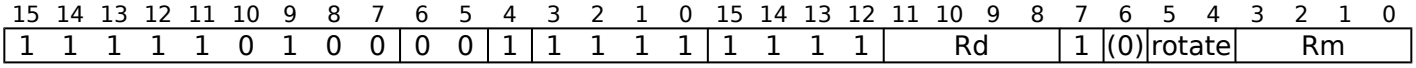
T1



T1

```
UXTH{<c>}{<q>} {<Rd>,} <Rm>

d = UInt(Rd);  m = UInt(Rm);  rotation = 0;
```

T2



T2

```
UXTH{<c>}.W {<Rd>,} <Rm> // (<Rd>, <Rm> can be represented in T1)

UXTH{<c>}{<q>} {<Rd>,} <Rm> {, ROR #<amount>}

d = UInt(Rd);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rm>	Is the general-purpose source register, encoded in the "Rm" field.
<amount>	Is the rotate amount, encoded in “rotate”:

rotate	<amount>
00	(omitted)
01	8
10	16
11	24

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d] = ZeroExtend(rotated<15:0>, 32);
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

WFE

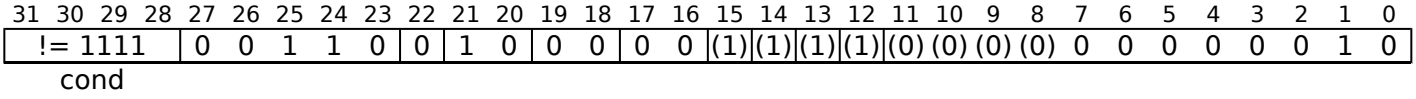
Wait For Event is a hint instruction that indicates that the PE can enter a low-power state and remain there until a wakeup event occurs. Wakeup events include the event signaled as a result of executing the SEV instruction on any PE in the multiprocessor system. For more information, see *Wait For Event and Send Event*.

As described in *Wait For Event and Send Event*, the execution of a WFE instruction that would otherwise cause entry to a low-power state can be trapped to a higher Exception level, see:

- *Traps to Undefined mode of PL0 execution of WFE and WFI instructions*.
- *Traps to Hyp mode of Non-secure EL0 and EL1 execution of WFE and WFI instructions*.
- *Traps to Monitor mode of the execution of WFE and WFI instructions in modes other than Monitor mode*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2) .

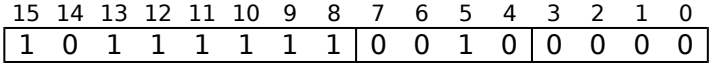
A1



A1

```
WFE{<c>}{<q>}

// No additional decoding required
```

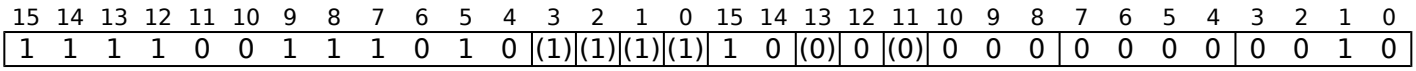
T1



T1

```
WFE{<c>}{<q>}

// No additional decoding required
```

T2



T2

```
WFE{<c>}.W

// No additional decoding required
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if IsEventRegisterSet() then
        ClearEventRegister();
    else
        if PSTATE.EL == EL0 then
            // Check for traps described by the OS.
            AArch32.CheckForWFxTrap(EL1, WFxType_WFE);
        if PSTATE.EL IN {EL0, EL1} && EL2Enabled() && !IsInHost() then
            // Check for traps described by the Hypervisor.
            AArch32.CheckForWFxTrap(EL2, WFxType_WFE);
        if HaveEL(EL3) && PSTATE.M != M32_Monitor then
            // Check for traps described by the Secure Monitor.
            AArch32.CheckForWFxTrap(EL3, WFxType_WFE);
        integer localtimeout = -1;  // No local timeout event is generated
        WaitForEvent(localtimeout);
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

WFI

Wait For Interrupt is a hint instruction that indicates that the PE can enter a low-power state and remain there until a wakeup event occurs. For more information, see *Wait For Interrupt*.

As described in *Wait For Interrupt*, the execution of a `WFI` instruction that would otherwise cause entry to a low-power state can be trapped to a higher Exception level, see:

- *Traps to Undefined mode of PL0 execution of WFE and WFI instructions*.
- *Traps to Hyp mode of Non-secure EL0 and EL1 execution of WFE and WFI instructions*.
- *Traps to Monitor mode of the execution of WFE and WFI instructions in modes other than Monitor mode*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	1	1	0	0	1	0	0	0	0	0	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	0	0	0	0	0	0	1	1
cond																															

A1

```
WFI{<c>}{<q>}

// No additional decoding required
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	0	0	1	1	0	0	0	0

T1

```
WFI{<c>}{<q>}

// No additional decoding required
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	0	(1)	(1)	(1)	(1)	1	0	(0)	0	(0)	0	0	0	0	0	0	0	0	0	1	1

T2

```
WFI{<c>}.W

// No additional decoding required
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if !InterruptPending() then
        if PSTATE.EL == EL0 then
            // Check for traps described by the OS.
            AArch32.CheckForWFxTrap(EL1, WFxType_WFI);
        if PSTATE.EL IN {EL0, EL1} && EL2Enabled() && !IsInHost() then
            // Check for traps described by the Hypervisor.
            AArch32.CheckForWFxTrap(EL2, WFxType_WFI);
        if HaveEL(EL3) && PSTATE.M != M32_Monitor then
            // Check for traps described by the Secure Monitor.
            AArch32.CheckForWFxTrap(EL3, WFxType_WFI);
        integer localtimeout = -1;  // No local timeout event is generated
        WaitForInterrupt(localtimeout);
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

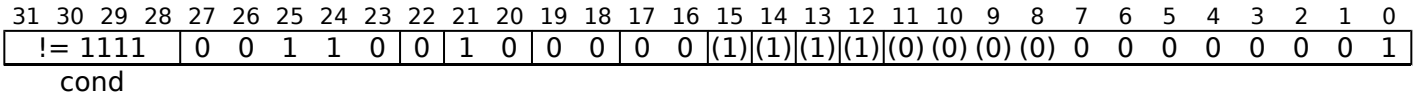
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

YIELD

YIELD is a hint instruction. Software with a multithreading capability can use a YIELD instruction to indicate to the PE that it is performing a task, for example a spin-lock, that could be swapped out to improve overall system performance. The PE can use this hint to suspend and resume multiple software threads if it supports the capability.
For more information about the recommended use of this instruction see *The Yield instruction*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 and T2) .

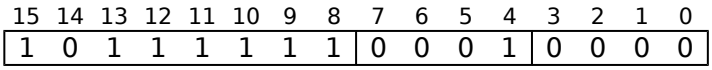
A1



A1

```
YIELD{<c>}{<q>}

// No additional decoding required
```

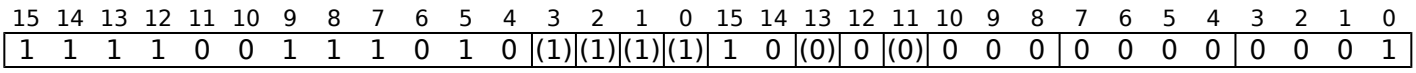
T1



T1

```
YIELD{<c>}{<q>}

// No additional decoding required
```

T2



T2

```
YIELD{<c>}.W

// No additional decoding required
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    Hint_Yield();
```

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

AArch32 -- SIMD&FP Instructions (alphabetic order)

AESD: AES single round decryption.

AESE: AES single round encryption.

AESIMC: AES inverse mix columns.

AESMC: AES mix columns.

FLDM*X (FLDMDBX, FLDMIAX): FLDM*X.

FSTMDBX, FSTMIAX: FSTMX.

SHA1C: SHA1 hash update (choose).

SHA1H: SHA1 fixed rotate.

SHA1M: SHA1 hash update (majority).

SHA1P: SHA1 hash update (parity).

SHA1SU0: SHA1 schedule update 0.

SHA1SU1: SHA1 schedule update 1.

SHA256H: SHA256 hash update part 1.

SHA256H2: SHA256 hash update part 2.

SHA256SU0: SHA256 schedule update 0.

SHA256SU1: SHA256 schedule update 1.

VABA: Vector Absolute Difference and Accumulate.

VABAL: Vector Absolute Difference and Accumulate Long.

VABD (floating-point): Vector Absolute Difference (floating-point).

VABD (integer): Vector Absolute Difference (integer).

VABDL (integer): Vector Absolute Difference Long (integer).

VABS: Vector Absolute.

VACGE: Vector Absolute Compare Greater Than or Equal.

VACGT: Vector Absolute Compare Greater Than.

VACLE: Vector Absolute Compare Less Than or Equal: an alias of VACGE.

VACLT: Vector Absolute Compare Less Than: an alias of VACGT.

VADD (floating-point): Vector Add (floating-point).

VADD (integer): Vector Add (integer).

VADDHN: Vector Add and Narrow, returning High Half.

VADDL: Vector Add Long.

VADDW: Vector Add Wide.

VAND (immediate): Vector Bitwise AND (immediate): an alias of VBIC (immediate).

VAND (register): Vector Bitwise AND (register).

VBIC (immediate): Vector Bitwise Bit Clear (immediate).

VBIC (register): Vector Bitwise Bit Clear (register).

VBIF: Vector Bitwise Insert if False.

VBIT: Vector Bitwise Insert if True.

VBSL: Vector Bitwise Select.

VCADD: Vector Complex Add.

VCEQ (immediate #0): Vector Compare Equal to Zero.

VCEQ (register): Vector Compare Equal.

VCGE (immediate #0): Vector Compare Greater Than or Equal to Zero.

VCGE (register): Vector Compare Greater Than or Equal.

VCGT (immediate #0): Vector Compare Greater Than Zero.

VCGT (register): Vector Compare Greater Than.

VCLE (immediate #0): Vector Compare Less Than or Equal to Zero.

VCLE (register): Vector Compare Less Than or Equal: an alias of VCGE (register).

VCLS: Vector Count Leading Sign Bits.

VCLT (immediate #0): Vector Compare Less Than Zero.

VCLT (register): Vector Compare Less Than: an alias of VCGT (register).

VCLZ: Vector Count Leading Zeros.

VCMLA: Vector Complex Multiply Accumulate.

VCMLA (by element): Vector Complex Multiply Accumulate (by element).

VCMP: Vector Compare.

VCMPE: Vector Compare, raising Invalid Operation on NaN.

VCNT: Vector Count Set Bits.

VCVT (between double-precision and single-precision): Convert between double-precision and single-precision.

VCVT (between floating-point and fixed-point, Advanced SIMD): Vector Convert between floating-point and fixed-point.

VCVT (between floating-point and fixed-point, floating-point): Convert between floating-point and fixed-point.

VCVT (between floating-point and integer, Advanced SIMD): Vector Convert between floating-point and integer.

VCVT (between half-precision and single-precision, Advanced SIMD): Vector Convert between half-precision and single-precision.

VCVT (floating-point to integer, floating-point): Convert floating-point to integer with Round towards Zero.

VCVT (from single-precision to BFloat16, Advanced SIMD): Vector Convert from single-precision to BFloat16.

VCVT (integer to floating-point, floating-point): Convert integer to floating-point.

VCVTA (Advanced SIMD): Vector Convert floating-point to integer with Round to Nearest with Ties to Away.

VCVTA (floating-point): Convert floating-point to integer with Round to Nearest with Ties to Away.

VCVTB: Convert to or from a half-precision value in the bottom half of a single-precision register.

VCVTB (BFloat16): Converts from a single-precision value to a BFloat16 value in the bottom half of a single-precision register.

VCVTM (Advanced SIMD): Vector Convert floating-point to integer with Round towards -Infinity.

VCVTM (floating-point): Convert floating-point to integer with Round towards -Infinity.

VCVTN (Advanced SIMD): Vector Convert floating-point to integer with Round to Nearest.

VCVTN (floating-point): Convert floating-point to integer with Round to Nearest.

VCVTP (Advanced SIMD): Vector Convert floating-point to integer with Round towards +Infinity.

VCVTP (floating-point): Convert floating-point to integer with Round towards +Infinity.

VCVTR: Convert floating-point to integer.

VCVTT: Convert to or from a half-precision value in the top half of a single-precision register.

VCVTT (BFloat16): Converts from a single-precision value to a BFloat16 value in the top half of a single-precision register..

VDIV: Divide.

VDOT (by element): BFloat16 floating-point indexed dot product (vector, by element).

VDOT (vector): BFloat16 floating-point (BF16) dot product (vector).

VDUP (general-purpose register): Duplicate general-purpose register to vector.

VDUP (scalar): Duplicate vector element to vector.

VEOR: Vector Bitwise Exclusive OR.

VEXT (byte elements): Vector Extract.

VEXT (multibyte elements): Vector Extract: an alias of VEXT (byte elements).

VFMA: Vector Fused Multiply Accumulate.

VFMAB, VFMAT (BFloat16, by scalar): BFloat16 floating-point widening multiply-add long (by scalar).

VFMAB, VFMAT (BFloat16, vector): BFloat16 floating-point widening multiply-add long (vector).

VFMAL (by scalar): Vector Floating-point Multiply-Add Long to accumulator (by scalar).

VFMAL (vector): Vector Floating-point Multiply-Add Long to accumulator (vector).

VFMS: Vector Fused Multiply Subtract.

VFMSL (by scalar): Vector Floating-point Multiply-Subtract Long from accumulator (by scalar).

VFMSL (vector): Vector Floating-point Multiply-Subtract Long from accumulator (vector).

VFNMA: Vector Fused Negate Multiply Accumulate.

VFNMS: Vector Fused Negate Multiply Subtract.

VHADD: Vector Halving Add.

VHSUB: Vector Halving Subtract.

VINS: Vector move Insertion.

VJCVT: Javascript Convert to signed fixed-point, rounding toward Zero.

VLD1 (multiple single elements): Load multiple single 1-element structures to one, two, three, or four registers.

VLD1 (single element to all lanes): Load single 1-element structure and replicate to all lanes of one register.

VLD1 (single element to one lane): Load single 1-element structure to one lane of one register.

VLD2 (multiple 2-element structures): Load multiple 2-element structures to two or four registers.

VLD2 (single 2-element structure to all lanes): Load single 2-element structure and replicate to all lanes of two registers.

VLD2 (single 2-element structure to one lane): Load single 2-element structure to one lane of two registers.

VLD3 (multiple 3-element structures): Load multiple 3-element structures to three registers.

VLD3 (single 3-element structure to all lanes): Load single 3-element structure and replicate to all lanes of three registers.

VLD3 (single 3-element structure to one lane): Load single 3-element structure to one lane of three registers.

VLD4 (multiple 4-element structures): Load multiple 4-element structures to four registers.

VLD4 (single 4-element structure to all lanes): Load single 4-element structure and replicate to all lanes of four registers.

VLD4 (single 4-element structure to one lane): Load single 4-element structure to one lane of four registers.

VLDM, VLDMDB, VLDMIA: Load Multiple SIMD&FP registers.

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.

VRSQRTE: 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_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

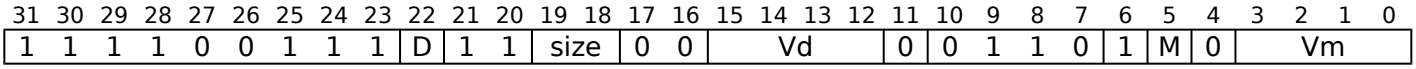
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

AESD

AES single round decryption.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

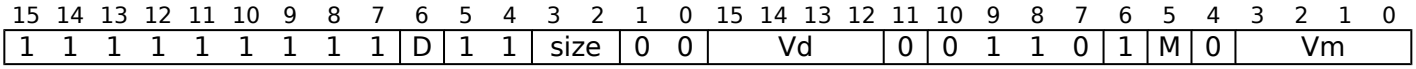


A1

```
AESD.<dt> <Qd>, <Qm>
```

```
if !HaveAESExt() then UNDEFINED;
if size != '00' then UNDEFINED;
if Vd<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); m = UInt(M:Vm);
```

T1



T1

```
AESD.<dt> <Qd>, <Qm>
```

```
if InITBlock() then UNPREDICTABLE;
if !HaveAESExt() then UNDEFINED;
if size != '00' then UNDEFINED;
if Vd<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); m = UInt(M:Vm);
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<dt> Is the data type, encoded in “size”:

size	<dt>
00	8
01	RESERVED
1x	RESERVED

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckCryptoEnabled32();
    op1 = Q[d>>1]; op2 = Q[m>>1];
    Q[d>>1] = AESInvSubBytes(AESInvShiftRows(op1 EOR op2));
```

Operational information

If CPSR.DIT is 1:

- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

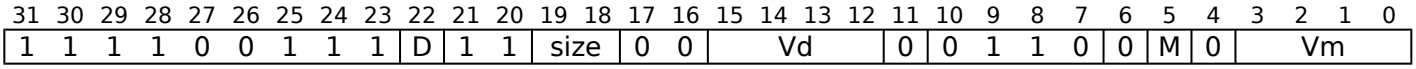
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

AESE

AES single round encryption.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

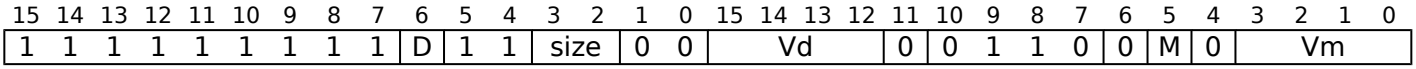


A1

```
AESE.<dt> <Qd>, <Qm>
```

```
if !HaveAESExt() then UNDEFINED;
if size != '00' then UNDEFINED;
if Vd<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); m = UInt(M:Vm);
```

T1



T1

```
AESE.<dt> <Qd>, <Qm>
```

```
if InITBlock() then UNPREDICTABLE;
if !HaveAESExt() then UNDEFINED;
if size != '00' then UNDEFINED;
if Vd<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); m = UInt(M:Vm);
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<dt> Is the data type, encoded in “size”:

size	<dt>
00	8
01	RESERVED
1x	RESERVED

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckCryptoEnabled32();
    op1 = Q[d>>1]; op2 = Q[m>>1];
    Q[d>>1] = AESSubBytes(AESShiftRows(op1 EOR op2));
```

Operational information

If CPSR.DIT is 1:

- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

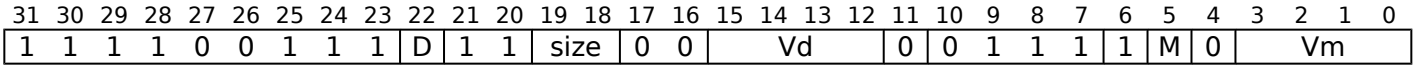
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

AESIMC

AES inverse mix columns.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

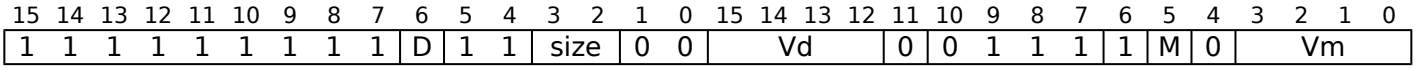


A1

```
AESIMC.<dt> <Qd>, <Qm>
```

```
if !HaveAESExt() then UNDEFINED;
if size != '00' then UNDEFINED;
if Vd<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); m = UInt(M:Vm);
```

T1



T1

```
AESIMC.<dt> <Qd>, <Qm>
```

```
if InITBlock() then UNPREDICTABLE;
if !HaveAESExt() then UNDEFINED;
if size != '00' then UNDEFINED;
if Vd<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); m = UInt(M:Vm);
```

Assembler Symbols

<dt> Is the data type, encoded in “size”:

size	<dt>
00	8
01	RESERVED
1x	RESERVED

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckCryptoEnabled32();
    Q[d>>1] = AESInvMixColumns(Q[m>>1]);
```

Operational information

If CPSR.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:

- The values of the data supplied in any of its registers.
- The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

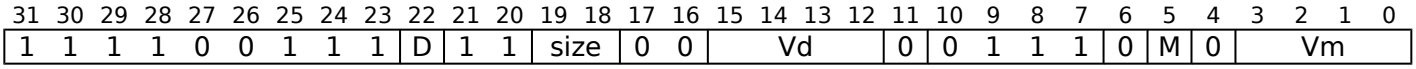
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

AESMC

AES mix columns.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

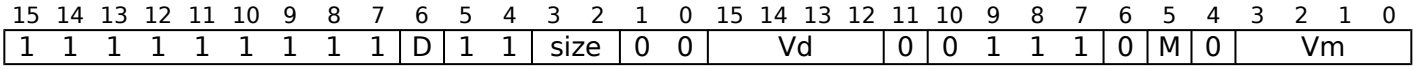


A1

```
AESMC.<dt> <Qd>, <Qm>
```

```
if !HaveAESExt() then UNDEFINED;
if size != '00' then UNDEFINED;
if Vd<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); m = UInt(M:Vm);
```

T1



T1

```
AESMC.<dt> <Qd>, <Qm>
```

```
if InITBlock() then UNPREDICTABLE;
if !HaveAESExt() then UNDEFINED;
if size != '00' then UNDEFINED;
if Vd<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); m = UInt(M:Vm);
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<dt> Is the data type, encoded in “size”:

size	<dt>
00	8
01	RESERVED
1x	RESERVED

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckCryptoEnabled32();
    Q[d>>1] = AESMixColumns(Q[m>>1]);
```

Operational information

If CPSR.DIT is 1:

- The execution time of this instruction is independent of:

    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FLDM*X (FLDMDBX, FLDMIAX)

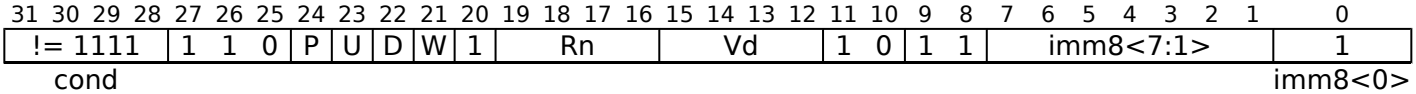
FLDMDBX is the Decrement Before variant of this instruction, and FLDMIAX is the Increment After variant. FLDM*X loads multiple SIMD&FP registers from consecutive locations in the Advanced SIMD and floating-point register file using an address from a general-purpose register.

Arm deprecates use of FLDMDBX and FLDMIAX, except for disassembly purposes, and reassembly of disassembled code.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



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

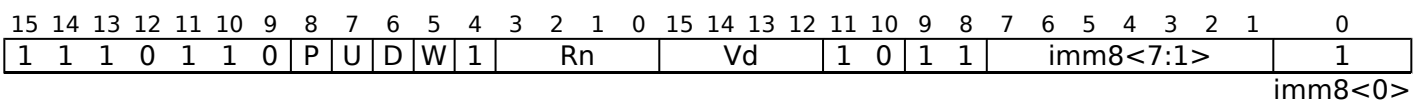
If `regs == 0`, then one of the following behaviors must occur:

- 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



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();  CheckVFPEnabled(TRUE);
    address = if add then R[n] else R[n]-imm32;
    for r = 0 to regs-1
        if single_regs then
            S[d+r] = MemA[address,4];  address = address+4;
        else
            word1 = MemA[address,4];  word2 = MemA[address+4,4];  address = address+8;
            // Combine the word-aligned words in the correct order for current endianness.
            D[d+r] = if BigEndian(AccType_ATOMIC) then word1:word2 else word2:word1;
    if wback then R[n] = if add then R[n]+imm32 else R[n]-imm32;
```

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

FSTMDBX, FSTMIAX

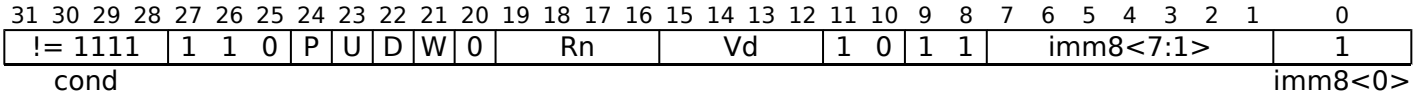
FSTMX stores multiple SIMD&FP registers from the Advanced SIMD and floating-point register file to consecutive locations in using an address from a general-purpose register.

Arm deprecates use of FSTMDBX and FSTMIAX, except for disassembly purposes, and reassembly of disassembled code.

Depending on settings in the *CPACR*, *NSACR*, *HCPTR*, and *FPEXC* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



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

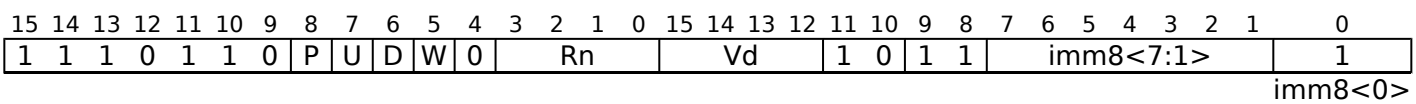
If `regs == 0`, then one of the following behaviors must occur:

- 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



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();  CheckVFPEnabled(TRUE);
    address = if add then R[n] else R[n]-imm32;
    for r = 0 to regs-1
        if single_regs then
            MemA[address,4] = S[d+r];  address = address+4;
        else
            // Store as two word-aligned words in the correct order for current endianness.
            MemA[address,4] = if BigEndian(AccType_ATOMIC) then D[d+r]<63:32> else D[d+r]<31:0>;
            MemA[address+4,4] = if BigEndian(AccType_ATOMIC) then D[d+r]<31:0> else D[d+r]<63:32>;
            address = address+8;
    if wback then R[n] = if add then R[n]+imm32 else R[n]-imm32;
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

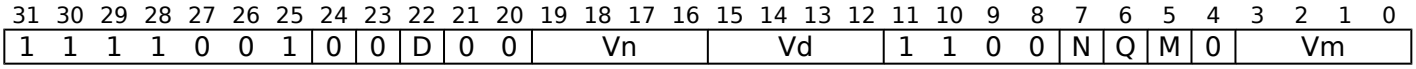
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SHA1C

SHA1 hash update (choose).

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

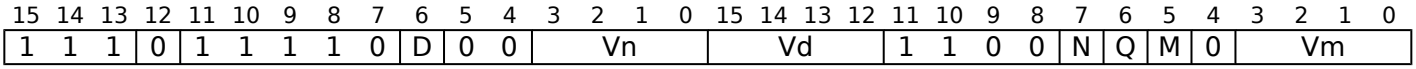


A1

```
SHA1C.32 <Qd>, <Qn>, <Qm>
```

```
if !HaveSHA1Ext() then UNDEFINED;
if Q != '1' then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

T1



T1

```
SHA1C.32 <Qd>, <Qn>, <Qm>
```

```
if InITBlock() then UNPREDICTABLE;
if !HaveSHA1Ext() then UNDEFINED;
if Q != '1' then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qn>	Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
<Qm>	Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckCryptoEnabled32();
    X = Q[d>>1];
    Y = Q[n>>1]<31:0>; // Note: 32 bits wide
    W = Q[m>>1];
    for e = 0 to 3
        t = SHAchoose(X<63:32>, X<95:64>, X<127:96>);
        Y = Y + ROL(X<31:0>, 5) + t + Elem[W, e, 32];
        X<63:32> = ROL(X<63:32>, 30);
        <Y, X> = ROL(Y:X, 32);
    Q[d>>1] = X;
```

Operational information

If CPSR.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SHA1H

SHA1 fixed rotate.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size		0	1	Vd				0	0	1	0	1	1	M	0	Vm			

A1

```
SHA1H.32 <Qd>, <Qm>
```

```
if !HaveSHA1Ext() then UNDEFINED;
if size != '10' then UNDEFINED;
if Vd<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); m = UInt(M:Vm);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size		0	1	Vd				0	0	1	0	1	1	M	0	Vm			

T1

```
SHA1H.32 <Qd>, <Qm>
```

```
if InITBlock() then UNPREDICTABLE;
if !HaveSHA1Ext() then UNDEFINED;
if size != '10' then UNDEFINED;
if Vd<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); m = UInt(M:Vm);
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qm>	Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckCryptoEnabled32();
    Q[d>>1] = ZeroExtend(ROL(Q[m>>1]<31:0>, 30), 128);
```

Operational information

If CPSR.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

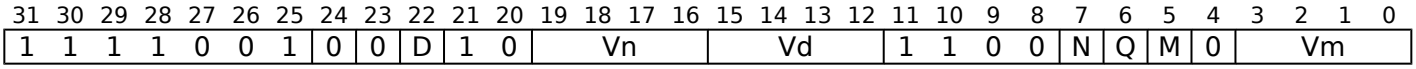
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SHA1M

SHA1 hash update (majority).

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

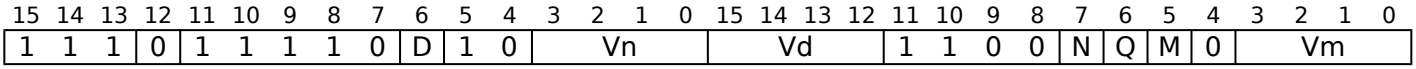


A1

SHA1M.32 <Qd>, <Qn>, <Qm>

```
if !HaveSHA1Ext() then UNDEFINED;
if Q != '1' then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

T1



T1

SHA1M.32 <Qd>, <Qn>, <Qm>

```
if InITBlock() then UNPREDICTABLE;
if !HaveSHA1Ext() then UNDEFINED;
if Q != '1' then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qn>	Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
<Qm>	Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckCryptoEnabled32();
    X = Q[d>>1];
    Y = Q[n>>1]<31:0>; // Note: 32 bits wide
    W = Q[m>>1];
    for e = 0 to 3
        t = SHAmajority(X<63:32>, X<95:64>, X<127:96>);
        Y = Y + ROL(X<31:0>, 5) + t + Elem[W, e, 32];
        X<63:32> = ROL(X<63:32>, 30);
        <Y, X> = ROL(Y:X, 32);
    Q[d>>1] = X;
```

Operational information

If CPSR.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

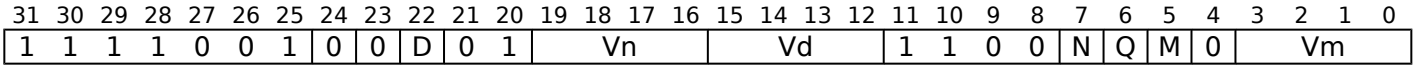
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SHA1P

SHA1 hash update (parity).

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

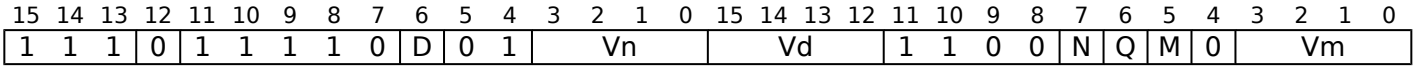


A1

```
SHA1P.32 <Qd>, <Qn>, <Qm>
```

```
if !HaveSHA1Ext() then UNDEFINED;
if Q != '1' then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

T1



T1

```
SHA1P.32 <Qd>, <Qn>, <Qm>
```

```
if InITBlock() then UNPREDICTABLE;
if !HaveSHA1Ext() then UNDEFINED;
if Q != '1' then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qn>	Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
<Qm>	Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckCryptoEnabled32();
    X = Q[d>>1];
    Y = Q[n>>1]<31:0>; // Note: 32 bits wide
    W = Q[m>>1];
    for e = 0 to 3
        t = SHAparity(X<63:32>, X<95:64>, X<127:96>);
        Y = Y + ROL(X<31:0>, 5) + t + Elem[W, e, 32];
        X<63:32> = ROL(X<63:32>, 30);
        <Y, X> = ROL(Y:X, 32);
    Q[d>>1] = X;
```

Operational information

If CPSR.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SHA1SU0

SHA1 schedule update 0.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	D	1	1	Vn				Vd				1	1	0	0	N	Q	M	0	Vm			

A1

```
SHA1SU0.32 <Qd>, <Qn>, <Qm>
```

```
if !HaveSHA1Ext() then UNDEFINED;
if Q != '1' then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	1	0	D	1	1	Vn				Vd				1	1	0	0	N	Q	M	0	Vm			

T1

```
SHA1SU0.32 <Qd>, <Qn>, <Qm>
```

```
if InITBlock() then UNPREDICTABLE;
if !HaveSHA1Ext() then UNDEFINED;
if Q != '1' then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qn>	Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
<Qm>	Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckCryptoEnabled32();
    op1 = Q[d>>1]; op2 = Q[n>>1]; op3 = Q[m>>1];
    op2 = op2<63:0> : op1<127:64>;
    Q[d>>1] = op1 EOR op2 EOR op3;
```

Operational information

If CPSR.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.

- The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

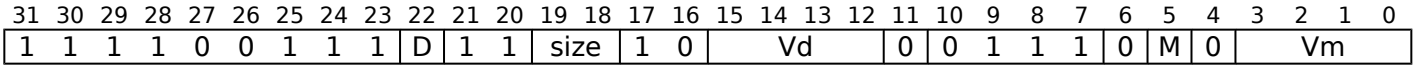
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SHA1SU1

SHA1 schedule update 1.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

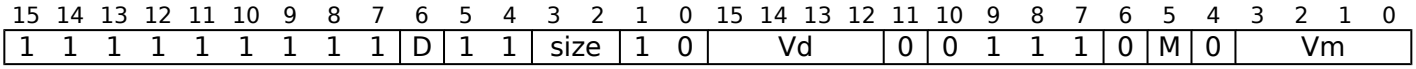


A1

```
SHA1SU1.32 <Qd>, <Qm>
```

```
if !HaveSHA1Ext() then UNDEFINED;
if size != '10' then UNDEFINED;
if Vd<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); m = UInt(M:Vm);
```

T1



T1

```
SHA1SU1.32 <Qd>, <Qm>
```

```
if InITBlock() then UNPREDICTABLE;
if !HaveSHA1Ext() then UNDEFINED;
if size != '10' then UNDEFINED;
if Vd<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); m = UInt(M:Vm);
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qm>	Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckCryptoEnabled32();
    X = Q[d>>1]; Y = Q[m>>1];
    T = X EOR LSR(Y, 32);
    W0 = ROL(T<31:0>, 1);
    W1 = ROL(T<63:32>, 1);
    W2 = ROL(T<95:64>, 1);
    W3 = ROL(T<127:96>, 1) EOR ROL(T<31:0>, 2);
    Q[d>>1] = W3:W2:W1:W0;
```

Operational information

If CPSR.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

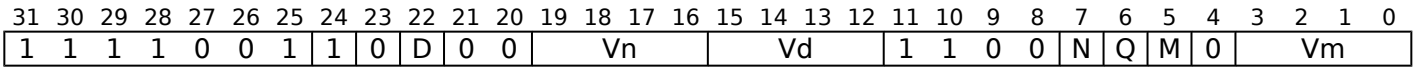
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SHA256H

SHA256 hash update part 1.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

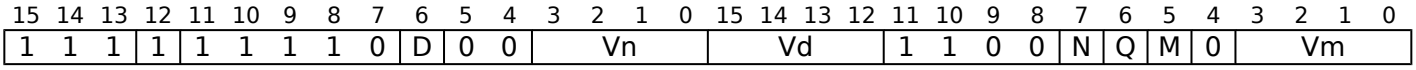


A1

```
SHA256H.32 <Qd>, <Qn>, <Qm>
```

```
if !HaveSHA256Ext() then UNDEFINED;
if Q != '1' then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

T1



T1

```
SHA256H.32 <Qd>, <Qn>, <Qm>
```

```
if InITBlock() then UNPREDICTABLE;
if !HaveSHA256Ext() then UNDEFINED;
if Q != '1' then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qn>	Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
<Qm>	Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckCryptoEnabled32();
    X = Q[d>>1]; Y = Q[n>>1]; W = Q[m>>1]; part1 = TRUE;
    Q[d>>1] = SHA256hash(X, Y, W, part1);
```

Operational information

If CPSR.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

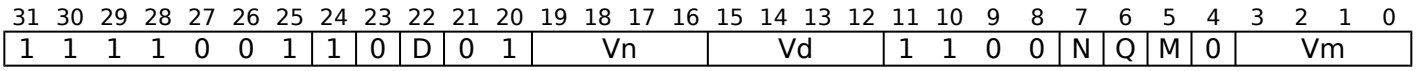
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SHA256H2

SHA256 hash update part 2.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

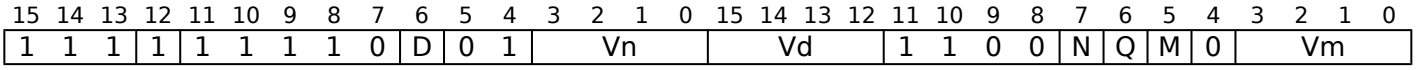


A1

```
SHA256H2.32 <Qd>, <Qn>, <Qm>
```

```
if !HaveSHA256Ext() then UNDEFINED;
if Q != '1' then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

T1



T1

```
SHA256H2.32 <Qd>, <Qn>, <Qm>
```

```
if InITBlock() then UNPREDICTABLE;
if !HaveSHA256Ext() then UNDEFINED;
if Q != '1' then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qn>	Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
<Qm>	Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckCryptoEnabled32();
    X = Q[n>>1]; Y = Q[d>>1]; W = Q[m>>1]; part1 = FALSE;
    Q[d>>1] = SHA256hash(X, Y, W, part1);
```

Operational information

If CPSR.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

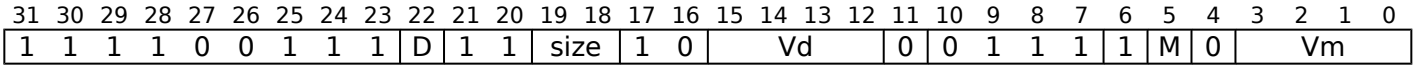
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SHA256SU0

SHA256 schedule update 0.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

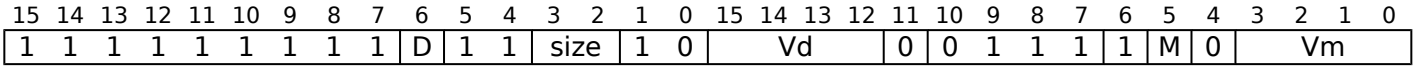


A1

```
SHA256SU0.32 <Qd>, <Qm>
```

```
if !HaveSHA256Ext() then UNDEFINED;
if size != '10' then UNDEFINED;
if Vd<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); m = UInt(M:Vm);
```

T1



T1

```
SHA256SU0.32 <Qd>, <Qm>
```

```
if InITBlock() then UNPREDICTABLE;
if !HaveSHA256Ext() then UNDEFINED;
if size != '10' then UNDEFINED;
if Vd<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); m = UInt(M:Vm);
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qm>	Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
if ConditionPassed() then
    bits(128) result;
    EncodingSpecificOperations(); CheckCryptoEnabled32();
    X = Q[d>>1]; Y = Q[m>>1];
    T = Y<31:0> : X<127:32>;
    for e = 0 to 3
        elt = Elem[T, e, 32];
        elt = ROR(elt, 7) EOR ROR(elt, 18) EOR LSR(elt, 3);
        Elem[result, e, 32] = elt + Elem[X, e, 32];
    Q[d>>1] = result;
```

Operational information

If CPSR.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.

    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

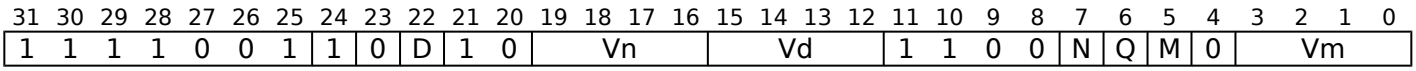
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SHA256SU1

SHA256 schedule update 1.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

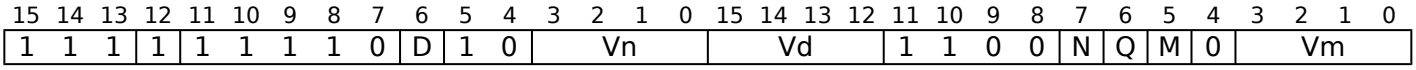


A1

```
SHA256SU1.32 <Qd>, <Qn>, <Qm>
```

```
if !HaveSHA256Ext() then UNDEFINED;
if Q != '1' then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

T1



T1

```
SHA256SU1.32 <Qd>, <Qn>, <Qm>
```

```
if InITBlock() then UNPREDICTABLE;
if !HaveSHA256Ext() then UNDEFINED;
if Q != '1' then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qn>	Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
<Qm>	Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
if ConditionPassed() then
    bits(128) result;
    EncodingSpecificOperations(); CheckCryptoEnabled32();
    X = Q[d>>1]; Y = Q[n>>1]; Z = Q[m>>1];
    T0 = Z<31:0> : Y<127:32>;

    T1 = Z<127:64>;
    for e = 0 to 1
        elt = Elem[T1, e, 32];
        elt = ROR(elt, 17) EOR ROR(elt, 19) EOR LSR(elt, 10);
        elt = elt + Elem[X, e, 32] + Elem[T0, e, 32];
        Elem[result, e, 32] = elt;

    T1 = result<63:0>;
    for e = 2 to 3
        elt = Elem[T1, e - 2, 32];
        elt = ROR(elt, 17) EOR ROR(elt, 19) EOR LSR(elt, 10);
        elt = elt + Elem[X, e, 32] + Elem[T0, e, 32];
        Elem[result, e, 32] = elt;

    Q[d>>1] = result;
```

Operational information

If CPSR.DIT is 1:

- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VABA

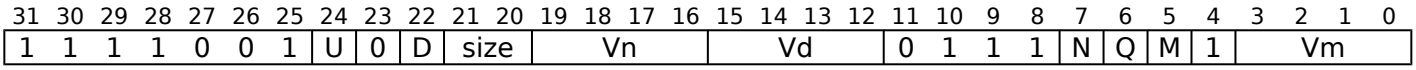
Vector Absolute Difference and Accumulate subtracts the elements of one vector from the corresponding elements of another vector, and accumulates the absolute values of the results into the elements of the destination vector.

Operand and result elements are all integers of the same length.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



64-bit SIMD vector (Q == 0)

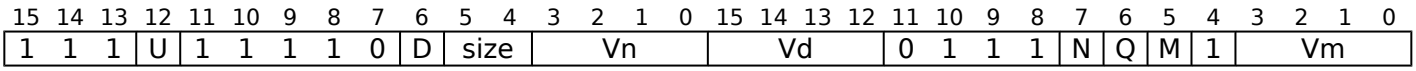
```
VABA{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VABA{<c>}{<q>}.<dt> <Qd>, <Qn>, <Qm>
```

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = (U == '1');  long_destination = FALSE;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1



64-bit SIMD vector (Q == 0)

```
VABA{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VABA{<c>}{<q>}.<dt> <Qd>, <Qn>, <Qm>
```

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = (U == '1');  long_destination = FALSE;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>	For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional. For encoding T1: see *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<dt>	Is the data type for the elements of the operands, encoded in “U:size”:

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	00	U8
1	01	U16
1	10	U32

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[Din[n+r],e,esize];
            op2 = Elem[Din[m+r],e,esize];
            absdiff = Abs(Int(op1,unsigned) - Int(op2,unsigned));
            if long_destination then
                Elem[Q[d>>1],e,2*esize] = Elem[Qin[d>>1],e,2*esize] + absdiff;
            else
                Elem[D[d+r],e,esize] = Elem[Din[d+r],e,esize] + absdiff;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VABAL

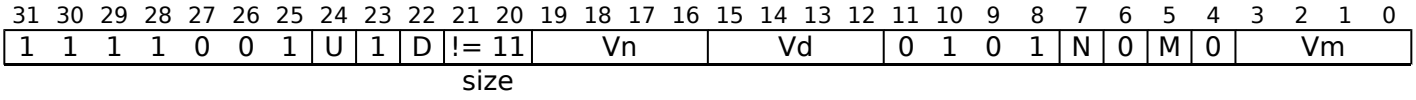
Vector Absolute Difference and Accumulate Long subtracts the elements of one vector from the corresponding elements of another vector, and accumulates the absolute values of the results into the elements of the destination vector.

Operand elements are all integers of the same length, and the result elements are double the length of the operands.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

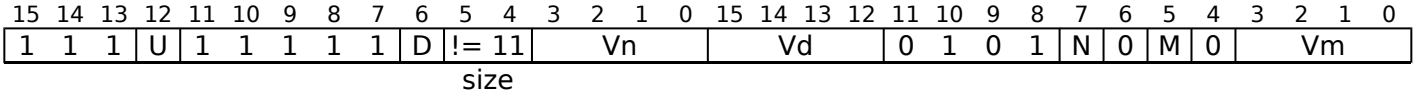


A1

```
VABAL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>
```

```
if size == '11' then SEE "Related encodings";
if Vd<0> == '1' then UNDEFINED;
unsigned = (U == '1');  long_destination = TRUE;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = 1;
```

T1



T1

```
VABAL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>
```

```
if size == '11' then SEE "Related encodings";
if Vd<0> == '1' then UNDEFINED;
unsigned = (U == '1');  long_destination = TRUE;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = 1;
```

Related encodings: See *Advanced SIMD data-processing* for the T32 instruction set, or *Advanced SIMD data-processing* for the A32 instruction set.

Assembler Symbols

<c>	For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional. For encoding T1: see *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<dt>	Is the data type for the elements of the operands, encoded in “U:size”:

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	00	U8
1	01	U16
1	10	U32

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[Din[n+r],e,esize];
            op2 = Elem[Din[m+r],e,esize];
            absdiff = Abs(Int(op1,unsigned) - Int(op2,unsigned));
            if long_destination then
                Elem[Q[d>>1],e,2*esize] = Elem[Qin[d>>1],e,2*esize] + absdiff;
            else
                Elem[D[d+r],e,esize] = Elem[Din[d+r],e,esize] + absdiff;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VABD (floating-point)

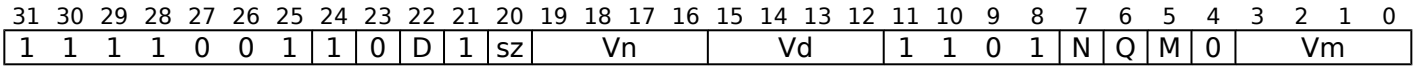
Vector Absolute Difference (floating-point) subtracts the elements of one vector from the corresponding elements of another vector, and places the absolute values of the results in the elements of the destination vector.

Operand and result elements are floating-point numbers of the same size.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



64-bit SIMD vector (Q == 0)

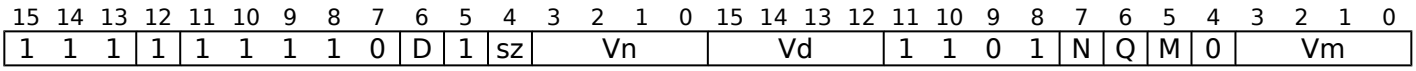
```
VABD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VABD{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1



64-bit SIMD vector (Q == 0)

```
VABD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VABD{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `sz == '1' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.
For encoding T1: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the vectors, encoded in “sz”:

sz	<dt>
0	F32
1	F16

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[D[n+r],e,esize];  op2 = Elem[D[m+r],e,esize];
            Elem[D[d+r],e,esize] = FPAbs(FPSub(op1,op2,StandardFPSCRValue()));
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VABD (integer)

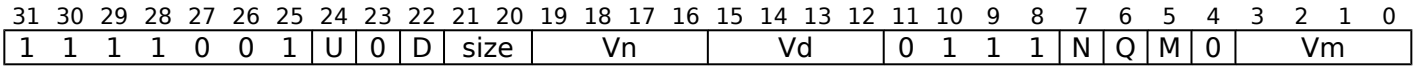
Vector Absolute Difference (integer) subtracts the elements of one vector from the corresponding elements of another vector, and places the absolute values of the results in the elements of the destination vector.

Operand and result elements are all integers of the same length.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



64-bit SIMD vector (Q == 0)

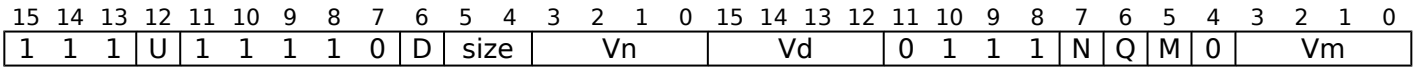
```
VABD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VABD{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = (U == '1');  long_destination = FALSE;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1



64-bit SIMD vector (Q == 0)

```
VABD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VABD{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = (U == '1');  long_destination = FALSE;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>	For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional. For encoding T1: see *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<dt>	Is the data type for the elements of the operands, encoded in “U:size”:

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	00	U8
1	01	U16
1	10	U32

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[Din[n+r],e,esize];
            op2 = Elem[Din[m+r],e,esize];
            absdiff = Abs(Int(op1,unsigned) - Int(op2,unsigned));
            if long_destination then
                Elem[Q[d>>1],e,2*esize] = absdiff<2*esize-1:0>;
            else
                Elem[D[d+r],e,esize] = absdiff<esize-1:0>;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VABDL (integer)

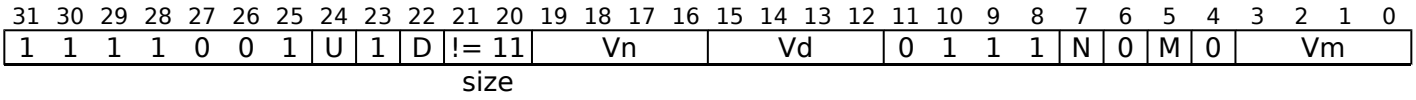
Vector Absolute Difference Long (integer) subtracts the elements of one vector from the corresponding elements of another vector, and places the absolute values of the results in the elements of the destination vector.

Operand elements are all integers of the same length, and the result elements are double the length of the operands.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

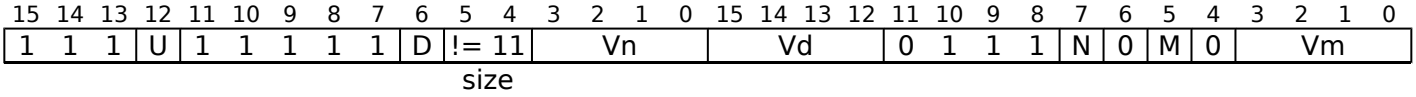


A1

```
VABDL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>
```

```
if size == '11' then SEE "Related encodings";
if Vd<0> == '1' then UNDEFINED;
unsigned = (U == '1');  long_destination = TRUE;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = 1;
```

T1



T1

```
VABDL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>
```

```
if size == '11' then SEE "Related encodings";
if Vd<0> == '1' then UNDEFINED;
unsigned = (U == '1');  long_destination = TRUE;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = 1;
```

Related encodings: See *Advanced SIMD data-processing* for the T32 instruction set, or *Advanced SIMD data-processing* for the A32 instruction set.

Assembler Symbols

<c>	For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional. For encoding T1: see *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<dt>	Is the data type for the elements of the operands, encoded in “U:size”:

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	00	U8
1	01	U16
1	10	U32

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[Din[n+r],e,esize];
            op2 = Elem[Din[m+r],e,esize];
            absdiff = Abs(Int(op1,unsigned) - Int(op2,unsigned));
            if long_destination then
                Elem[Q[d>>1],e,2*esize] = absdiff<2*esize-1:0>;
            else
                Elem[D[d+r],e,esize] = absdiff<esize-1:0>;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

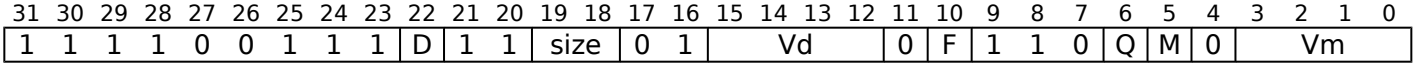
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VABS

Vector Absolute takes the absolute value of each element in a vector, and places the results in a second vector. The floating-point version only clears the sign bit.
Depending on settings in the *CPACR*, *NSACR*, *HCPTR*, and *FPEXC* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2) .

A1



64-bit SIMD vector (Q == 0)

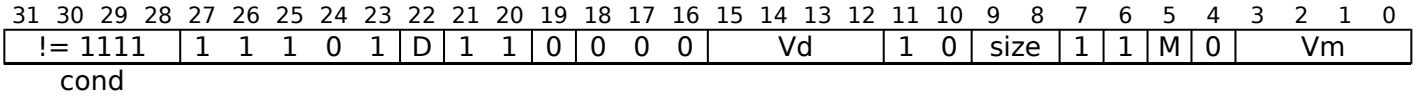
```
VABS{<c>}{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VABS{<c>}{<q>}.<dt> <Qd>, <Qm>
```

```
if size == '11' then UNDEFINED;
if F == '1' && ((size == '01' && !HaveFP16Ext()) || size == '00') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
advsimd = TRUE;  floating_point = (F == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

A2



Half-precision scalar (size == 01)
(FEAT_FP16)

```
VABS{<c>}{<q>}.F16 <Sd>, <Sm>
```

Single-precision scalar (size == 10)

```
VABS{<c>}{<q>}.F32 <Sd>, <Sm>
```

Double-precision scalar (size == 11)

```
VABS{<c>}{<q>}.F64 <Dd>, <Dm>
```

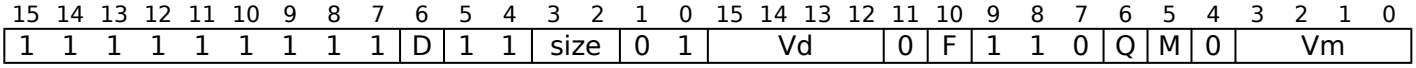
```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
advsimd = FALSE;
case size of
    when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
    when '10' esize = 32; 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



64-bit SIMD vector (Q == 0)

```
VABS{<c>}{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VABS{<c>}{<q>}.<dt> <Qd>, <Qm>
```

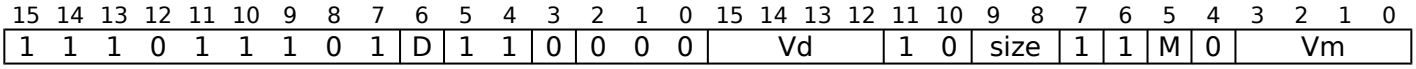
```
if size == '11' then UNDEFINED;
if F == '1' && ((size == '01' && !HaveFP16Ext()) || size == '00') then UNDEFINED;
if F == '1' && size == '01' && InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
advsimd = TRUE;  floating_point = (F == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `F == '1' && size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T2



Half-precision scalar (size == 01)
(FEAT_FP16)

```
VABS{<c>}{<q>}.F16 <Sd>, <Sm>
```

Single-precision scalar (size == 10)

```
VABS{<c>}{<q>}.F32 <Sd>, <Sm>
```

Double-precision scalar (size == 11)

```
VABS{<c>}{<q>}.F64 <Dd>, <Dm>
```

```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock()  then UNPREDICTABLE;
advsimd = FALSE;
case size of
    when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.

For encoding A2, T1 and T2: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the vectors, encoded in “F:size”:

F	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	01	F16
1	10	F32

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

<Sm> Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
    if advsimd then  // Advanced SIMD instruction
        for r = 0 to regs-1
            for e = 0 to elements-1
                if floating_point then
                    Elem[D[d+r],e,esize] = FPAbs(Elem[D[m+r],e,esize]);
                else
                    result = Abs(SInt(Elem[D[m+r],e,esize]));
                    Elem[D[d+r],e,esize] = result<esize-1:0>;
    else             // VFP instruction
        case esize of
            when 16 S[d] = Zeros(16) : FPAbs(S[m]<15:0>);
            when 32 S[d] = FPAbs(S[m]);
            when 64 D[d] = FPAbs(D[m]);
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check and is operating only on integer vector elements, then the following apply:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VACGE

Vector Absolute Compare Greater Than or Equal takes the absolute value of each element in a vector, and compares it with the absolute value of the corresponding element of a second vector. If the first is greater than or equal to the second, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

The operands and result can be quadword or doubleword vectors. They must all be the same size.

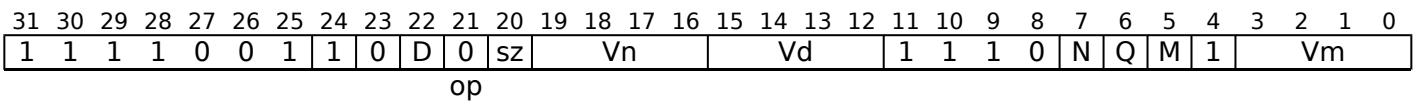
The operand vector elements are floating-point numbers. The result vector elements are the same size as the operand vector elements.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

This instruction is used by the pseudo-instruction VACLE.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



64-bit SIMD vector (Q == 0)

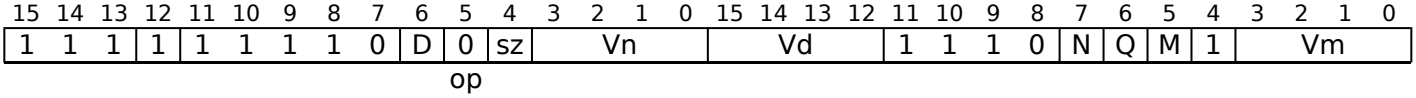
```
VACGE{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VACGE{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
or_equal = (op == '0');
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1



64-bit SIMD vector (Q == 0)

```
VACGE{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VACGE{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
or_equal = (op == '0');
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `sz == '1' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.
For encoding T1: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the vectors, encoded in “sz”:

sz	<dt>
0	F32
1	F16

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = FPAbs(Elem[D[n+r],e,esize]);  op2 = FPAbs(Elem[D[m+r],e,esize]);
            if or_equal then
                test_passed = FPCompareGE(op1, op2, StandardFPSCRValue());
            else
                test_passed = FPCompareGT(op1, op2, StandardFPSCRValue());
            Elem[D[d+r],e,esize] = if test_passed then Ones(esize) else Zeros(esize);
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VACGT

Vector Absolute Compare Greater Than takes the absolute value of each element in a vector, and compares it with the absolute value of the corresponding element of a second vector. If the first is greater than the second, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

The operands and result can be quadword or doubleword vectors. They must all be the same size.

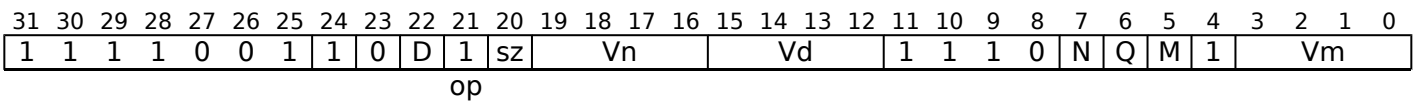
The operand vector elements are floating-point numbers. The result vector elements are the same size as the operand vector elements.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

This instruction is used by the pseudo-instruction VACLT.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



64-bit SIMD vector (Q == 0)

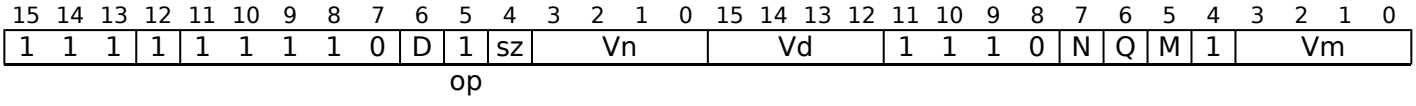
```
VACGT{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VACGT{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
or_equal = (op == '0');
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1



64-bit SIMD vector (Q == 0)

```
VACGT{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VACGT{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
or_equal = (op == '0');
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `sz == '1' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.
For encoding T1: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the vectors, encoded in “sz”:

sz	<dt>
0	F32
1	F16

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = FPAbs(Elem[D[n+r],e,esize]);  op2 = FPAbs(Elem[D[m+r],e,esize]);
            if or_equal then
                test_passed = FPCompareGE(op1, op2, StandardFPSCRValue());
            else
                test_passed = FPCompareGT(op1, op2, StandardFPSCRValue());
            Elem[D[d+r],e,esize] = if test_passed then Ones(esize) else Zeros(esize);
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VACLE

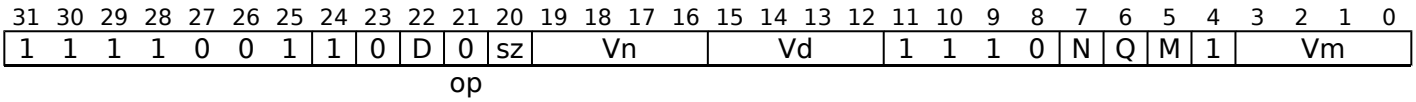
Vector Absolute Compare Less Than or Equal takes the absolute value of each element in a vector, and compares it with the absolute value of the corresponding element of a second vector. If the first is less than or equal to the second, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

This is a pseudo-instruction of VACGE. This means:

- The encodings in this description are named to match the encodings of VACGE.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of VACGE gives the operational pseudocode for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



64-bit SIMD vector (Q == 0)

```
VACLE{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

is equivalent to

```
VACGE{<c>}{<q>}.<dt> <Dd>, <Dm>, <Dn>
```

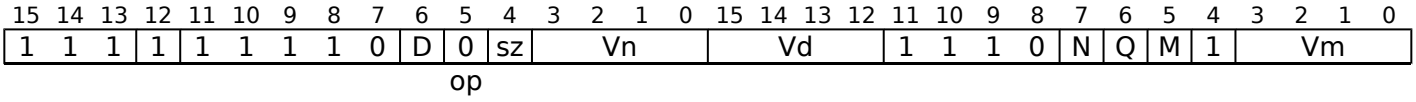
128-bit SIMD vector (Q == 1)

```
VACLE{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

is equivalent to

```
VACGE{<c>}{<q>}.<dt> <Qd>, <Qm>, <Qn>
```

T1



64-bit SIMD vector (Q == 0)

```
VACLE{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

is equivalent to

```
VACGE{<c>}{<q>}.<dt> <Dd>, <Dm>, <Dn>
```

128-bit SIMD vector (Q == 1)

```
VACLE{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

is equivalent to

```
VACGE{<c>}{<q>}.<dt> <Qd>, <Qm>, <Qn>
```

Assembler Symbols

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.
For encoding T1: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the vectors, encoded in “sz”:

sz	<dt>
0	F32
1	F16

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

Operation

The description of VACGE gives the operational pseudocode for this instruction.

Operational information

The description of *VACGE* gives the operational pseudocode for this instruction.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VACLT

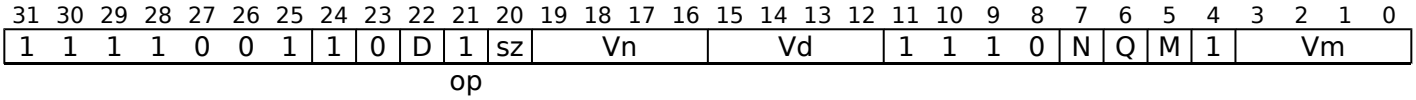
Vector Absolute Compare Less Than takes the absolute value of each element in a vector, and compares it with the absolute value of the corresponding element of a second vector. If the first is less than the second, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

This is a pseudo-instruction of VACGT. This means:

- The encodings in this description are named to match the encodings of VACGT.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of VACGT gives the operational pseudocode for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



64-bit SIMD vector (Q == 0)

```
VACLT{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

is equivalent to

```
VACGT{<c>}{<q>}.<dt> <Dd>, <Dm>, <Dn>
```

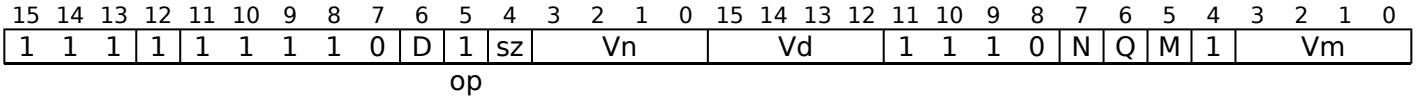
128-bit SIMD vector (Q == 1)

```
VACLT{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

is equivalent to

```
VACGT{<c>}{<q>}.<dt> <Qd>, <Qm>, <Qn>
```

T1



64-bit SIMD vector (Q == 0)

```
VACLT{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

is equivalent to

```
VACGT{<c>}{<q>}.<dt> <Dd>, <Dm>, <Dn>
```

128-bit SIMD vector (Q == 1)

```
VACLT{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

is equivalent to

```
VACGT{<c>}{<q>}.<dt> <Qd>, <Qm>, <Qn>
```

Assembler Symbols

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.
For encoding T1: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the vectors, encoded in “sz”:

sz	<dt>
0	F32
1	F16

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

Operation

The description of VACGT gives the operational pseudocode for this instruction.

Operational information

The description of *VACGT* gives the operational pseudocode for this instruction.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

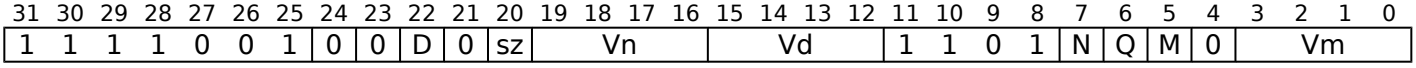
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VADD (floating-point)

Vector Add (floating-point) adds corresponding elements in two vectors, and places the results in the destination vector.
Depending on settings in the *CPACR*, *NSACR*, *HCPTR*, and *FPEXC* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2) .

A1



64-bit SIMD vector (Q == 0)

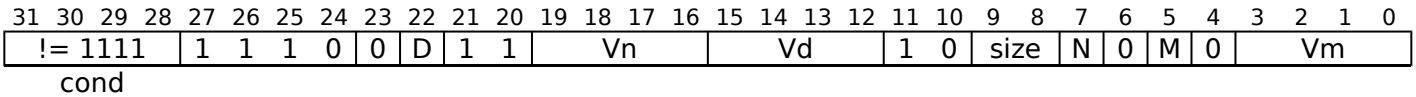
```
VADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

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



Half-precision scalar (size == 01)
(FEAT_FP16)

```
VADD{<c>}{<q>}.F16 {<Sd>,} <Sn>, <Sm>
```

Single-precision scalar (size == 10)

```
VADD{<c>}{<q>}.F32 {<Sd>,} <Sn>, <Sm>
```

Double-precision scalar (size == 11)

```
VADD{<c>}{<q>}.F64 {<Dd>,} <Dn>, <Dm>
```

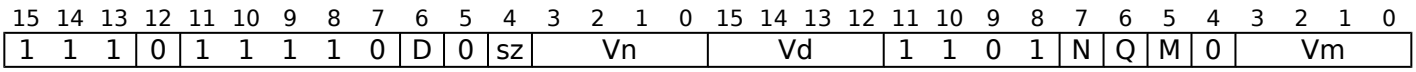
```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
advsimd = FALSE;
case size of
    when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '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)

```
VADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VADD{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

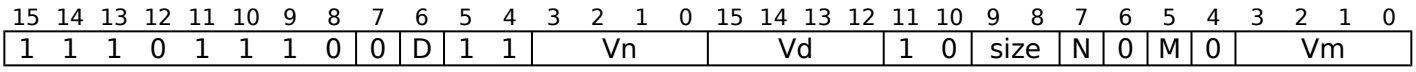
```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
advsimd = TRUE;
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd);  n = 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



Half-precision scalar (size == 01)
(FEAT_FP16)

```
VADD{<c>}{<q>}.F16 {<Sd>,} <Sn>, <Sm>
```

Single-precision scalar (size == 10)

```
VADD{<c>}{<q>}.F32 {<Sd>,} <Sn>, <Sm>
```

Double-precision scalar (size == 11)

```
VADD{<c>}{<q>}.F64 {<Dd>,} <Dn>, <Dm>
```

```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock()  then UNPREDICTABLE;
advsimd = FALSE;
case size of
    when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.
For encoding A2, T1 and T2: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the vectors, encoded in “sz”:

sz	<dt>
0	F32
1	F16

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

<Sn> Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.

<Sm> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
    if advsimd then  // Advanced SIMD instruction
        for r = 0 to regs-1
            for e = 0 to elements-1
                Elem[D[d+r],e,esize] = FPAdd(Elem[D[n+r],e,esize], Elem[D[m+r],e,esize],
                                             StandardFPSCRValue());
    else             // VFP instruction
        case esize of
            when 16
                S[d] = Zeros(16) : FPAdd(S[n]<15:0>, S[m]<15:0>, FPSCR[]);
            when 32
                S[d] = FPAdd(S[n], S[m], FPSCR[]);
            when 64
                D[d] = FPAdd(D[n], D[m], FPSCR[]);
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

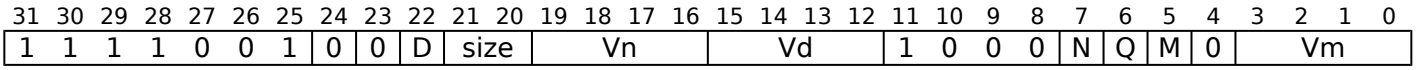
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VADD (integer)

Vector Add (integer) adds corresponding elements in two vectors, and places the results in the destination vector. Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



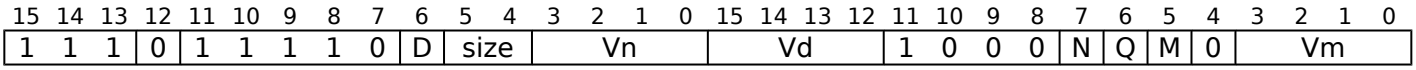
64-bit SIMD vector (Q == 0)

```
VADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VADD{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1



64-bit SIMD vector (Q == 0)

```
VADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VADD{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.
For encoding T1: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the vectors, encoded in “size”:

size	<dt>
00	I8
01	I16
10	I32
11	I64

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            Elem[D[d+r],e,esize] = Elem[D[n+r],e,esize] + Elem[D[m+r],e,esize];
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VADDHN

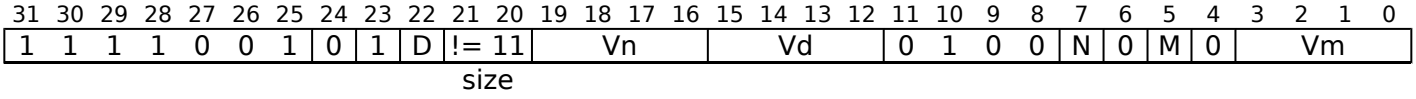
Vector Add and Narrow, returning High Half adds corresponding elements in two quadword vectors, and places the most significant half of each result in a doubleword vector. The results are truncated. For rounded results, see *VRADDHN*.

The operand elements can be 16-bit, 32-bit, or 64-bit integers. There is no distinction between signed and unsigned integers.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

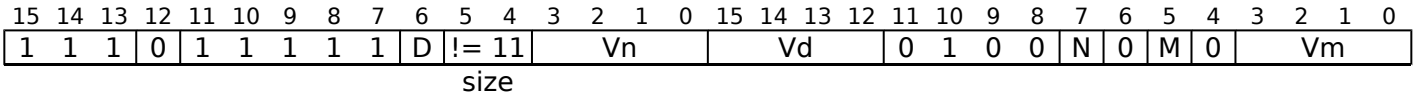


A1

```
VADDHN{<c>}{<q>}.<dt> <Dd>, <Qn>, <Qm>
```

```
if size == '11' then SEE "Related encodings";
if Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

T1



T1

```
VADDHN{<c>}{<q>}.<dt> <Dd>, <Qn>, <Qm>
```

```
if size == '11' then SEE "Related encodings";
if Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

Related encodings: See *Advanced SIMD data-processing* for the T32 instruction set, or *Advanced SIMD data-processing* for the A32 instruction set.

Assembler Symbols

<c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the operands, encoded in “size”:

size	<dt>
00	I16
01	I32
10	I64

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for e = 0 to elements-1
        result = Elem[Qin[n>>1],e,2*esize] + Elem[Qin[m>>1],e,2*esize];
        Elem[D[d],e,esize] = result<2*esize-1:esize>;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

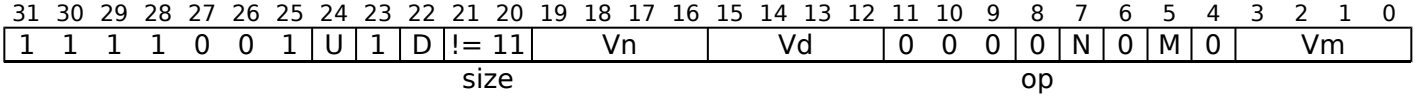
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VADDL

Vector Add Long adds corresponding elements in two doubleword vectors, and places the results in a quadword vector. Before adding, it sign-extends or zero-extends the elements of both operands.
Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

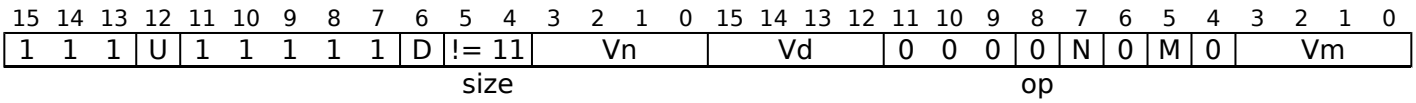


A1

```
VADDL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>
```

```
if size == '11' then SEE "Related encodings";
if Vd<0> == '1' || (op == '1' && Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;  is_vaddw = (op == '1');
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

T1



T1

```
VADDL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>
```

```
if size == '11' then SEE "Related encodings";
if Vd<0> == '1' || (op == '1' && Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;  is_vaddw = (op == '1');
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

Related encodings: See *Advanced SIMD data-processing* for the T32 instruction set, or *Advanced SIMD data-processing* for the A32 instruction set.

Assembler Symbols

<c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.
For encoding T1: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the second operand vector, encoded in “U:size”:

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	00	U8
1	01	U16
1	10	U32

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for e = 0 to elements-1
        if is_vaddw then
            op1 = Int(Elem[Qin[n>>1],e,2*esize], unsigned);
        else
            op1 = Int(Elem[Din[n],e,esize], unsigned);
        result = op1 + Int(Elem[Din[m],e,esize],unsigned);
        Elem[Q[d>>1],e,2*esize] = result<2*esize-1:0>;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

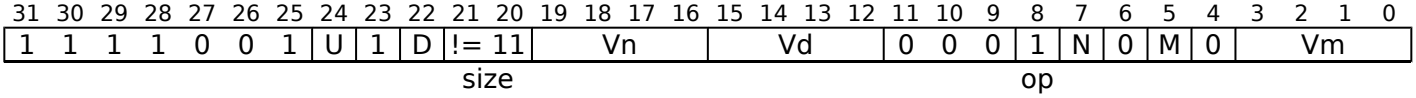
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VADDW

Vector Add Wide adds corresponding elements in one quadword and one doubleword vector, and places the results in a quadword vector. Before adding, it sign-extends or zero-extends the elements of the doubleword operand.
Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

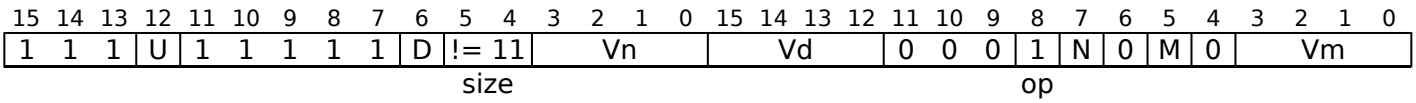
A1



A1

```
VADDW{<c>}{<q>}.<dt> {<Qd>,} <Qn>, <Dm>

if size == '11' then SEE "Related encodings";
if Vd<0> == '1' || (op == '1' && Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;  is_vaddw = (op == '1');
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

T1



T1

```
VADDW{<c>}{<q>}.<dt> {<Qd>,} <Qn>, <Dm>

if size == '11' then SEE "Related encodings";
if Vd<0> == '1' || (op == '1' && Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;  is_vaddw = (op == '1');
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

Related encodings: See *Advanced SIMD data-processing* for the T32 instruction set, or *Advanced SIMD data-processing* for the A32 instruction set.

Assembler Symbols

<c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.
For encoding T1: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the second operand vector, encoded in “U:size”:

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	00	U8
1	01	U16
1	10	U32

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for e = 0 to elements-1
        if is_vaddw then
            op1 = Int(Elem[Qin[n>>1],e,2*esize], unsigned);
        else
            op1 = Int(Elem[Din[n],e,esize], unsigned);
        result = op1 + Int(Elem[Din[m],e,esize],unsigned);
        Elem[Q[d>>1],e,2*esize] = result<2*esize-1:0>;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

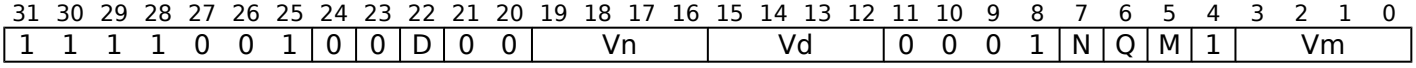
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VAND (register)

Vector Bitwise AND (register) performs a bitwise AND operation between two registers, and places the result in the destination register.
Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



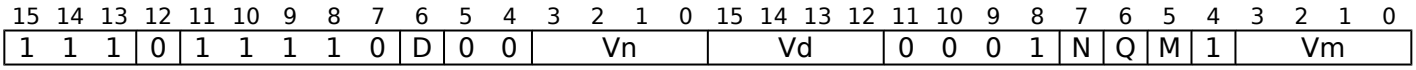
64-bit SIMD vector (Q == 0)

```
VAND{<c>}{<q>}{.<dt>} {<Dd>,} <Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VAND{<c>}{<q>}{.<dt>} {<Qd>,} <Qn>, <Qm>

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1



64-bit SIMD vector (Q == 0)

```
VAND{<c>}{<q>}{.<dt>} {<Dd>,} <Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VAND{<c>}{<q>}{.<dt>} {<Qd>,} <Qn>, <Qm>

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>	For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional. For encoding T1: see *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<dt>	An optional data type. It is ignored by assemblers, and does not affect the encoding.
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qn>	Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
<Qm>	Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        D[d+r] = D[n+r] AND D[m+r];
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VAND (immediate)

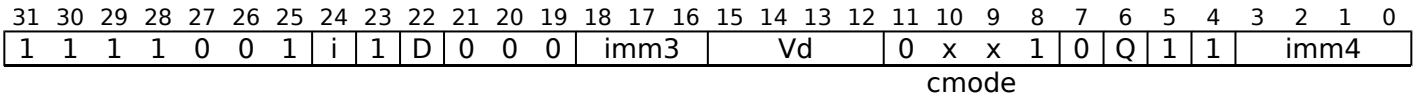
Vector Bitwise AND (immediate) performs a bitwise AND between a register value and an immediate value, and returns the result into the destination vector

This is a pseudo-instruction of VBIC (immediate). This means:

- The encodings in this description are named to match the encodings of VBIC (immediate).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of VBIC (immediate) gives the operational pseudocode for this instruction.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2) .

A1



64-bit SIMD vector (Q == 0)

```
VAND{<c>}{<q>}.I16 {<Dd>,} <Dd>, #<imm>
```

is equivalent to

```
VBIC{<c>}{<q>}.I16 <Dd>, #~<imm>
```

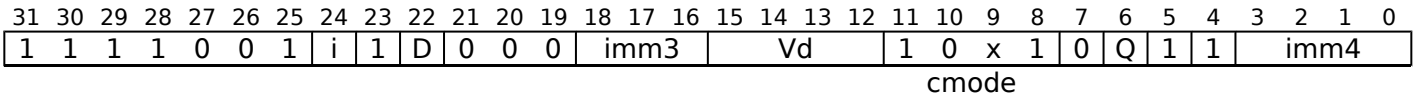
128-bit SIMD vector (Q == 1)

```
VAND{<c>}{<q>}.I16 {<Qd>,} <Qd>, #<imm>
```

is equivalent to

```
VBIC{<c>}{<q>}.I16 <Qd>, #~<imm>
```

A2



64-bit SIMD vector (Q == 0)

```
VAND{<c>}{<q>}.I32 {<Dd>,} <Dd>, #<imm>
```

is equivalent to

```
VBIC{<c>}{<q>}.I32 <Dd>, #~<imm>
```

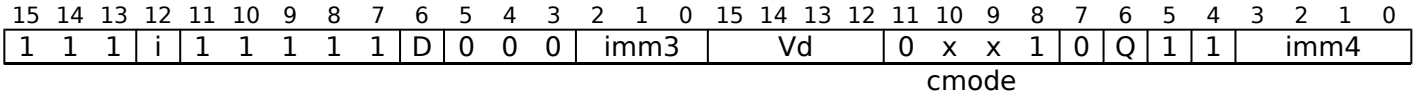
128-bit SIMD vector (Q == 1)

```
VAND{<c>}{<q>}.I32 {<Qd>,} <Qd>, #<imm>
```

is equivalent to

```
VBIC{<c>}{<q>}.I32 <Qd>, #~<imm>
```

T1



64-bit SIMD vector (Q == 0)

```
VAND{<c>}{<q>}.I16 {<Dd>,} <Dd>, #<imm>
```

is equivalent to

```
VBIC{<c>}{<q>}.I16 <Dd>, #~<imm>
```

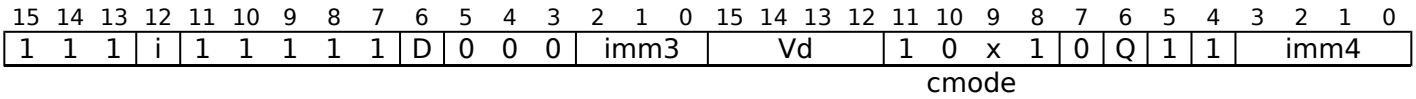
128-bit SIMD vector (Q == 1)

```
VAND{<c>}{<q>}.I16 {<Qd>,} <Qd>, #<imm>
```

is equivalent to

```
VBIC{<c>}{<q>}.I16 <Qd>, #~<imm>
```

T2



64-bit SIMD vector (Q == 0)

```
VAND{<c>}{<q>}.I32 {<Dd>,} <Dd>, #<imm>
```

is equivalent to

```
VBIC{<c>}{<q>}.I32 <Dd>, #~<imm>
```

128-bit SIMD vector (Q == 1)

```
VAND{<c>}{<q>}.I32 {<Qd>,} <Qd>, #<imm>
```

is equivalent to

```
VBIC{<c>}{<q>}.I32 <Qd>, #~<imm>
```

Assembler Symbols

<c>	For encoding A1 and A2: see *Standard assembler syntax fields*. This encoding must be unconditional. For encoding T1 and T2: see *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<imm>	Is a constant of the specified type that is replicated to fill the destination register. For details of the range of constants available and the encoding of <imm>, see *Modified immediate constants in T32 and A32 Advanced SIMD instructions*.

Operation

The description of VBIC (immediate) gives the operational pseudocode for this instruction.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VBIC (immediate)

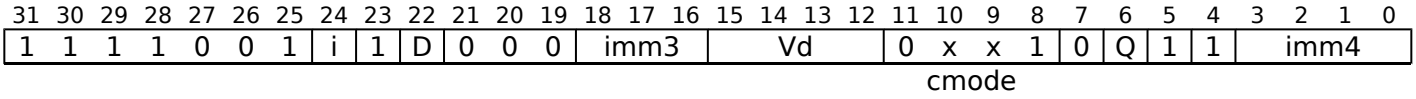
Vector Bitwise Bit Clear (immediate) performs a bitwise AND between a register value and the complement of an immediate value, and returns the result into the destination vector.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

This instruction is used by the pseudo-instruction VAND (immediate).

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2) .

A1



64-bit SIMD vector (Q == 0)

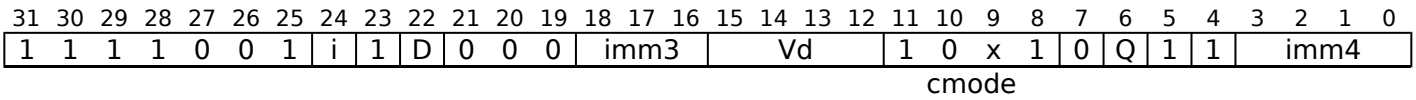
```
VBIC{<c>}{<q>}.I32 {<Dd>,} <Dd>, #<imm>
```

128-bit SIMD vector (Q == 1)

```
VBIC{<c>}{<q>}.I32 {<Qd>,} <Qd>, #<imm>
```

```
if cmode<0> == '0' || cmode<3:2> == '11' then SEE "Related encodings";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
imm64 = AdvSIMDExpandImm('1', cmode, i:imm3:imm4);
d = UInt(D:Vd);  regs = if Q == '0' then 1 else 2;
```

A2



64-bit SIMD vector (Q == 0)

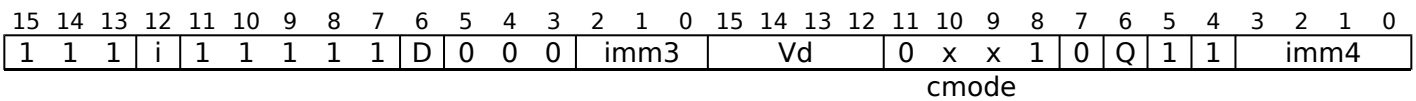
```
VBIC{<c>}{<q>}.I16 {<Dd>,} <Dd>, #<imm>
```

128-bit SIMD vector (Q == 1)

```
VBIC{<c>}{<q>}.I16 {<Qd>,} <Qd>, #<imm>
```

```
if cmode<0> == '0' || cmode<3:2> == '11' then SEE "Related encodings";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
imm64 = AdvSIMDExpandImm('1', cmode, i:imm3:imm4);
d = UInt(D:Vd);  regs = if Q == '0' then 1 else 2;
```

T1



64-bit SIMD vector (Q == 0)

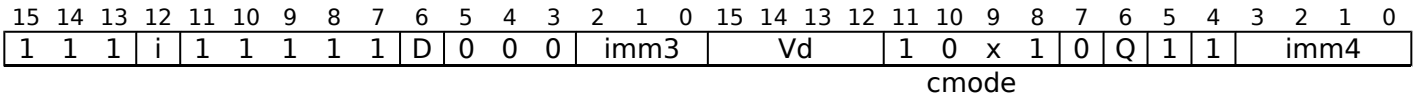
```
VBIC{<c>}{<q>}.I32 {<Dd>,} <Dd>, #<imm>
```

128-bit SIMD vector (Q == 1)

```
VBIC{<c>}{<q>}.I32 {<Qd>,} <Qd>, #<imm>
```

```
if cmode<0> == '0' || cmode<3:2> == '11' then SEE "Related encodings";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
imm64 = AdvSIMDExpandImm('1', cmode, i:imm3:imm4);
d = UInt(D:Vd);  regs = if Q == '0' then 1 else 2;
```

T2



64-bit SIMD vector (Q == 0)

```
VBIC{<c>}{<q>}.I16 {<Dd>,} <Dd>, #<imm>
```

128-bit SIMD vector (Q == 1)

```
VBIC{<c>}{<q>}.I16 {<Qd>,} <Qd>, #<imm>
```

```
if cmode<0> == '0' || cmode<3:2> == '11' then SEE "Related encodings";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
imm64 = AdvSIMDExpandImm('1', cmode, i:imm3:imm4);
d = UInt(D:Vd);  regs = if Q == '0' then 1 else 2;
```

Related encodings: See *Advanced SIMD one register and modified immediate* for the T32 instruction set, or *Advanced SIMD one register and modified immediate* for the A32 instruction set.

Assembler Symbols

<c>	For encoding A1 and A2: see *Standard assembler syntax fields*. This encoding must be unconditional. For encoding T1 and T2: see *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<imm>	Is a constant of the specified type that is replicated to fill the destination register. For details of the range of constants available and the encoding of <imm>, see *Modified immediate constants in T32 and A32 Advanced SIMD instructions*.

The I8, I64, and F32 data types are permitted as pseudo-instructions, if the immediate can be represented by this instruction, and are encoded using a permitted encoding of the I16 or I32 data type.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        D[d+r] = D[d+r] AND NOT(imm64);
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.

    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

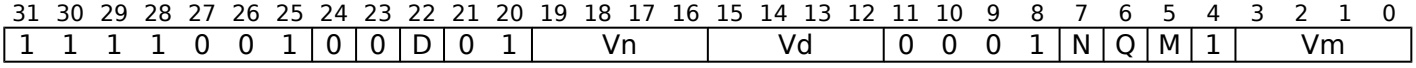
VBIC (register)

Vector Bitwise Bit Clear (register) performs a bitwise AND between a register value and the complement of a register value, and places the result in the destination register.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



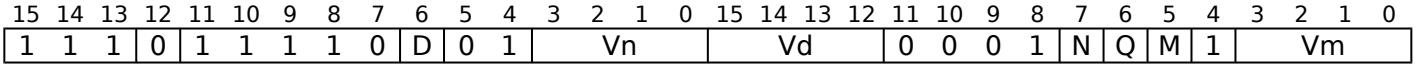
64-bit SIMD vector (Q == 0)

```
VBIC{<c>}{<q>}{.<dt>} {<Dd>,} <Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VBIC{<c>}{<q>}{.<dt>} {<Qd>,} <Qn>, <Qm>

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1



64-bit SIMD vector (Q == 0)

```
VBIC{<c>}{<q>}{.<dt>} {<Dd>,} <Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VBIC{<c>}{<q>}{.<dt>} {<Qd>,} <Qn>, <Qm>

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>	For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional. For encoding T1: see *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<dt>	An optional data type. It is ignored by assemblers, and does not affect the encoding.
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qn>	Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
<Qm>	Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        D[d+r] = D[n+r] AND NOT(D[m+r]);
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

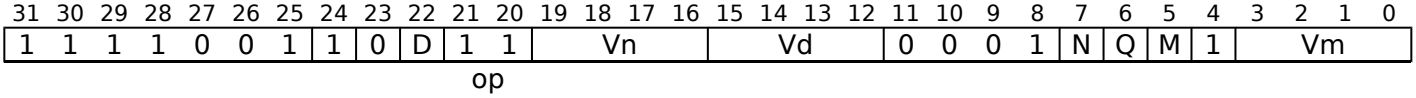
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VBIF

Vector Bitwise Insert if False inserts each bit from the first source register into the destination register if the corresponding bit of the second source register is 0, otherwise leaves the bit in the destination register unchanged.
Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



64-bit SIMD vector (Q == 0)

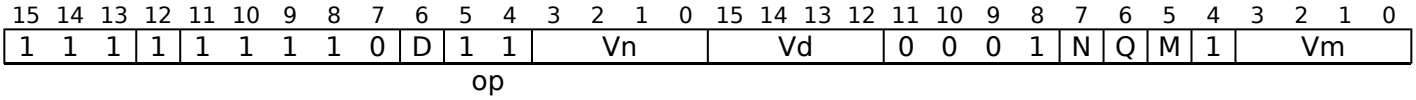
```
VBIF{<c>}{<q>}{.<dt>} {<Dd>,} <Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VBIF{<c>}{<q>}{.<dt>} {<Qd>,} <Qn>, <Qm>
```

```
enumeration VBitOps {VBitOps_VBIF, VBitOps_VBIT, VBitOps_VBSL};

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if op == '00' then SEE "VEOR";
if op == '01' then operation = VBitOps_VBSL;
if op == '10' then operation = VBitOps_VBIT;
if op == '11' then operation = VBitOps_VBIF;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1



64-bit SIMD vector (Q == 0)

```
VBIF{<c>}{<q>}{.<dt>} {<Dd>,} <Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VBIF{<c>}{<q>}{.<dt>} {<Qd>,} <Qn>, <Qm>
```

```
enumeration VBitOps {VBitOps_VBIF, VBitOps_VBIT, VBitOps_VBSL};

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if op == '00' then SEE "VEOR";
if op == '01' then operation = VBitOps_VBSL;
if op == '10' then operation = VBitOps_VBIT;
if op == '11' then operation = VBitOps_VBIF;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.
For encoding T1: see *Standard assembler syntax fields*.

<q>	See *Standard assembler syntax fields*.
<dt>	An optional data type. It is ignored by assemblers, and does not affect the encoding.
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qn>	Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
<Qm>	Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        case operation of
            when VBitOps_VBIF  D[d+r] = (D[d+r] AND D[m+r]) OR (D[n+r] AND NOT(D[m+r]));
            when VBitOps_VBIT  D[d+r] = (D[n+r] AND D[m+r]) OR (D[d+r] AND NOT(D[m+r]));
            when VBitOps_VBSL  D[d+r] = (D[n+r] AND D[d+r]) OR (D[m+r] AND NOT(D[d+r]));
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

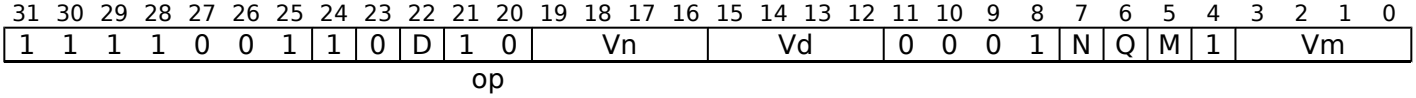
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VBIT

Vector Bitwise Insert if True inserts each bit from the first source register into the destination register if the corresponding bit of the second source register is 1, otherwise leaves the bit in the destination register unchanged.
Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



64-bit SIMD vector (Q == 0)

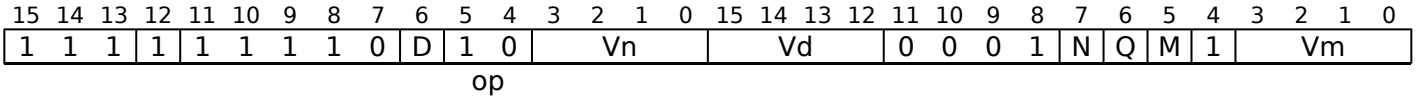
```
VBIT{<c>}{<q>}{.<dt>} {<Dd>,} <Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VBIT{<c>}{<q>}{.<dt>} {<Qd>,} <Qn>, <Qm>
```

```
enumeration VBitOps {VBitOps_VBIF, VBitOps_VBIT, VBitOps_VBSL};

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if op == '00' then SEE "VEOR";
if op == '01' then operation = VBitOps_VBSL;
if op == '10' then operation = VBitOps_VBIT;
if op == '11' then operation = VBitOps_VBIF;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1



64-bit SIMD vector (Q == 0)

```
VBIT{<c>}{<q>}{.<dt>} {<Dd>,} <Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VBIT{<c>}{<q>}{.<dt>} {<Qd>,} <Qn>, <Qm>
```

```
enumeration VBitOps {VBitOps_VBIF, VBitOps_VBIT, VBitOps_VBSL};

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if op == '00' then SEE "VEOR";
if op == '01' then operation = VBitOps_VBSL;
if op == '10' then operation = VBitOps_VBIT;
if op == '11' then operation = VBitOps_VBIF;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.
For encoding T1: see *Standard assembler syntax fields*.

<q>	See *Standard assembler syntax fields*.
<dt>	An optional data type. It is ignored by assemblers, and does not affect the encoding.
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qn>	Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
<Qm>	Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        case operation of
            when VBitOps_VBIF  D[d+r] = (D[d+r] AND D[m+r]) OR (D[n+r] AND NOT(D[m+r]));
            when VBitOps_VBIT  D[d+r] = (D[n+r] AND D[m+r]) OR (D[d+r] AND NOT(D[m+r]));
            when VBitOps_VBSL  D[d+r] = (D[n+r] AND D[d+r]) OR (D[m+r] AND NOT(D[d+r]));
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

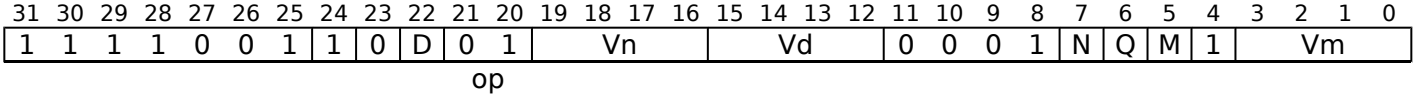
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VBSL

Vector Bitwise Select sets each bit in the destination to the corresponding bit from the first source operand when the original destination bit was 1, otherwise from the second source operand.
Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



64-bit SIMD vector (Q == 0)

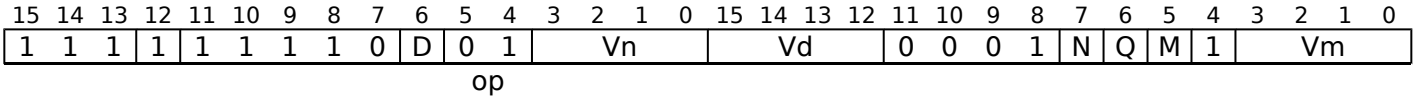
```
VBSL{<c>}{<q>}{.<dt>} {<Dd>,} <Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VBSL{<c>}{<q>}{.<dt>} {<Qd>,} <Qn>, <Qm>
```

```
enumeration VBitOps {VBitOps_VBIF, VBitOps_VBIT, VBitOps_VBSL};

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if op == '00' then SEE "VEOR";
if op == '01' then operation = VBitOps_VBSL;
if op == '10' then operation = VBitOps_VBIT;
if op == '11' then operation = VBitOps_VBIF;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1



64-bit SIMD vector (Q == 0)

```
VBSL{<c>}{<q>}{.<dt>} {<Dd>,} <Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VBSL{<c>}{<q>}{.<dt>} {<Qd>,} <Qn>, <Qm>
```

```
enumeration VBitOps {VBitOps_VBIF, VBitOps_VBIT, VBitOps_VBSL};

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if op == '00' then SEE "VEOR";
if op == '01' then operation = VBitOps_VBSL;
if op == '10' then operation = VBitOps_VBIT;
if op == '11' then operation = VBitOps_VBIF;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.
For encoding T1: see *Standard assembler syntax fields*.

<q>	See *Standard assembler syntax fields*.
<dt>	An optional data type. It is ignored by assemblers, and does not affect the encoding.
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qn>	Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
<Qm>	Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        case operation of
            when VBitOps_VBIF  D[d+r] = (D[d+r] AND D[m+r]) OR (D[n+r] AND NOT(D[m+r]));
            when VBitOps_VBIT  D[d+r] = (D[n+r] AND D[m+r]) OR (D[d+r] AND NOT(D[m+r]));
            when VBitOps_VBSL  D[d+r] = (D[n+r] AND D[d+r]) OR (D[m+r] AND NOT(D[d+r]));
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VCADD

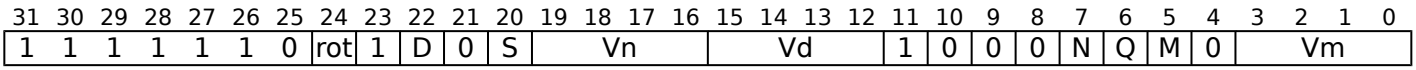
Vector Complex Add.
This instruction operates on complex numbers that are represented in SIMD&FP registers as pairs of elements, with the more significant element holding the imaginary part of the number and the less significant element holding the real part of the number. Each element holds a floating-point value. It performs the following computation on the corresponding complex number element pairs from the two source registers:

- Considering the complex number from the second source register on an Argand diagram, the number is rotated counterclockwise by 90 or 270 degrees.
- The rotated complex number is added to the complex number from the first source register.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1
(FEAT_FCMA)



64-bit SIMD vector (Q == 0)

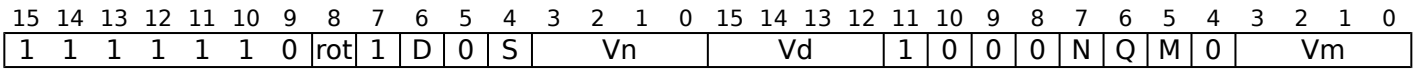
```
VCADD{<q>}.<dt> <Dd>, <Dn>, <Dm>, #<rotate>
```

128-bit SIMD vector (Q == 1)

```
VCADD{<q>}.<dt> <Qd>, <Qn>, <Qm>, #<rotate>
```

```
if !HaveFCADDExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
esize = 16 << UInt(S);
if !HaveFP16Ext() && esize == 16 then UNDEFINED;
elements = 64 DIV esize;
regs = if Q == '0' then 1 else 2;
```

T1
(FEAT_FCMA)



64-bit SIMD vector (Q == 0)

```
VCADD{<q>}.<dt> <Dd>, <Dn>, <Dm>, #<rotate>
```

128-bit SIMD vector (Q == 1)

```
VCADD{<q>}.<dt> <Qd>, <Qn>, <Qm>, #<rotate>
```

```
if InITBlock() then UNPREDICTABLE;
if !HaveFCADDExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
esize = 16 << UInt(S);
if !HaveFP16Ext() && esize == 16 then UNDEFINED;
elements = 64 DIV esize;
regs = if Q == '0' then 1 else 2;
```

Assembler Symbols

<q> See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the vectors, encoded in “S”:

S	<dt>
0	F16
1	F32

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

<rotate> Is the rotation to be applied to elements in the second SIMD&FP source register, encoded in “rot”:

rot	<rotate>
0	90
1	270

Operation

```
EncodingSpecificOperations();
CheckAdvSIMDEnabled();
for r = 0 to regs-1
    operand1 = D[n+r];
    operand2 = D[m+r];
    operand3 = D[d+r];
    for e = 0 to (elements DIV 2)-1
        case rot of
            when '0'
                element1 = FPNeg(Elem[operand2,e*2+1,esize]);
                element3 = Elem[operand2,e*2,esize];
            when '1'
                element1 = Elem[operand2,e*2+1,esize];
                element3 = FPNeg(Elem[operand2,e*2,esize]);
        result1 = FPAdd(Elem[operand1,e*2,esize],element1,StandardFPSCRValue());
        result2 = FPAdd(Elem[operand1,e*2+1,esize],element3,StandardFPSCRValue());
        Elem[D[d+r],e*2,esize] = result1;
        Elem[D[d+r],e*2+1,esize] = result2;
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VCEQ (immediate #0)

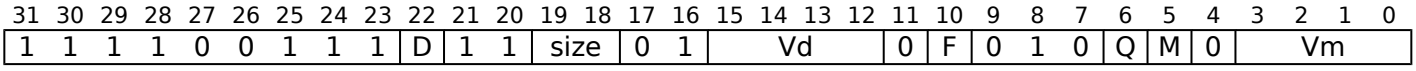
Vector Compare Equal to Zero takes each element in a vector, and compares it with zero. If it is equal to zero, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

The operand vector elements are the same type, and are integers or floating-point numbers. The result vector elements are fields the same size as the operand vector elements.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



64-bit SIMD vector (Q == 0)

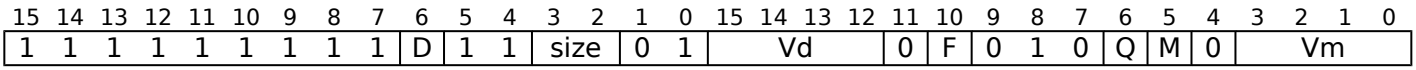
```
VCEQ{<c>}{<q>}.<dt> {<Dd>,} <Dm>, #0
```

128-bit SIMD vector (Q == 1)

```
VCEQ{<c>}{<q>}.<dt> {<Qd>,} <Qm>, #0
```

```
if size == '11' then UNDEFINED;
if F == '1' && ((size == '01' && !HaveFP16Ext()) || size == '00') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
floating_point = (F == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1



64-bit SIMD vector (Q == 0)

```
VCEQ{<c>}{<q>}.<dt> {<Dd>,} <Dm>, #0
```

128-bit SIMD vector (Q == 1)

```
VCEQ{<c>}{<q>}.<dt> {<Qd>,} <Qm>, #0
```

```
if size == '11' then UNDEFINED;
if F == '1' && ((size == '01' && !HaveFP16Ext()) || size == '00') then UNDEFINED;
if F == '1' && size == '01' && InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
floating_point = (F == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `F == '1' && size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.
For encoding T1: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the operands, encoded in “F:size”:

F	size	<dt>
0	00	I8
0	01	I16
0	10	I32
1	01	F16
1	10	F32

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            if floating_point then
                bits(esize) zero = FPZero('0');
                test_passed = FPCompareEQ(Elem[D[m+r],e,esize], zero, StandardFPSCRValue());
            else
                test_passed = (Elem[D[m+r],e,esize] == Zeros(esize));
            Elem[D[d+r],e,esize] = if test_passed then Ones(esize) else Zeros(esize);
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VCEQ (register)

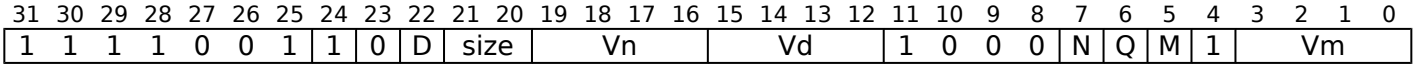
Vector Compare Equal takes each element in a vector, and compares it with the corresponding element of a second vector. If they are equal, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

The operand vector elements are the same type, and are integers or floating-point numbers. The result vector elements are fields the same size as the operand vector elements.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2) .

A1



64-bit SIMD vector (Q == 0)

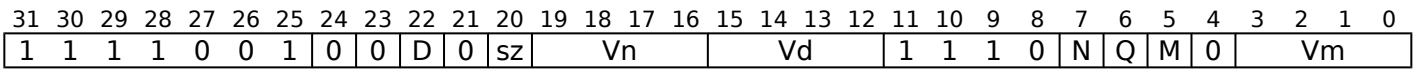
```
VCEQ{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VCEQ{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
int_operation = TRUE;  esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

A2



64-bit SIMD vector (Q == 0)

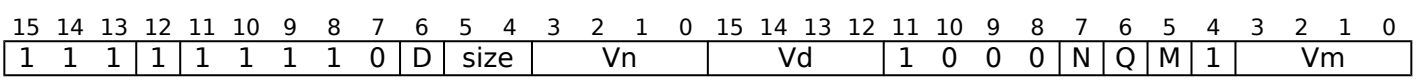
```
VCEQ{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VCEQ{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
int_operation = FALSE;
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1



64-bit SIMD vector (Q == 0)

```
VCEQ{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VCEQ{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
int_operation = TRUE;  esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	1	0	D	0	sz	Vn				Vd				1	1	1	0	N	Q	M	0	Vm			

64-bit SIMD vector (Q == 0)

```
VCEQ{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VCEQ{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
int_operation = FALSE;
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `sz == '1' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c> For encoding A1 and A2: see *Standard assembler syntax fields*. This encoding must be unconditional.

For encoding T1 and T2: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<dt> For encoding A1 and T1: is the data type for the elements of the vectors, encoded in “size”:

size	<dt>
00	I8
01	I16
10	I32

For encoding A2 and T2: is the data type for the elements of the vectors, encoded in “sz”:

sz	<dt>
0	F32
1	F16

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[D[n+r],e,esize];  op2 = Elem[D[m+r],e,esize];
            if int_operation then
                test_passed = (op1 == op2);
            else
                test_passed = FPCompareEQ(op1, op2, StandardFPSCRValue());
            Elem[D[d+r],e,esize] = if test_passed then Ones(esize) else Zeros(esize);
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VCGE (immediate #0)

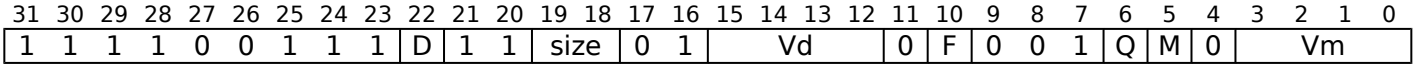
Vector Compare Greater Than or Equal to Zero takes each element in a vector, and compares it with zero. If it is greater than or equal to zero, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

The operand vector elements are the same type, and are signed integers or floating-point numbers. The result vector elements are fields the same size as the operand vector elements.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



64-bit SIMD vector (Q == 0)

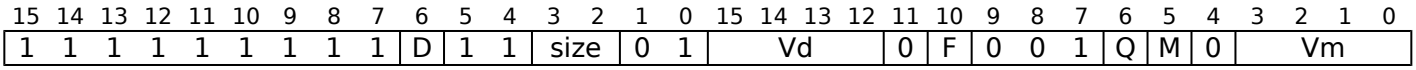
```
VCGE{<c>}{<q>}.<dt> {<Dd>,} <Dm>, #0
```

128-bit SIMD vector (Q == 1)

```
VCGE{<c>}{<q>}.<dt> {<Qd>,} <Qm>, #0
```

```
if size == '11' then UNDEFINED;
if F == '1' && ((size == '01' && !HaveFP16Ext()) || size == '00') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
floating_point = (F == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1



64-bit SIMD vector (Q == 0)

```
VCGE{<c>}{<q>}.<dt> {<Dd>,} <Dm>, #0
```

128-bit SIMD vector (Q == 1)

```
VCGE{<c>}{<q>}.<dt> {<Qd>,} <Qm>, #0
```

```
if size == '11' then UNDEFINED;
if F == '1' && ((size == '01' && !HaveFP16Ext()) || size == '00') then UNDEFINED;
if F == '1' && size == '01' && InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
floating_point = (F == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `F == '1' && size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the operands, encoded in “F:size”:

F	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	01	F16
1	10	F32

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            if floating_point then
                bits(esize) zero = FPZero('0');
                test_passed = FPCompareGE(Elem[D[m+r],e,esize], zero, StandardFPSCRValue());
            else
                test_passed = (SInt(Elem[D[m+r],e,esize]) >= 0);
            Elem[D[d+r],e,esize] = if test_passed then Ones(esize) else Zeros(esize);
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check and is operating only on integer vector elements, then the following apply:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VCGE (register)

Vector Compare Greater Than or Equal takes each element in a vector, and compares it with the corresponding element of a second vector. If the first is greater than or equal to the second, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

The operand vector elements are the same type, and are signed integers, unsigned integers, or floating-point numbers. The result vector elements are fields the same size as the operand vector elements.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

This instruction is used by the pseudo-instruction VCLE (register).

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	0	D	size		Vn				Vd				0	0	1	1	N	Q	M	1	Vm			

64-bit SIMD vector (Q == 0)

```
VCGE{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VCGE{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
vtype = if U == '1' then VCGEType_unsigned else VCGEType_signed;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

A2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	0	D	0	sz	Vn				Vd				1	1	1	0	N	Q	M	0	Vm			

64-bit SIMD vector (Q == 0)

```
VCGE{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VCGE{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
vtype = VCGEType_fp;
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	0	D	size		Vn				Vd				0	0	1	1	N	Q	M	1	Vm			

64-bit SIMD vector (Q == 0)

```
VCGE{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VCGE{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
vtype = if U == '1' then VCGEType_unsigned else VCGEType_signed;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	0	D	0	sz	Vn				Vd				1	1	1	0	N	Q	M	0	Vm			

64-bit SIMD vector (Q == 0)

```
VCGE{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VCGE{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
vtype = VCGEType_fp;
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `sz == '1' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c> For encoding A1 and A2: see *Standard assembler syntax fields*. This encoding must be unconditional.
For encoding T1 and T2: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<dt> For encoding A1 and T1: is the data type for the elements of the operands, encoded in “U:size”:

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	00	U8
1	01	U16
1	10	U32

For encoding A2 and T2: is the data type for the elements of the vectors, encoded in “sz”:

sz	<dt>
0	F32
1	F16

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[D[n+r],e,esize];  op2 = Elem[D[m+r],e,esize];
            case vtype of
                when VCGEType_signed    test_passed = (SInt(op1) >= SInt(op2));
                when VCGEType_unsigned  test_passed = (UInt(op1) >= UInt(op2));
                when VCGEType_fp        test_passed = FPCompareGE(op1, op2, StandardFPSCRValue());
            Elem[D[d+r],e,esize] = if test_passed then Ones(esize) else Zeros(esize);
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check and is operating only on integer vector elements, then the following apply:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VCGT (immediate #0)

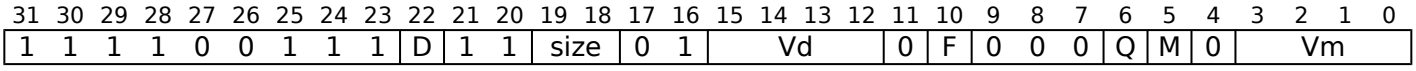
Vector Compare Greater Than Zero takes each element in a vector, and compares it with zero. If it is greater than zero, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

The operand vector elements are the same type, and are signed integers or floating-point numbers. The result vector elements are fields the same size as the operand vector elements.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



64-bit SIMD vector (Q == 0)

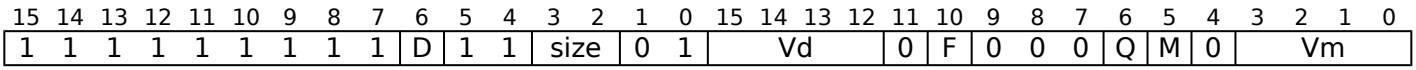
```
VCGT{<c>}{<q>}.<dt> {<Dd>,} <Dm>, #0
```

128-bit SIMD vector (Q == 1)

```
VCGT{<c>}{<q>}.<dt> {<Qd>,} <Qm>, #0
```

```
if size == '11' then UNDEFINED;
if F == '1' && ((size == '01' && !HaveFP16Ext()) || size == '00') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
floating_point = (F == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1



64-bit SIMD vector (Q == 0)

```
VCGT{<c>}{<q>}.<dt> {<Dd>,} <Dm>, #0
```

128-bit SIMD vector (Q == 1)

```
VCGT{<c>}{<q>}.<dt> {<Qd>,} <Qm>, #0
```

```
if size == '11' then UNDEFINED;
if F == '1' && ((size == '01' && !HaveFP16Ext()) || size == '00') then UNDEFINED;
if F == '1' && size == '01' && InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
floating_point = (F == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `F == '1' && size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the operands, encoded in “F:size”:

F	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	01	F16
1	10	F32

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            if floating_point then
                bits(esize) zero = FPZero('0');
                test_passed = FPCompareGT(Elem[D[m+r],e,esize], zero, StandardFPSCRValue());
            else
                test_passed = (SInt(Elem[D[m+r],e,esize]) > 0);
            Elem[D[d+r],e,esize] = if test_passed then Ones(esize) else Zeros(esize);
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check and is operating only on integer vector elements, then the following apply:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VCGT (register)

Vector Compare Greater Than takes each element in a vector, and compares it with the corresponding element of a second vector. If the first is greater than the second, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

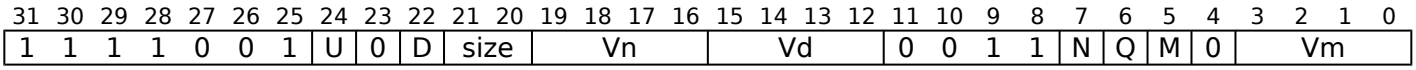
The operand vector elements are the same type, and are signed integers, unsigned integers, or floating-point numbers. The result vector elements are fields the same size as the operand vector elements.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

This instruction is used by the pseudo-instruction VCLT (register).

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2) .

A1



64-bit SIMD vector (Q == 0)

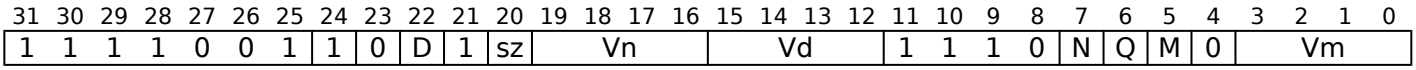
```
VCGT{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VCGT{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
vtype = if U == '1' then VCGTtype_unsigned else VCGTtype_signed;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

A2



64-bit SIMD vector (Q == 0)

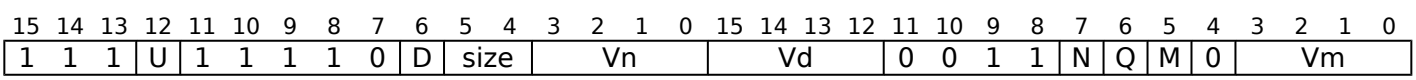
```
VCGT{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VCGT{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
vtype = VCGTtype_fp;
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1



64-bit SIMD vector (Q == 0)

```
VCGT{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VCGT{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
vtype = if U == '1' then VCGTtype_unsigned else VCGTtype_signed;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	0	D	1	sz	Vn				Vd				1	1	1	0	N	Q	M	0	Vm			

64-bit SIMD vector (Q == 0)

```
VCGT{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VCGT{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
vtype = VCGTtype_fp;
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `sz == '1' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c> For encoding A1 and A2: see *Standard assembler syntax fields*. This encoding must be unconditional.
For encoding T1 and T2: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<dt> For encoding A1 and T1: is the data type for the elements of the operands, encoded in “U:size”:

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	00	U8
1	01	U16
1	10	U32

For encoding A2 and T2: is the data type for the elements of the vectors, encoded in “sz”:

sz	<dt>
0	F32
1	F16

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
enumeration VCGTtype {VCGTtype_signed, VCGTtype_unsigned, VCGTtype_fp};

if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[D[n+r],e,esize];  op2 = Elem[D[m+r],e,esize];
            case vtype of
                when VCGTtype_signed    test_passed = (SInt(op1) > SInt(op2));
                when VCGTtype_unsigned  test_passed = (UInt(op1) > UInt(op2));
                when VCGTtype_fp        test_passed = FPCompareGT(op1, op2, StandardFPSCRValue());
            Elem[D[d+r],e,esize] = if test_passed then Ones(esize) else Zeros(esize);
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check and is operating only on integer vector elements, then the following apply:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

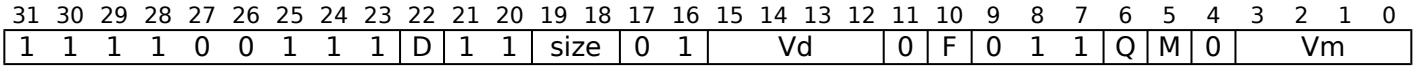
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VCLE (immediate #0)

Vector Compare Less Than or Equal to Zero takes each element in a vector, and compares it with zero. If it is less than or equal to zero, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.
The operand vector elements are the same type, and are signed integers or floating-point numbers. The result vector elements are fields the same size as the operand vector elements.
Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



64-bit SIMD vector (Q == 0)

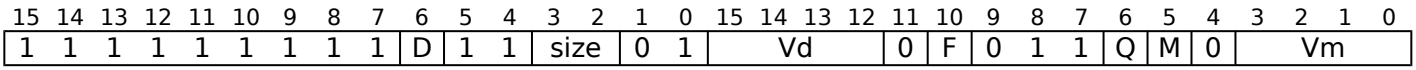
```
VCLE{<c>}{<q>}.<dt> {<Dd>,} <Dm>, #0
```

128-bit SIMD vector (Q == 1)

```
VCLE{<c>}{<q>}.<dt> {<Qd>,} <Qm>, #0
```

```
if size == '11' then UNDEFINED;
if F == '1' && ((size == '01' && !HaveFP16Ext()) || size == '00') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
floating_point = (F == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1



64-bit SIMD vector (Q == 0)

```
VCLE{<c>}{<q>}.<dt> {<Dd>,} <Dm>, #0
```

128-bit SIMD vector (Q == 1)

```
VCLE{<c>}{<q>}.<dt> {<Qd>,} <Qm>, #0
```

```
if size == '11' then UNDEFINED;
if F == '1' && ((size == '01' && !HaveFP16Ext()) || size == '00') then UNDEFINED;
if F == '1' && size == '01' && InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
floating_point = (F == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `F == '1' && size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the operands, encoded in “F:size”:

F	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	01	F16
1	10	F32

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            if floating_point then
                bits(esize) zero = FPZero('0');
                test_passed = FPCompareGE(zero, Elem[D[m+r],e,esize], StandardFPSCRValue());
            else
                test_passed = (SInt(Elem[D[m+r],e,esize]) <= 0);
            Elem[D[d+r],e,esize] = if test_passed then Ones(esize) else Zeros(esize);
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check and is operating only on integer vector elements, then the following apply:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VCLE (register)

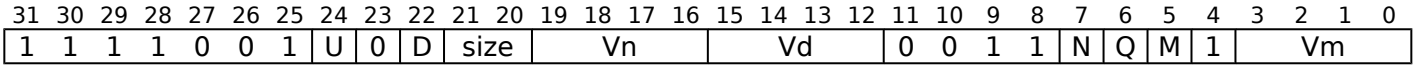
Vector Compare Less Than or Equal takes each element in a vector, and compares it with the corresponding element of a second vector. If the first is less than or equal to the second, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

This is a pseudo-instruction of VCGE (register). This means:

- The encodings in this description are named to match the encodings of VCGE (register).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of VCGE (register) gives the operational pseudocode for this instruction.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2) .

A1



64-bit SIMD vector (Q == 0)

```
VCLE{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

is equivalent to

```
VCGE{<c>}{<q>}.<dt> <Dd>, <Dm>, <Dn>
```

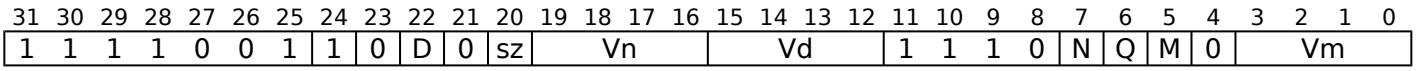
128-bit SIMD vector (Q == 1)

```
VCLE{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

is equivalent to

```
VCGE{<c>}{<q>}.<dt> <Qd>, <Qm>, <Qn>
```

A2



64-bit SIMD vector (Q == 0)

```
VCLE{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

is equivalent to

```
VCGE{<c>}{<q>}.<dt> <Dd>, <Dm>, <Dn>
```

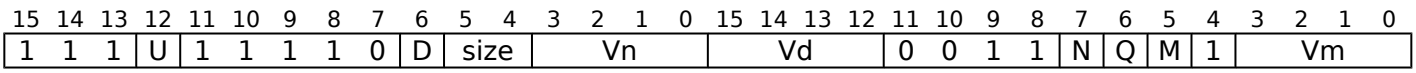
128-bit SIMD vector (Q == 1)

```
VCLE{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

is equivalent to

```
VCGE{<c>}{<q>}.<dt> <Qd>, <Qm>, <Qn>
```

T1



64-bit SIMD vector (Q == 0)

```
VCLE{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

is equivalent to

```
VCGE{<c>}{<q>}.<dt> <Dd>, <Dm>, <Dn>
```

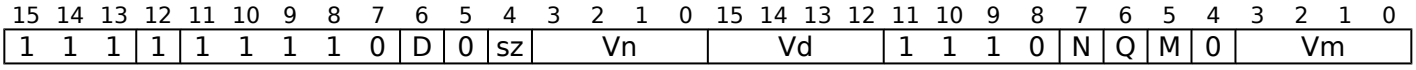
128-bit SIMD vector (Q == 1)

```
VCLE{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

is equivalent to

```
VCGE{<c>}{<q>}.<dt> <Qd>, <Qm>, <Qn>
```

T2



64-bit SIMD vector (Q == 0)

```
VCLE{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

is equivalent to

```
VCGE{<c>}{<q>}.<dt> <Dd>, <Dm>, <Dn>
```

128-bit SIMD vector (Q == 1)

```
VCLE{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

is equivalent to

```
VCGE{<c>}{<q>}.<dt> <Qd>, <Qm>, <Qn>
```

Assembler Symbols

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<c> For encoding A1 and A2: see *Standard assembler syntax fields*. This encoding must be unconditional.
For encoding T1 and T2: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<dt> For encoding A1 and T1: is the data type for the elements of the operands, encoded in “U:size”:

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	00	U8
1	01	U16
1	10	U32

For encoding A2 and T2: is the data type for the elements of the vectors, encoded in “sz”:

sz	<dt>
0	F32
1	F16

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

Operation

The description of VCGE (register) gives the operational pseudocode for this instruction.

Operational information

The description of *VCGE_r* gives the operational pseudocode for this instruction.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VCLS

Vector Count Leading Sign Bits counts the number of consecutive bits following the topmost bit, that are the same as the topmost bit, in each element in a vector, and places the results in a second vector. The count does not include the topmost bit itself.

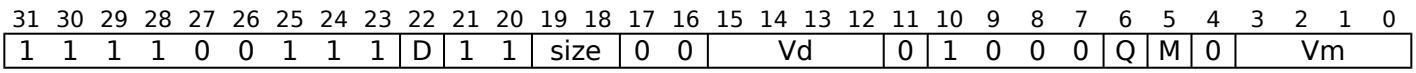
The operand vector elements can be any one of 8-bit, 16-bit, or 32-bit signed integers.

The result vector elements are the same data type as the operand vector elements.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



64-bit SIMD vector (Q == 0)

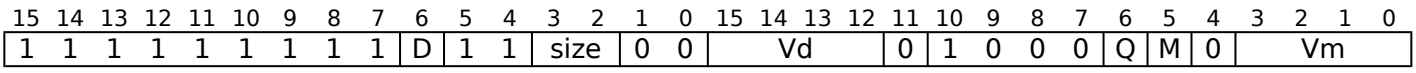
```
VCLS{<c>}{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VCLS{<c>}{<q>}.<dt> <Qd>, <Qm>
```

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1



64-bit SIMD vector (Q == 0)

```
VCLS{<c>}{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VCLS{<c>}{<q>}.<dt> <Qd>, <Qm>
```

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>	For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional. For encoding T1: see *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<dt>	Is the data type for the elements of the operands, encoded in “size”:

size	<dt>
00	S8
01	S16
10	S32
11	RESERVED

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            Elem[D[d+r],e,esize] = CountLeadingSignBits(Elem[D[m+r],e,esize])<esize-1:0>;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VCLT (immediate #0)

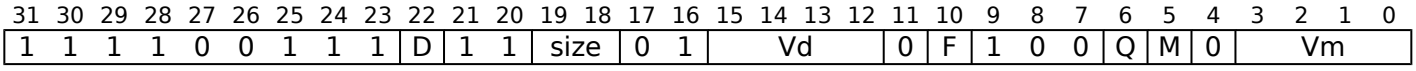
Vector Compare Less Than Zero takes each element in a vector, and compares it with zero. If it is less than zero, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

The operand vector elements are the same type, and are signed integers or floating-point numbers. The result vector elements are fields the same size as the operand vector elements.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



64-bit SIMD vector (Q == 0)

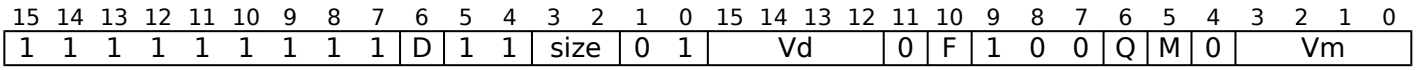
```
VCLT{<c>}{<q>}.<dt> {<Dd>,} <Dm>, #0
```

128-bit SIMD vector (Q == 1)

```
VCLT{<c>}{<q>}.<dt> {<Qd>,} <Qm>, #0
```

```
if size == '11' then UNDEFINED;
if F == '1' && ((size == '01' && !HaveFP16Ext()) || size == '00') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
floating_point = (F == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1



64-bit SIMD vector (Q == 0)

```
VCLT{<c>}{<q>}.<dt> {<Dd>,} <Dm>, #0
```

128-bit SIMD vector (Q == 1)

```
VCLT{<c>}{<q>}.<dt> {<Qd>,} <Qm>, #0
```

```
if size == '11' then UNDEFINED;
if F == '1' && ((size == '01' && !HaveFP16Ext()) || size == '00') then UNDEFINED;
if F == '1' && size == '01' && InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
floating_point = (F == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `F == '1' && size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the operands, encoded in “F:size”:

F	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	01	F16
1	10	F32

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            if floating_point then
                bits(esize) zero = FPZero('0');
                test_passed = FPCompareGT(zero, Elem[D[m+r],e,esize], StandardFPSCRValue());
            else
                test_passed = (SInt(Elem[D[m+r],e,esize]) < 0);
            Elem[D[d+r],e,esize] = if test_passed then Ones(esize) else Zeros(esize);
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check and is operating only on integer vector elements, then the following apply:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VCLT (register)

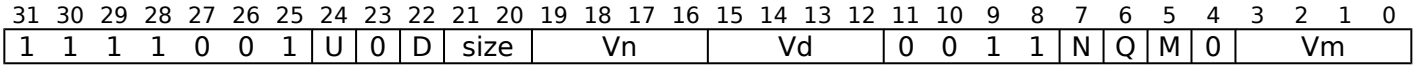
Vector Compare Less Than takes each element in a vector, and compares it with the corresponding element of a second vector. If the first is less than the second, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

This is a pseudo-instruction of VCGT (register). This means:

- The encodings in this description are named to match the encodings of VCGT (register).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of VCGT (register) gives the operational pseudocode for this instruction.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2) .

A1



64-bit SIMD vector (Q == 0)

```
VCLT{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

is equivalent to

```
VCGT{<c>}{<q>}.<dt> <Dd>, <Dm>, <Dn>
```

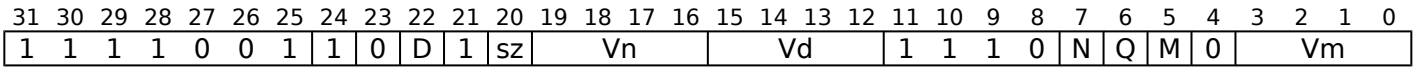
128-bit SIMD vector (Q == 1)

```
VCLT{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

is equivalent to

```
VCGT{<c>}{<q>}.<dt> <Qd>, <Qm>, <Qn>
```

A2



64-bit SIMD vector (Q == 0)

```
VCLT{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

is equivalent to

```
VCGT{<c>}{<q>}.<dt> <Dd>, <Dm>, <Dn>
```

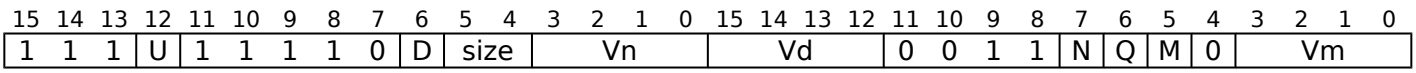
128-bit SIMD vector (Q == 1)

```
VCLT{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

is equivalent to

```
VCGT{<c>}{<q>}.<dt> <Qd>, <Qm>, <Qn>
```

T1



64-bit SIMD vector (Q == 0)

```
VCLT{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

is equivalent to

```
VCGT{<c>}{<q>}.<dt> <Dd>, <Dm>, <Dn>
```

128-bit SIMD vector (Q == 1)

```
VCLT{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

is equivalent to

```
VCGT{<c>}{<q>}.<dt> <Qd>, <Qm>, <Qn>
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	0	D	1	sz	Vn				Vd				1	1	1	0	N	Q	M	0	Vm			

64-bit SIMD vector (Q == 0)

```
VCLT{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

is equivalent to

```
VCGT{<c>}{<q>}.<dt> <Dd>, <Dm>, <Dn>
```

128-bit SIMD vector (Q == 1)

```
VCLT{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

is equivalent to

```
VCGT{<c>}{<q>}.<dt> <Qd>, <Qm>, <Qn>
```

Assembler Symbols

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<c> For encoding A1 and A2: see *Standard assembler syntax fields*. This encoding must be unconditional.
For encoding T1 and T2: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<dt> For encoding A1 and T1: is the data type for the elements of the operands, encoded in “U:size”:

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	00	U8
1	01	U16
1	10	U32

For encoding A2 and T2: is the data type for the elements of the vectors, encoded in “sz”:

sz	<dt>
0	F32
1	F16

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

Operation

The description of VCGT (register) gives the operational pseudocode for this instruction.

Operational information

The description of *VCGT_r* gives the operational pseudocode for this instruction.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VCLZ

Vector Count Leading Zeros counts the number of consecutive zeros, starting from the most significant bit, in each element in a vector, and places the results in a second vector.

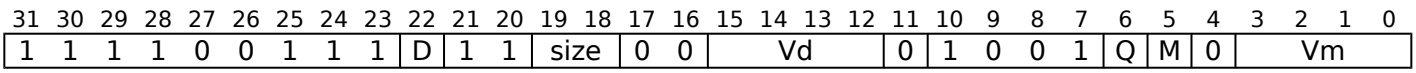
The operand vector elements can be any one of 8-bit, 16-bit, or 32-bit integers. There is no distinction between signed and unsigned integers.

The result vector elements are the same data type as the operand vector elements.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



64-bit SIMD vector (Q == 0)

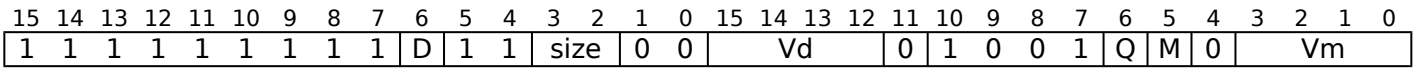
```
VCLZ{<c>}{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VCLZ{<c>}{<q>}.<dt> <Qd>, <Qm>
```

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1



64-bit SIMD vector (Q == 0)

```
VCLZ{<c>}{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VCLZ{<c>}{<q>}.<dt> <Qd>, <Qm>
```

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>	For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional. For encoding T1: see *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<dt>	Is the data type for the elements of the operands, encoded in “size”:

size	<dt>
00	I8
01	I16
10	I32
11	RESERVED

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            Elem[D[d+r],e,esize] = CountLeadingZeroBits(Elem[D[m+r],e,esize])<esize-1:0>;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VCMLA

Vector Complex Multiply Accumulate.

This instruction operates on complex numbers that are represented in SIMD&FP registers as pairs of elements, with the more significant element holding the imaginary part of the number and the less significant element holding the real part of the number. Each element holds a floating-point value. It performs the following computation on the corresponding complex number element pairs from the two source registers and the destination register:

- Considering the complex number from the second source register on an Argand diagram, the number is rotated counterclockwise by 0, 90, 180, or 270 degrees.
- The two elements of the transformed complex number are multiplied by:
  - The real element of the complex number from the first source register, if the transformation was a rotation by 0 or 180 degrees.
  - The imaginary element of the complex number from the first source register, if the transformation was a rotation by 90 or 270 degrees.
- The complex number resulting from that multiplication is added to the complex number from the destination register.

The multiplication and addition operations are performed as a fused multiply-add, without any intermediate rounding.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

(FEAT_FCMA)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	0	rot		D	1	S	Vn				Vd				1	0	0	0	N	Q	M	0	Vm			

64-bit SIMD vector (Q == 0)

```
VCMLA{<q>}.<dt> <Dd>, <Dn>, <Dm>, #<rotate>
```

128-bit SIMD vector (Q == 1)

```
VCMLA{<q>}.<dt> <Qd>, <Qn>, <Qm>, #<rotate>
```

```
if !HaveFCADDExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
esize = 16 << UInt(S);
if !HaveFP16Ext() && esize == 16 then UNDEFINED;
elements = 64 DIV esize;
regs = if Q == '0' then 1 else 2;
```

T1

(FEAT_FCMA)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	0	rot		D	1	S	Vn				Vd				1	0	0	0	N	Q	M	0	Vm			

64-bit SIMD vector (Q == 0)

```
VCMLA{<q>}.<dt> <Dd>, <Dn>, <Dm>, #<rotate>
```

128-bit SIMD vector (Q == 1)

```
VCMLA{<q>}.<dt> <Qd>, <Qn>, <Qm>, #<rotate>
```

```
if InITBlock() then UNPREDICTABLE;
if !HaveFCADDExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
esize = 16 << UInt(S);
if !HaveFP16Ext() && esize == 16 then UNDEFINED;
elements = 64 DIV esize;
regs = if Q == '0' then 1 else 2;
```

Assembler Symbols

<q> See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the vectors, encoded in “S”:

S	<dt>
0	F16
1	F32

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

<rotate> Is the rotation to be applied to elements in the second SIMD&FP source register, encoded in “rot”:

rot	<rotate>
00	0
01	90
10	180
11	270

Operation

```
EncodingSpecificOperations();
CheckAdvSIMDEnabled();
for r = 0 to regs-1
    operand1 = D[n+r];
    operand2 = D[m+r];
    operand3 = D[d+r];
    for e = 0 to (elements DIV 2)-1
        case rot of
            when '00'
               element1 = Elem[operand2,e*2,esize];
               element2 = Elem[operand1,e*2,esize];
               element3 = Elem[operand2,e*2+1,esize];
               element4 = Elem[operand1,e*2,esize];
            when '01'
               element1 = FPNeg(Elem[operand2,e*2+1,esize]);
               element2 = Elem[operand1,e*2+1,esize];
               element3 = Elem[operand2,e*2,esize];
               element4 = Elem[operand1,e*2+1,esize];
            when '10'
               element1 = FPNeg(Elem[operand2,e*2,esize]);
               element2 = Elem[operand1,e*2,esize];
               element3 = FPNeg(Elem[operand2,e*2+1,esize]);
               element4 = Elem[operand1,e*2,esize];
            when '11'
               element1 = Elem[operand2,e*2+1,esize];
               element2 = Elem[operand1,e*2+1,esize];
               element3 = FPNeg(Elem[operand2,e*2,esize]);
               element4 = Elem[operand1,e*2+1,esize];
        result1 = FPMulAdd(Elem[operand3,e*2,esize],element2,element1, StandardFPSCRValue());
        result2 = FPMulAdd(Elem[operand3,e*2+1,esize],element4,element3, StandardFPSCRValue());
        Elem[D[d+r],e*2,esize] = result1;
        Elem[D[d+r],e*2+1,esize] = result2;
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VCMLA (by element)

Vector Complex Multiply Accumulate (by element).

This instruction operates on complex numbers that are represented in SIMD&FP registers as pairs of elements, with the more significant element holding the imaginary part of the number and the less significant element holding the real part of the number. Each element holds a floating-point value. It performs the following computation on complex numbers from the first source register and the destination register with the specified complex number from the second source register:

- Considering the complex number from the second source register on an Argand diagram, the number is rotated counterclockwise by 0, 90, 180, or 270 degrees.
- The two elements of the transformed complex number are multiplied by:
  - The real element of the complex number from the first source register, if the transformation was a rotation by 0 or 180 degrees.
  - The imaginary element of the complex number from the first source register, if the transformation was a rotation by 90 or 270 degrees.
- The complex number resulting from that multiplication is added to the complex number from the destination register.

The multiplication and addition operations are performed as a fused multiply-add, without any intermediate rounding.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

(FEAT_FCMA)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0	S	D	rot		Vn				Vd				1	0	0	0	N	Q	M	0	Vm			

64-bit SIMD vector of half-precision floating-point (S == 0 && Q == 0)

```
VCMLA{<q>}.F16 <Dd>, <Dn>, <Dm>[<index>], #<rotate>
```

64-bit SIMD vector of single-precision floating-point (S == 1 && Q == 0)

```
VCMLA{<q>}.F32 <Dd>, <Dn>, <Dm>[0], #<rotate>
```

128-bit SIMD vector of half-precision floating-point (S == 0 && Q == 1)

```
VCMLA{<q>}.F16 <Qd>, <Qn>, <Dm>[<index>], #<rotate>
```

128-bit SIMD vector of single-precision floating-point (S == 1 && Q == 1)

```
VCMLA{<q>}.F32 <Qd>, <Qn>, <Dm>[0], #<rotate>
```

```
if !HaveFCADDExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn);
m = if S=='1' then UInt(M:Vm) else UInt(Vm);
esize = 16 << UInt(S);
if !HaveFP16Ext() && esize == 16 then UNDEFINED;
elements = 64 DIV esize;
regs = if Q == '0' then 1 else 2;
index = if S=='1' then 0 else UInt(M);
```

T1

(FEAT_FCMA)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0	S	D	rot		Vn				Vd				1	0	0	0	N	Q	M	0	Vm			

64-bit SIMD vector of half-precision floating-point (S == 0 && Q == 0)

```
VCMLA{<q>}.F16 <Dd>, <Dn>, <Dm>[<index>], #<rotate>
```

64-bit SIMD vector of single-precision floating-point (S == 1 && Q == 0)

```
VCMLA{<q>}.F32 <Dd>, <Dn>, <Dm>[0], #<rotate>
```

128-bit SIMD vector of half-precision floating-point (S == 0 && Q == 1)

```
VCMLA{<q>}.F16 <Qd>, <Qn>, <Dm>[<index>], #<rotate>
```

128-bit SIMD vector of single-precision floating-point (S == 1 && Q == 1)

```
VCMLA{<q>}.F32 <Qd>, <Qn>, <Dm>[0], #<rotate>
```

```
if InITBlock() then UNPREDICTABLE;
if !HaveFCADDExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn);
m = if S=='1' then UInt(M:Vm) else UInt(Vm);
esize = 16 << UInt(S);
if !HaveFP16Ext() && esize == 16 then UNDEFINED;
elements = 64 DIV esize;
regs = if Q == '0' then 1 else 2;
index = if S=='1' then 0 else UInt(M);
```

Assembler Symbols

<q>	See *Standard assembler syntax fields*.
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qn>	Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<Dm>	For the half-precision scalar variant: is the 64-bit name of the second SIMD&FP source register, encoded in the "Vm" field. For the single-precision scalar variant: is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.
<index>	Is the element index in the range 0 to 1, encoded in the "M" field.
<rotate>	Is the rotation to be applied to elements in the second SIMD&FP source register, encoded in “rot”:

rot	<rotate>
00	0
01	90
10	180
11	270

Operation

```
EncodingSpecificOperations();
CheckAdvSIMDEnabled();
for r = 0 to regs-1
    operand1 = D[n+r];
    operand2 = Din[m];
    operand3 = D[d+r];
    for e = 0 to (elements DIV 2)-1
        case rot of
            when '00'
                element1 = Elem[operand2,index*2,esize];
                element2 = Elem[operand1,e*2,esize];
                element3 = Elem[operand2,index*2+1,esize];
                element4 = Elem[operand1,e*2,esize];
            when '01'
                element1 = FPNeg(Elem[operand2,index*2+1,esize]);
                element2 = Elem[operand1,e*2+1,esize];
                element3 = Elem[operand2,index*2,esize];
                element4 = Elem[operand1,e*2+1,esize];
            when '10'
                element1 = FPNeg(Elem[operand2,index*2,esize]);
                element2 = Elem[operand1,e*2,esize];
                element3 = FPNeg(Elem[operand2,index*2+1,esize]);
                element4 = Elem[operand1,e*2,esize];
            when '11'
                element1 = Elem[operand2,index*2+1,esize];
                element2 = Elem[operand1,e*2+1,esize];
                element3 = FPNeg(Elem[operand2,index*2,esize]);
                element4 = Elem[operand1,e*2+1,esize];
        result1 = FPMulAdd(Elem[operand3,e*2,esize],element2,element1, StandardFPSCRValue());
        result2 = FPMulAdd(Elem[operand3,e*2+1,esize],element4,element3,StandardFPSCRValue());
        Elem[D[d+r],e*2,esize] = result1;
        Elem[D[d+r],e*2+1,esize] = result2;
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VCMP

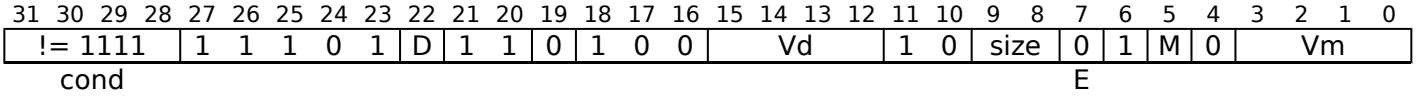
Vector Compare compares two floating-point registers, or one floating-point register and zero. It writes the result to the *FPSCR* flags. These are normally transferred to the *PSTATE*.{N, Z, C, V} Condition flags by a subsequent VMRS instruction.

This instruction raises an Invalid Operation floating-point exception if either or both of the operands is a signaling NaN.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to 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



Half-precision scalar (size == 01)
(FEAT_FP16)

```
VCMP{<c>}{<q>}.F16 <Sd>, <Sm>
```

Single-precision scalar (size == 10)

```
VCMP{<c>}{<q>}.F32 <Sd>, <Sm>
```

Double-precision scalar (size == 11)

```
VCMP{<c>}{<q>}.F64 <Dd>, <Dm>
```

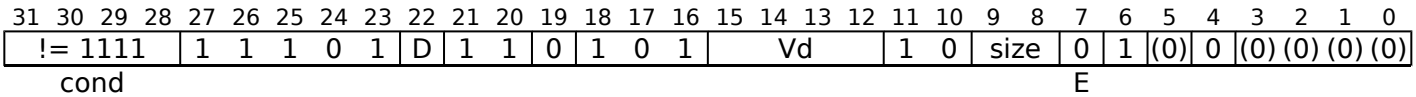
```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
quiet_nan_exc = (E == '1');  with_zero = FALSE;
case size of
    when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
    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



Half-precision scalar (size == 01)
(FEAT_FP16)

```
VCMP{<c>}{<q>}.F16 <Sd>, #0.0
```

Single-precision scalar (size == 10)

```
VCMP{<c>}{<q>}.F32 <Sd>, #0.0
```

Double-precision scalar (size == 11)

```
VCMP{<c>}{<q>}.F64 <Dd>, #0.0
```

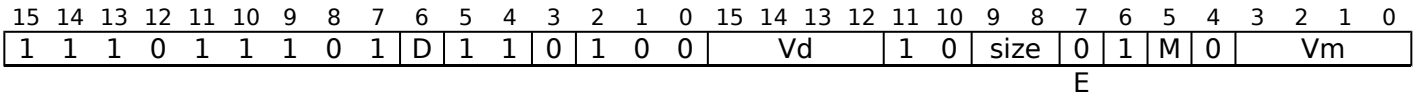
```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
quiet_nan_exc = (E == '1');  with_zero = TRUE;
case size of
    when '01' esize = 16; d = UInt(Vd:D);
    when '10' esize = 32; d = UInt(Vd:D);
    when '11' esize = 64; d = UInt(D:Vd);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1



Half-precision scalar (size == 01)
(FEAT_FP16)

```
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>
```

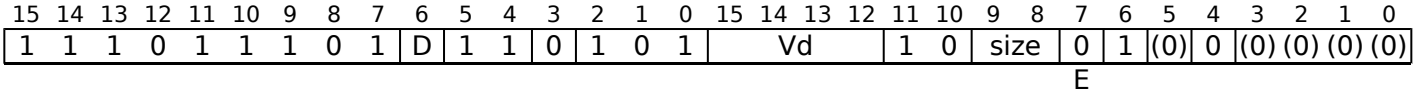
```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock()  then UNPREDICTABLE;
quiet_nan_exc = (E == '1');  with_zero = FALSE;
case size of
    when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
    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)
(FEAT_FP16)

```
VCMP{<c>}{<q>}.F16 <Sd>, #0.0
```

Single-precision scalar (size == 10)

```
VCMP{<c>}{<q>}.F32 <Sd>, #0.0
```

Double-precision scalar (size == 11)

```
VCMP{<c>}{<q>}.F64 <Dd>, #0.0
```

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock()  then UNPREDICTABLE;
quiet_nan_exc = (E == '1');  with_zero = TRUE;
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();  CheckVFPEnabled(TRUE);
    bits(4) nzcv;
    case esize of
        when 16
            bits(16) op16 = if with_zero then FPZero('0') else S[m]<15:0>;
            nzcv = FPCompare(S[d]<15:0>, op16, quiet_nan_exc, FPSCR[]);
        when 32
            bits(32) op32 = if with_zero then FPZero('0') else S[m];
            nzcv = FPCompare(S[d], op32, quiet_nan_exc, FPSCR[]);
        when 64
            bits(64) op64 = if with_zero then FPZero('0') else D[m];
            nzcv = FPCompare(D[d], op64, quiet_nan_exc, FPSCR[]);

    FPSCR<31:28> = nzcv; // FPSCR.<N,Z,C,V> set to nzcv
```

Operational information

The IEEE 754 standard specifies that the result of a comparison is precisely one of <, ==, > or unordered. If either or both of the operands is a NaN, they are unordered, and all three of (Operand1 < Operand2), (Operand1 == Operand2) and (Operand1 > Operand2) are false. An unordered comparison sets the *FPSCR* condition flags to N=0, Z=0, C=1, and V=1.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VCMPE

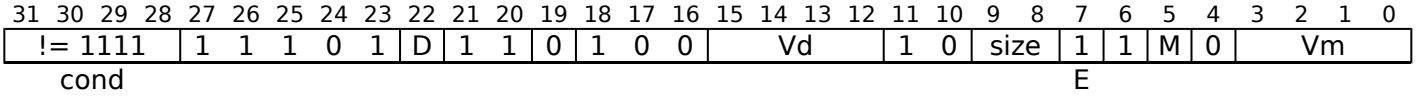
Vector Compare, raising Invalid Operation on NaN compares two floating-point registers, or one floating-point register and zero. It writes the result to the *FPSCR* flags. These are normally transferred to the *PSTATE*.{N, Z, C, V} Condition flags by a subsequent VMRS instruction.

This instruction raises an Invalid Operation floating-point exception if either or both of the operands is any type of NaN.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2) .

A1



Half-precision scalar (size == 01)

(FEAT_FP16)

```
VCMPE{<c>}{<q>}.F16 <Sd>, <Sm>
```

Single-precision scalar (size == 10)

```
VCMPE{<c>}{<q>}.F32 <Sd>, <Sm>
```

Double-precision scalar (size == 11)

```
VCMPE{<c>}{<q>}.F64 <Dd>, <Dm>
```

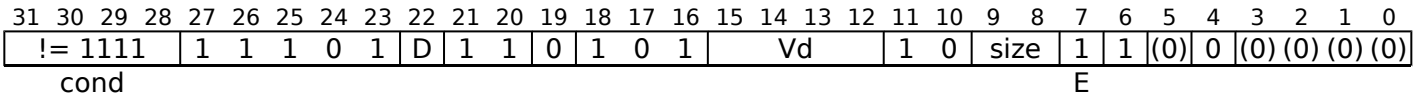
```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
quiet_nan_exc = (E == '1');  with_zero = FALSE;
case size of
    when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
    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



Half-precision scalar (size == 01)
(FEAT_FP16)

```
VCMPE{<c>}{<q>}.F16 <Sd>, #0.0
```

Single-precision scalar (size == 10)

```
VCMPE{<c>}{<q>}.F32 <Sd>, #0.0
```

Double-precision scalar (size == 11)

```
VCMPE{<c>}{<q>}.F64 <Dd>, #0.0
```

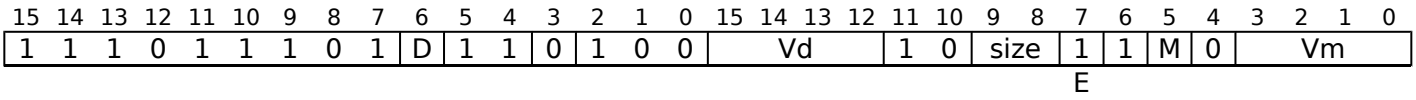
```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
quiet_nan_exc = (E == '1');  with_zero = TRUE;
case size of
    when '01' esize = 16; d = UInt(Vd:D);
    when '10' esize = 32; d = UInt(Vd:D);
    when '11' esize = 64; d = UInt(D:Vd);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1



Half-precision scalar (size == 01)
(FEAT_FP16)

```
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>
```

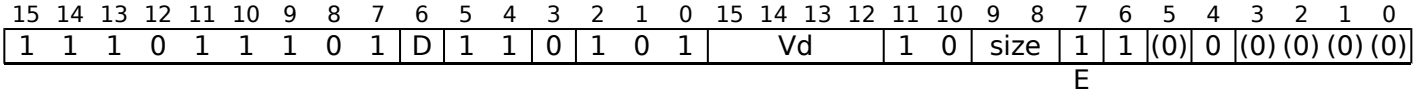
```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock()  then UNPREDICTABLE;
quiet_nan_exc = (E == '1');  with_zero = FALSE;
case size of
    when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
    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



Half-precision scalar (size == 01)
(FEAT_FP16)

```
VCMPE{<c>}{<q>}.F16 <Sd>, #0.0
```

Single-precision scalar (size == 10)

```
VCMPE{<c>}{<q>}.F32 <Sd>, #0.0
```

Double-precision scalar (size == 11)

```
VCMPE{<c>}{<q>}.F64 <Dd>, #0.0
```

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock()  then UNPREDICTABLE;
quiet_nan_exc = (E == '1');  with_zero = TRUE;
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();  CheckVFPEnabled(TRUE);
    bits(4) nzcv;
    case esize of
        when 16
            bits(16) op16 = if with_zero then FPZero('0') else S[m]<15:0>;
            nzcv = FPCompare(S[d]<15:0>, op16, quiet_nan_exc, FPSCR[]);
        when 32
            bits(32) op32 = if with_zero then FPZero('0') else S[m];
            nzcv = FPCompare(S[d], op32, quiet_nan_exc, FPSCR[]);
        when 64
            bits(64) op64 = if with_zero then FPZero('0') else D[m];
            nzcv = FPCompare(D[d], op64, quiet_nan_exc, FPSCR[]);

    FPSCR<31:28> = nzcv; // FPSCR.<N,Z,C,V> set to nzcv
```

Operational information

The IEEE 754 standard specifies that the result of a comparison is precisely one of <, ==, > or unordered. If either or both of the operands is a NaN, they are unordered, and all three of (Operand1 < Operand2), (Operand1 == Operand2) and (Operand1 > Operand2) are false. An unordered comparison sets the *FPSCR* condition flags to N=0, Z=0, C=1, and V=1.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VCNT

Vector Count Set Bits counts the number of bits that are one in each element in a vector, and places the results in a second vector.

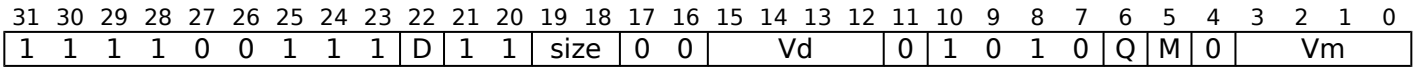
The operand vector elements must be 8-bit fields.

The result vector elements are 8-bit integers.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



64-bit SIMD vector (Q == 0)

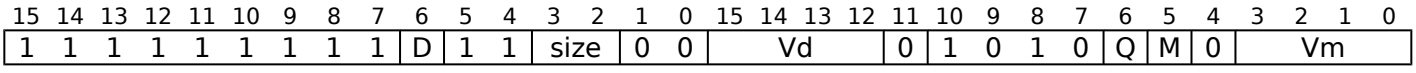
```
VCNT{<c>}{<q>}.8 <Dd>, <Dm> // (Encoded as Q = 0)
```

128-bit SIMD vector (Q == 1)

```
VCNT{<c>}{<q>}.8 <Qd>, <Qm> // (Encoded as Q = 1)
```

```
if size != '00' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8;  elements = 8;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1



64-bit SIMD vector (Q == 0)

```
VCNT{<c>}{<q>}.8 <Dd>, <Dm> // (Encoded as Q = 0)
```

128-bit SIMD vector (Q == 1)

```
VCNT{<c>}{<q>}.8 <Qd>, <Qm> // (Encoded as Q = 1)
```

```
if size != '00' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8;  elements = 8;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>	For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional. For encoding T1: see *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qm>	Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dm>	Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            Elem[D[d+r],e,esize] = BitCount(Elem[D[m+r],e,esize])<esize-1:0>;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VCVT (from single-precision to BFloat16, Advanced SIMD)

Vector Convert from single-precision to BFloat16 converts each 32-bit element in a vector from single-precision floating-point to BFloat16 format, and writes the result into a second vector. The result vector elements are half the width of the source vector elements.
Unlike the BFloat16 multiplication instructions, this instruction uses the Round to Nearest rounding mode, and can generate a floating-point exception that causes cumulative exception bits in the *FPSCR* to be set.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1
(FEAT_AA32BF16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	0	1	1	0	Vd				0	1	1	0	0	1	M	0	Vm			

A1

```
VCVT{<c>}{<q>}.BF16.F32 <Dd>, <Qm>

if !HaveAArch32BF16Ext() then UNDEFINED;
if Vm<0> == '1' then UNDEFINED;
integer d = UInt(D:Vd);
integer m = UInt(M:Vm);
```

T1
(FEAT_AA32BF16)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	0	1	1	0	Vd				0	1	1	0	0	1	M	0	Vm			

T1

```
VCVT{<c>}{<q>}.BF16.F32 <Dd>, <Qm>

if !HaveAArch32BF16Ext() then UNDEFINED;
if Vm<0> == '1' then UNDEFINED;
integer d = UInt(D:Vd);
integer m = UInt(M:Vm);
```

Assembler Symbols

<c>	For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional. For encoding T1: see *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Qm>	Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
bits(128) operand;
bits(64) result;

if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();

    operand = Q[m>>1];
    for e = 0 to 3
        bits(32) op = Elem[operand, e, 32];
        Elem[result, e, 16] = FPConvertBF(op, StandardFPSCRValue());
    D[d] = result;
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VCVT (between double-precision and single-precision)

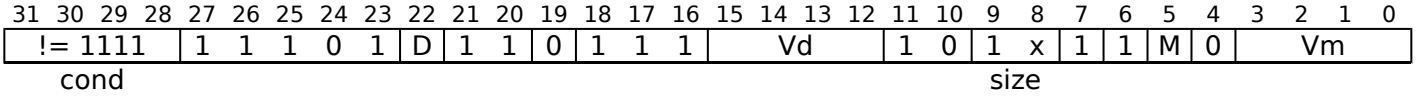
Convert between double-precision and single-precision does one of the following:

- Converts the value in a double-precision register to single-precision and writes the result to a single-precision register.
- Converts the value in a single-precision register to double-precision and writes the result to a double-precision register.

Depending on settings in the *CPACR*, *NSACR*, *HCPTR*, and *FPEXC* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



Single-precision to double-precision (size == 10)

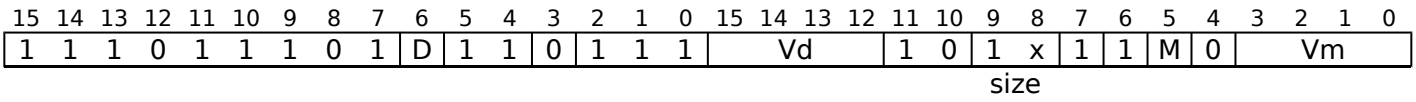
```
VCVT{<c>}{<q>}.F64.F32 <Dd>, <Sm>
```

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



Single-precision to double-precision (size == 10)

```
VCVT{<c>}{<q>}.F64.F32 <Dd>, <Sm>
```

Double-precision to single-precision (size == 11)

```
VCVT{<c>}{<q>}.F32.F64 <Sd>, <Dm>
```

```
double_to_single = (size == '11');
d = if double_to_single then UInt(Vd:D) else UInt(D:Vd);
m = if double_to_single then UInt(M:Vm) else UInt(Vm:M);
```

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
<Dm>	Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Sm>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    if double_to_single then
        S[d] = FPConvert(D[m], FPSCR[]);
    else
        D[d] = FPConvert(S[m], FPSCR[]);
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

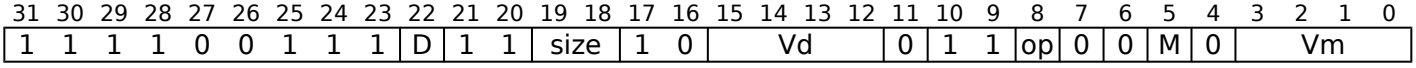
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VCVT (between half-precision and single-precision, Advanced SIMD)

Vector Convert between half-precision and single-precision converts each element in a vector from single-precision to half-precision floating-point, or from half-precision to single-precision, and places the results in a second vector.
Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



Half-precision to single-precision (op == 1)

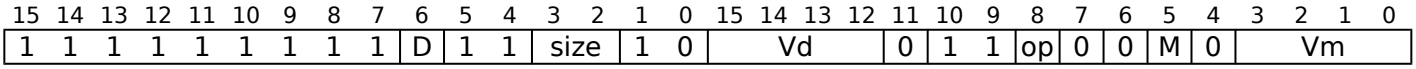
```
VCVT{<c>}{<q>}.F32.F16 <Qd>, <Dm> // (Encoded as op = 1)
```

Single-precision to half-precision (op == 0)

```
VCVT{<c>}{<q>}.F16.F32 <Dd>, <Qm> // (Encoded as op = 0)
```

```
if size != '01' then UNDEFINED;
half_to_single = (op == '1');
if half_to_single && Vd<0> == '1' then UNDEFINED;
if !half_to_single && Vm<0> == '1' then UNDEFINED;
esize = 16;  elements = 4;
m = UInt(M:Vm);  d = UInt(D:Vd);
```

T1



Half-precision to single-precision (op == 1)

```
VCVT{<c>}{<q>}.F32.F16 <Qd>, <Dm> // (Encoded as op = 1)
```

Single-precision to half-precision (op == 0)

```
VCVT{<c>}{<q>}.F16.F32 <Dd>, <Qm> // (Encoded as op = 0)
```

```
if size != '01' then UNDEFINED;
half_to_single = (op == '1');
if half_to_single && Vd<0> == '1' then UNDEFINED;
if !half_to_single && Vm<0> == '1' then UNDEFINED;
esize = 16;  elements = 4;
m = UInt(M:Vm);  d = UInt(D:Vd);
```

Assembler Symbols

<c>	For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional. For encoding T1: see *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Dm>	Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for e = 0 to elements-1
        if half_to_single then
            Elem[Q[d>>1],e,32] = FPConvert(Elem[Din[m],e,16], StandardFPSCRValue());
        else
            Elem[D[d],e,16] = FPConvert(Elem[Qin[m>>1],e,32], StandardFPSCRValue());
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VCVT (between floating-point and integer, Advanced SIMD)

Vector Convert between floating-point and integer converts each element in a vector from floating-point to integer, or from integer to floating-point, and places the results in a second vector.

The vector elements are the same type, and are floating-point numbers or integers. Signed and unsigned integers are distinct.

The floating-point to integer operation uses the Round towards Zero rounding mode. The integer to floating-point operation uses the Round to Nearest rounding mode.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size		1	1	Vd				0	1	1	op		Q	M	0	Vm			

64-bit SIMD vector (Q == 0)

```
VCVT{<c>}{<q>}.<dt1>.<dt2> <Dd>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VCVT{<c>}{<q>}.<dt1>.<dt2> <Qd>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
to_integer = (op<1> == '1');  unsigned = (op<0> == '1');
case size of
    when '01' esize = 16; elements = 4;
    when '10' esize = 32; elements = 2;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size		1	1	Vd				0	1	1	op		Q	M	0	Vm			

64-bit SIMD vector (Q == 0)

```
VCVT{<c>}{<q>}.<dt1>.<dt2> <Dd>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VCVT{<c>}{<q>}.<dt1>.<dt2> <Qd>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
if size == '01' && InITBlock()  then UNPREDICTABLE;
to_integer = (op<1> == '1');  unsigned = (op<0> == '1');
case size of
    when '01' esize = 16; elements = 4;
    when '10' esize = 32; elements = 2;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<dt1> Is the data type for the elements of the destination vector, encoded in “size:op”:

size	op	<dt1>
01	0x	F16
01	10	S16
01	11	U16
10	0x	F32
10	10	S32
10	11	U32

<dt2> Is the data type for the elements of the source vector, encoded in “size:op”:

size	op	<dt2>
01	00	S16
01	01	U16
01	1x	F16
10	00	S32
10	01	U32
10	1x	F32

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    bits(esize) result;
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[D[m+r],e,esize];
            if to_integer then
                result = FPToFixed(op1, 0, unsigned, StandardFPSCRValue(), FPRounding_ZERO);
            else
                result = FixedToFP(op1, 0, unsigned, StandardFPSCRValue(), FPRounding_TIEEVEN);
            Elem[D[d+r],e,esize] = result;
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VCVT (floating-point to integer, floating-point)

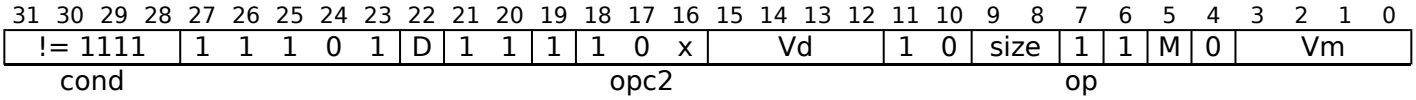
Convert floating-point to integer with Round towards Zero converts a value in a register from floating-point to a 32-bit integer, using the Round towards Zero rounding mode, and places the result in a second register.

VCVT (between floating-point and fixed-point, floating-point) describes conversions between floating-point and 16-bit integers.

Depending on settings in the *CPACR*, *NSACR*, *HCPTR*, and *FPEXC* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



Half-precision scalar (opc2 == 100 && size == 01)
(FEAT_FP16)

```
VCVT{<c>}{<q>}.U32.F16 <Sd>, <Sm>
```

Half-precision scalar (opc2 == 101 && size == 01)
(FEAT_FP16)

```
VCVT{<c>}{<q>}.S32.F16 <Sd>, <Sm>
```

Single-precision scalar (opc2 == 100 && size == 10)

```
VCVT{<c>}{<q>}.U32.F32 <Sd>, <Sm>
```

Single-precision scalar (opc2 == 101 && size == 10)

```
VCVT{<c>}{<q>}.S32.F32 <Sd>, <Sm>
```

Double-precision scalar (opc2 == 100 && size == 11)

```
VCVT{<c>}{<q>}.U32.F64 <Sd>, <Dm>
```

Double-precision scalar (opc2 == 101 && size == 11)

```
VCVT{<c>}{<q>}.S32.F64 <Sd>, <Dm>
```

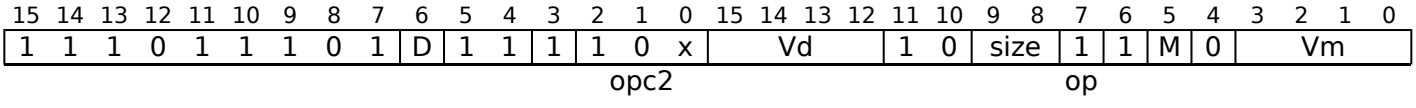
```
if opc2 != '000' && opc2 != '10x' then SEE "Related encodings";
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
to_integer = (opc2<2> == '1');
if to_integer then
    unsigned = (opc2<0> == '0');
    rounding = if op == '1' then FPRounding_ZERO else FPRoundingMode(FPSCR[]);
    d = UInt(Vd:D);
    case size of
        when '01' esize = 16; m = UInt(Vm:M);
        when '10' esize = 32; m = UInt(Vm:M);
        when '11' esize = 64; m = UInt(M:Vm);
else
    unsigned = (op == '0');
    rounding = FPRoundingMode(FPSCR[]);
    m = UInt(Vm:M);
    case size of
        when '01' esize = 16; d = UInt(Vd:D);
        when '10' esize = 32; d = UInt(Vd:D);
        when '11' esize = 64; d = UInt(D:Vd);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1



Half-precision scalar (opc2 == 100 && size == 01)
(FEAT_FP16)

```
VCVT{<c>}{<q>}.U32.F16 <Sd>, <Sm>
```

Half-precision scalar (opc2 == 101 && size == 01)
(FEAT_FP16)

```
VCVT{<c>}{<q>}.S32.F16 <Sd>, <Sm>
```

Single-precision scalar (opc2 == 100 && size == 10)

```
VCVT{<c>}{<q>}.U32.F32 <Sd>, <Sm>
```

Single-precision scalar (opc2 == 101 && size == 10)

```
VCVT{<c>}{<q>}.S32.F32 <Sd>, <Sm>
```

Double-precision scalar (opc2 == 100 && size == 11)

```
VCVT{<c>}{<q>}.U32.F64 <Sd>, <Dm>
```

Double-precision scalar (opc2 == 101 && size == 11)

```
VCVT{<c>}{<q>}.S32.F64 <Sd>, <Dm>
```

```
if opc2 != '000' && opc2 != '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[]);
    d = UInt(Vd:D);
    case size of
        when '01' esize = 16; m = UInt(Vm:M);
        when '10' esize = 32; m = UInt(Vm:M);
        when '11' esize = 64; m = UInt(M:Vm);
else
    unsigned = (op == '0');
    rounding = FPRoundingMode(FPSCR[]);
    m = UInt(Vm:M);
    case size of
        when '01' esize = 16; d = UInt(Vd:D);
        when '10' esize = 32; d = UInt(Vd:D);
        when '11' esize = 64; d = UInt(D:Vd);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Related encodings: See *Floating-point data-processing* for the T32 instruction set, or *Floating-point data-processing* for the A32 instruction set.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

<Sm> Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    if to_integer then
        case esize of
            when 16
                S[d] = FPToFixed(S[m]<15:0>, 0, unsigned, FPSCR[], rounding);
            when 32
                S[d] = FPToFixed(S[m], 0, unsigned, FPSCR[], rounding);
            when 64
                S[d] = FPToFixed(D[m], 0, unsigned, FPSCR[], rounding);
    else
        case esize of
            when 16
                bits(16) fp16 = FixedToFP(S[m], 0, unsigned, FPSCR[], rounding);
                S[d] = Zeros(16):fp16;
            when 32
                S[d] = FixedToFP(S[m], 0, unsigned, FPSCR[], rounding);
            when 64
                D[d] = FixedToFP(S[m], 0, unsigned, FPSCR[], rounding);
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VCVT (integer to floating-point, floating-point)

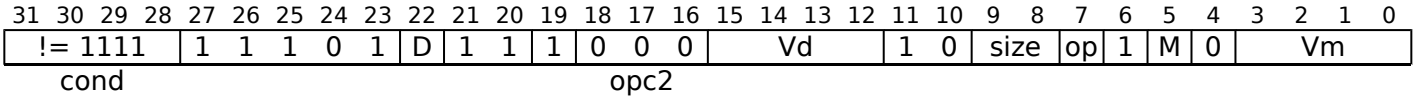
Convert integer to floating-point converts a 32-bit integer to floating-point using the rounding mode specified by the *FPSCR*, and places the result in a second register.

VCVT (between floating-point and fixed-point, floating-point) describes conversions between floating-point and 16-bit integers.

Depending on settings in the *CPACR*, *NSACR*, *HCPTR*, and *FPEXC* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



Half-precision scalar (size == 01)
(FEAT_FP16)

```
VCVT{<c>}{<q>}.F16.<dt> <Sd>, <Sm>
```

Single-precision scalar (size == 10)

```
VCVT{<c>}{<q>}.F32.<dt> <Sd>, <Sm>
```

Double-precision scalar (size == 11)

```
VCVT{<c>}{<q>}.F64.<dt> <Dd>, <Sm>
```

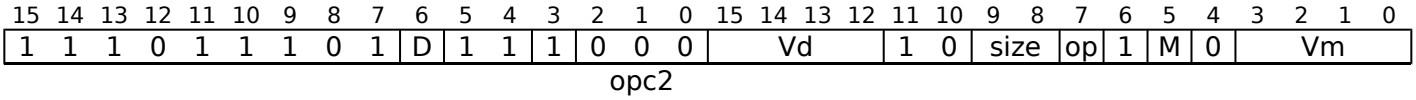
```
if opc2 != '000' && opc2 != '10x' then SEE "Related encodings";
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
to_integer = (opc2<2> == '1');
if to_integer then
    unsigned = (opc2<0> == '0');
    rounding = if op == '1' then FPRounding_ZERO else FPRoundingMode(FPSCR[]);
    d = UInt(Vd:D);
    case size of
        when '01' esize = 16; m = UInt(Vm:M);
        when '10' esize = 32; m = UInt(Vm:M);
        when '11' esize = 64; m = UInt(M:Vm);
else
    unsigned = (op == '0');
    rounding = FPRoundingMode(FPSCR[]);
    m = UInt(Vm:M);
    case size of
        when '01' esize = 16; d = UInt(Vd:D);
        when '10' esize = 32; d = UInt(Vd:D);
        when '11' esize = 64; d = UInt(D:Vd);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1



Half-precision scalar (size == 01)
(FEAT_FP16)

```
VCVT{<c>}{<q>}.F16.<dt> <Sd>, <Sm>
```

Single-precision scalar (size == 10)

```
VCVT{<c>}{<q>}.F32.<dt> <Sd>, <Sm>
```

Double-precision scalar (size == 11)

```
VCVT{<c>}{<q>}.F64.<dt> <Dd>, <Sm>
```

```
if opc2 != '000' && opc2 != '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[]);
    d = UInt(Vd:D);
    case size of
        when '01' esize = 16; m = UInt(Vm:M);
        when '10' esize = 32; m = UInt(Vm:M);
        when '11' esize = 64; m = UInt(M:Vm);
else
    unsigned = (op == '0');
    rounding = FPRoundingMode(FPSCR[]);
    m = UInt(Vm:M);
    case size of
        when '01' esize = 16; d = UInt(Vd:D);
        when '10' esize = 32; d = UInt(Vd:D);
        when '11' esize = 64; d = UInt(D:Vd);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Related encodings: See *Floating-point data-processing* for the T32 instruction set, or *Floating-point data-processing* for the A32 instruction set.

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<dt> Is the data type for the operand, encoded in “op”:

op	<dt>
0	U32
1	S32

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Sm> Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    if to_integer then
        case esize of
            when 16
                S[d] = FPToFixed(S[m]<15:0>, 0, unsigned, FPSCR[], rounding);
            when 32
                S[d] = FPToFixed(S[m], 0, unsigned, FPSCR[], rounding);
            when 64
                S[d] = FPToFixed(D[m], 0, unsigned, FPSCR[], rounding);
    else
        case esize of
            when 16
                bits(16) fp16 = FixedToFP(S[m], 0, unsigned, FPSCR[], rounding);
                S[d] = Zeros(16):fp16;
            when 32
                S[d] = FixedToFP(S[m], 0, unsigned, FPSCR[], rounding);
            when 64
                D[d] = FixedToFP(S[m], 0, unsigned, FPSCR[], rounding);
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VCVT (between floating-point and fixed-point, Advanced SIMD)

Vector Convert between floating-point and fixed-point converts each element in a vector from floating-point to fixed-point, or from fixed-point to floating-point, and places the results in a second vector.

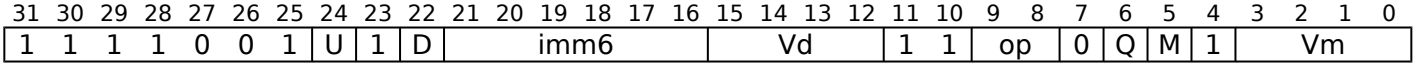
The vector elements are the same type, and are floating-point numbers or integers. Signed and unsigned integers are distinct.

The floating-point to fixed-point operation uses the Round towards Zero rounding mode. The fixed-point to floating-point operation uses the Round to Nearest rounding mode.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



64-bit SIMD vector (imm6 != 000xxx && Q == 0)

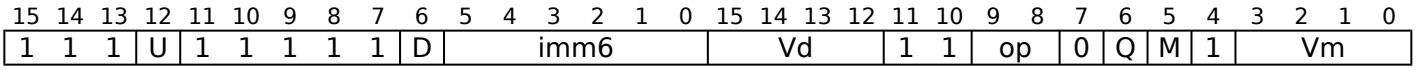
```
VCVT{<c>}{<q>}.<dt1>.<dt2> <Dd>, <Dm>, #<fbits>
```

128-bit SIMD vector (imm6 != 000xxx && Q == 1)

```
VCVT{<c>}{<q>}.<dt1>.<dt2> <Qd>, <Qm>, #<fbits>
```

```
if imm6 == '000xxx' then SEE "Related encodings";
if op<1> == '0' && !HaveFP16Ext() then UNDEFINED;
if op<1> == '0' && imm6 == '10xxxx' then UNDEFINED;
if imm6 == '0xxxxx' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
to_fixed = (op<0> == '1');  frac_bits = 64 - UInt(imm6);
unsigned = (U == '1');
case op<1> of
     when '0' esize = 16; elements = 4;
     when '1' esize = 32; elements = 2;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1



64-bit SIMD vector (imm6 != 000xxx && Q == 0)

```
VCVT{<c>}{<q>}.<dt1>.<dt2> <Dd>, <Dm>, #<fbits>
```

128-bit SIMD vector (imm6 != 000xxx && Q == 1)

```
VCVT{<c>}{<q>}.<dt1>.<dt2> <Qd>, <Qm>, #<fbits>
```

```
if imm6 == '000xxx' then SEE "Related encodings";
if op<1> == '0' && !HaveFP16Ext() then UNDEFINED;
if op<1> == '0' && imm6 == '10xxxx' then UNDEFINED;
if imm6 == '0xxxxx' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
to_fixed = (op<0> == '1');  frac_bits = 64 - UInt(imm6);
unsigned = (U == '1');
case op<1> of
     when '0' esize = 16; elements = 4;
     when '1' esize = 32; elements = 2;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Related encodings: See *Advanced SIMD one register and modified immediate* for the T32 instruction set, or *Advanced SIMD one register and modified immediate* for the A32 instruction set.

Assembler Symbols

<c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<dt1> Is the data type for the elements of the destination vector, encoded in “op:U”:

op	U	<dt1>
00	x	F16
01	0	S16
01	1	U16
10	x	F32
11	0	S32
11	1	U32

<dt2> Is the data type for the elements of the source vector, encoded in “op:U”:

op	U	<dt2>
00	0	S16
00	1	U16
01	x	F16
10	0	S32
10	1	U32
11	x	F32

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

<fbits> The number of fraction bits in the fixed point number, in the range 1 to 32 for 32-bit elements, or in the range 1 to 16 for 16-bit elements:

- (64 - <fbits>) is encoded in imm6.

An assembler can permit an <fbits> value of 0. This is encoded as floating-point to integer or integer to floating-point instruction, see *VCVT (between floating-point and integer, Advanced SIMD)*.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    bits(esize) result;
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[D[m+r],e,esize];
            if to_fixed then
                result = FPToFixed(op1, frac_bits, unsigned, StandardFPSCRValue(),
                                   FPRounding_ZERO);
            else
                result = FixedToFP(op1, frac_bits, unsigned, StandardFPSCRValue(),
                                   FPRounding_TIEEVEN);
            Elem[D[d+r],e,esize] = result;
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VCVT (between floating-point and fixed-point, floating-point)

Convert between floating-point and fixed-point converts a value in a register from floating-point to fixed-point, or from fixed-point to floating-point. Software can specify the fixed-point value as either signed or unsigned.

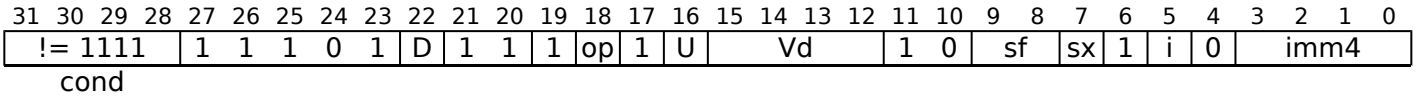
The fixed-point value can be 16-bit or 32-bit. Conversions from fixed-point values take their operand from the low-order bits of the source register and ignore any remaining bits. Signed conversions to fixed-point values sign-extend the result value to the destination register width. Unsigned conversions to fixed-point values zero-extend the result value to the destination register width.

The floating-point to fixed-point operation uses the Round towards Zero rounding mode. The fixed-point to floating-point operation uses the Round to Nearest rounding mode.

Depending on settings in the *CPACR*, *NSACR*, *HCPTR*, and *FPEXC* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



Half-precision scalar (op == 0 && sf == 01)
(FEAT_FP16)

```
VCVT{<c>}{<q>}.F16.<dt> <Sdm>, <Sdm>, #<fbits>
```

Half-precision scalar (op == 1 && sf == 01)
(FEAT_FP16)

```
VCVT{<c>}{<q>}.<dt>.F16 <Sdm>, <Sdm>, #<fbits>
```

Single-precision scalar (op == 0 && sf == 10)

```
VCVT{<c>}{<q>}.F32.<dt> <Sdm>, <Sdm>, #<fbits>
```

Single-precision scalar (op == 1 && sf == 10)

```
VCVT{<c>}{<q>}.<dt>.F32 <Sdm>, <Sdm>, #<fbits>
```

Double-precision scalar (op == 0 && sf == 11)

```
VCVT{<c>}{<q>}.F64.<dt> <Ddm>, <Ddm>, #<fbits>
```

Double-precision scalar (op == 1 && sf == 11)

```
VCVT{<c>}{<q>}.<dt>.F64 <Ddm>, <Ddm>, #<fbits>
```

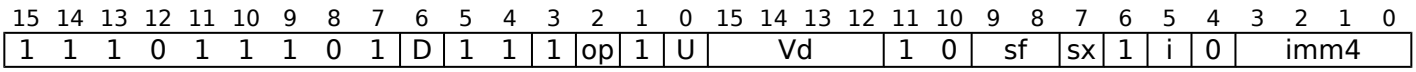
```
if sf == '00' || (sf == '01' && !HaveFP16Ext()) then UNDEFINED;
if sf == '01' && cond != '1110' then UNPREDICTABLE;
to_fixed = (op == '1');  unsigned = (U == '1');
size = if sx == '0' then 16 else 32;
frac_bits = size - UInt(imm4:i);
case sf of
    when '01' fp_size = 16; d = UInt(Vd:D);
    when '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



Half-precision scalar (op == 0 && sf == 01)
(FEAT_FP16)

```
VCVT{<c>}{<q>}.F16.<dt> <Sdm>, <Sdm>, #<fbits>
```

Half-precision scalar (op == 1 && sf == 01)
(FEAT_FP16)

```
VCVT{<c>}{<q>}.<dt>.F16 <Sdm>, <Sdm>, #<fbits>
```

Single-precision scalar (op == 0 && sf == 10)

```
VCVT{<c>}{<q>}.F32.<dt> <Sdm>, <Sdm>, #<fbits>
```

Single-precision scalar (op == 1 && sf == 10)

```
VCVT{<c>}{<q>}.<dt>.F32 <Sdm>, <Sdm>, #<fbits>
```

Double-precision scalar (op == 0 && sf == 11)

```
VCVT{<c>}{<q>}.F64.<dt> <Ddm>, <Ddm>, #<fbits>
```

Double-precision scalar (op == 1 && sf == 11)

```
VCVT{<c>}{<q>}.<dt>.F64 <Ddm>, <Ddm>, #<fbits>
```

```
if sf == '00' || (sf == '01' && !HaveFP16Ext()) then UNDEFINED;
if sf == '01' && InITBlock() then UNPREDICTABLE;
to_fixed = (op == '1');  unsigned = (U == '1');
size = if sx == '0' then 16 else 32;
frac_bits = size - UInt(imm4:i);
case sf of
    when '01' fp_size = 16; d = UInt(Vd:D);
    when '10' fp_size = 32; d = UInt(Vd:D);
    when '11' fp_size = 64; d = UInt(D:Vd);

if frac_bits < 0 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `frac_bits < 0`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *VCVT (between floating-point and fixed-point)*.

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<dt> Is the data type for the fixed-point number, encoded in “U:sx”:

U	sx	<dt>
0	0	S16
0	1	S32
1	0	U16
1	1	U32

<Sdm> Is the 32-bit name of the SIMD&FP destination and source register, encoded in the "Vd:D" field.

<Ddm> Is the 64-bit name of the SIMD&FP destination and source register, encoded in the "D:Vd" field.

<fbits> The number of fraction bits in the fixed-point number:

- If <dt> is S16 or U16, <fbits> must be in the range 0-16. (16 - <fbits>) is encoded in [imm4, i]
- If <dt> is S32 or U32, <fbits> must be in the range 1-32. (32 - <fbits>) is encoded in [imm4, i].

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    if to_fixed then
        bits(size) result;
        case fp_size of
            when 16
                result = FPToFixed(S[d]<15:0>, frac_bits, unsigned, FPSCR[], FPRounding_ZERO);
                S[d] = Extend(result, 32, unsigned);
            when 32
                result = FPToFixed(S[d], frac_bits, unsigned, FPSCR[], FPRounding_ZERO);
                S[d] = Extend(result, 32, unsigned);
            when 64
                result = FPToFixed(D[d], frac_bits, unsigned, FPSCR[], FPRounding_ZERO);
                D[d] = Extend(result, 64, unsigned);
    else
        case fp_size of
            when 16
                bits(16) fp16 = FixedToFP(S[d]<size-1:0>, frac_bits, unsigned, FPSCR[], FPRounding_TIEEVE
                S[d] = Zeros(16):fp16;
            when 32
                S[d] = FixedToFP(S[d]<size-1:0>, frac_bits, unsigned, FPSCR[], FPRounding_TIEEVEN);
            when 64
                D[d] = FixedToFP(D[d]<size-1:0>, frac_bits, unsigned, FPSCR[], FPRounding_TIEEVEN);
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VCVTA (Advanced SIMD)

Vector Convert floating-point to integer with Round to Nearest with Ties to Away converts each element in a vector from floating-point to integer using the Round to Nearest with Ties to Away rounding mode, and places the results in a second vector.

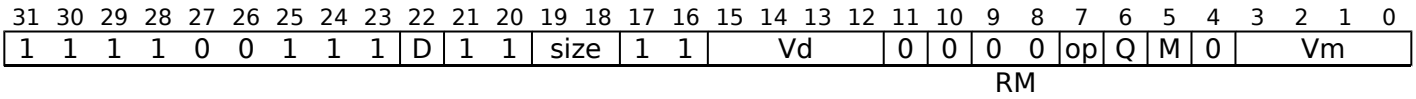
The operand vector elements are floating-point numbers.

The result vector elements are integers, and the same size as the operand vector elements. Signed and unsigned integers are distinct.

Depending on settings in the *CPACR*, *NSACR*, *HCPTR*, and *FPEXC* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



64-bit SIMD vector (Q == 0)

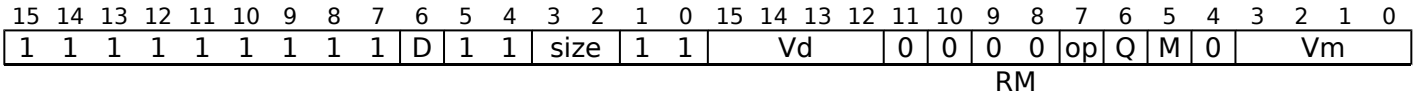
```
VCVTA{<q>}.<dt>.<dt2> <Dd>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VCVTA{<q>}.<dt>.<dt2> <Qd>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
rounding = FPDecodeRM(RM);  unsigned = (op == '1');
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



64-bit SIMD vector (Q == 0)

```
VCVTA{<q>}.<dt>.<dt2> <Dd>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VCVTA{<q>}.<dt>.<dt2> <Qd>, <Qm>
```

```
if InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
rounding = FPDecodeRM(RM);  unsigned = (op == '1');
case size of
    when '01' esize = 16; elements = 4;
    when '10' esize = 32; elements = 2;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<q> See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the destination, encoded in “op:size”:

op	size	<dt>
0	01	S16
0	10	S32
1	01	U16
1	10	U32

<dt2> Is the data type for the elements of the source vector, encoded in “size”:

size	<dt2>
01	F16
10	F32

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
EncodingSpecificOperations(); CheckAdvSIMDEnabled();
bits(esize) result;
for r = 0 to regs-1
    for e = 0 to elements-1
        Elem[D[d+r],e,esize] = FPToFixed(Elem[D[m+r],e,esize], 0, unsigned,
                                         StandardFPSCRValue(), rounding);
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

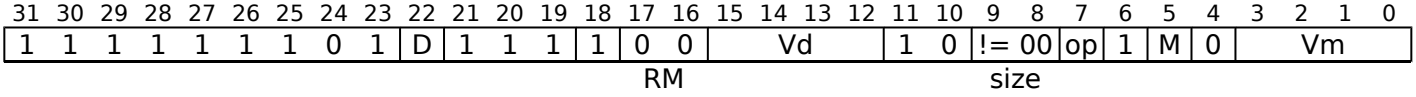
VCVTA (floating-point)

Convert floating-point to integer with Round to Nearest with Ties to Away converts a value in a register from floating-point to a 32-bit integer using the Round to Nearest with Ties to Away rounding mode, and places the result in a second register.

Depending on settings in the *CPACR*, *NSACR*, *HCPTR*, and *FPEXC* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



Half-precision scalar (size == 01)
(FEAT_FP16)

```
VCVTA{<q>}.<dt>.F16 <Sd>, <Sm>
```

Single-precision scalar (size == 10)

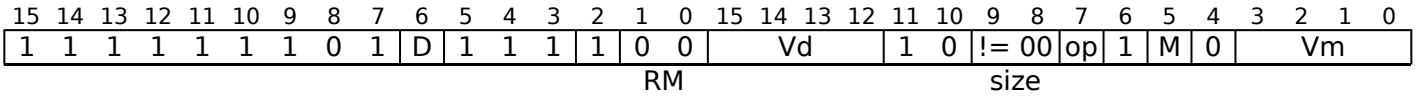
```
VCVTA{<q>}.<dt>.F32 <Sd>, <Sm>
```

Double-precision scalar (size == 11)

```
VCVTA{<q>}.<dt>.F64 <Sd>, <Dm>
```

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
rounding = FPDecodeRM(RM);  unsigned = (op == '0');
d = UInt(Vd:D);
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



Half-precision scalar (size == 01)
(FEAT_FP16)

```
VCVTA{<q>}.<dt>.F16 <Sd>, <Sm>
```

Single-precision scalar (size == 10)

```
VCVTA{<q>}.<dt>.F32 <Sd>, <Sm>
```

Double-precision scalar (size == 11)

```
VCVTA{<q>}.<dt>.F64 <Sd>, <Dm>
```

```
if InITBlock() then UNPREDICTABLE;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
rounding = FPDecodeRM(RM);  unsigned = (op == '0');
d = UInt(Vd:D);
case size of
    when '01' esize = 16; m = UInt(Vm:M);
    when '10' esize = 32; m = UInt(Vm:M);
    when '11' esize = 64; m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<q> See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the destination, encoded in “op”:

op	<dt>
0	U32
1	S32

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

<Sm> Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
case esize of
    when 16
        S[d] = FPToFixed(S[m]<15:0>, 0, unsigned, FPSCR[], rounding);
    when 32
        S[d] = FPToFixed(S[m], 0, unsigned, FPSCR[], rounding);
    when 64
        S[d] = FPToFixed(D[m], 0, unsigned, FPSCR[], rounding);
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VCVTB

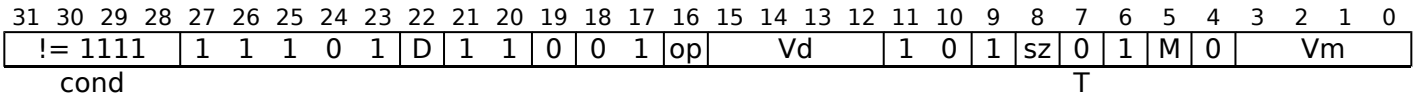
Convert to or from a half-precision value in the bottom half of a single-precision register does one of the following:

- Converts the half-precision value in the bottom half of a single-precision register to single-precision and writes the result to a single-precision register.
- Converts the half-precision value in the bottom half of a single-precision register to double-precision and writes the result to a double-precision register.
- Converts the single-precision value in a single-precision register to half-precision and writes the result into the bottom half of a single-precision register, preserving the other half of the destination register.
- Converts the double-precision value in a double-precision register to half-precision and writes the result into the bottom half of a single-precision register, preserving the other half of the destination register.

Depending on settings in the *CPACR*, *NSACR*, *HCPTR*, and *FPEXC* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



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)

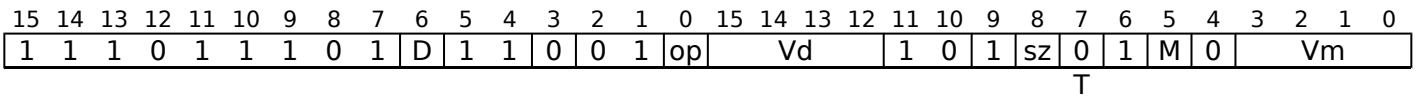
```
VCVTB{<c>}{<q>}.F16.F32 <Sd>, <Sm>
```

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



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(); CheckVFPEnabled(TRUE);
    bits(16) hp;
    if convert_from_half then
        hp = S[m]<lowbit+15:lowbit>;
        if uses_double then
            D[d] = FPConvert(hp, FPSCR[]);
        else
            S[d] = FPConvert(hp, FPSCR[]);
    else
        if uses_double then
            hp = FPConvert(D[m], FPSCR[]);
        else
            hp = FPConvert(S[m], FPSCR[]);
        S[d]<lowbit+15:lowbit> = hp;
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

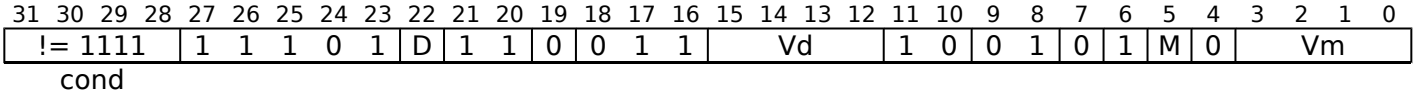
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VCVTB (BFloat16)

Converts the single-precision value in a single-precision register to BFloat16 format and writes the result into the bottom half of a single precision register, preserving the top 16 bits of the destination register.
Unlike the BFloat16 multiplication instructions, this instruction honors all the control bits in the *FPSCR* that apply to single-precision arithmetic, including the rounding mode. This instruction can generate a floating-point exception which causes a cumulative exception bit in the *FPSCR* to be set, or a synchronous exception to be taken, depending on the enable bits in the *FPSCR*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1
(FEAT_AA32BF16)

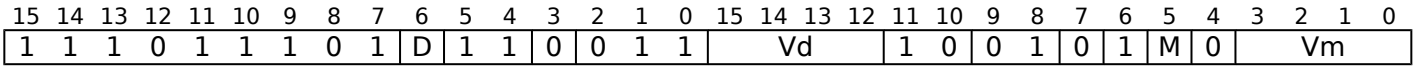


A1

```
VCVTB{<c>}{<q>}.BF16.F32 <Sd>, <Sm>
```

```
if !HaveAArch32BF16Ext() then UNDEFINED;
integer d = UInt(Vd:D);
integer m = UInt(Vm:M);
```

T1
(FEAT_AA32BF16)



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();
    CheckVFPEnabled(TRUE);

    S[d]<15:0> = FPConvertBF(S[m], FPSCR[]);
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VCVTM (Advanced SIMD)

Vector Convert floating-point to integer with Round towards -Infinity converts each element in a vector from floating-point to integer using the Round towards -Infinity rounding mode, and places the results in a second vector.

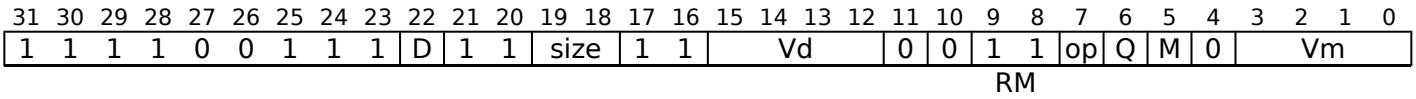
The operand vector elements are floating-point numbers.

The result vector elements are integers, and the same size as the operand vector elements. Signed and unsigned integers are distinct.

Depending on settings in the *CPACR*, *NSACR*, *HCPTR*, and *FPEXC* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



64-bit SIMD vector (Q == 0)

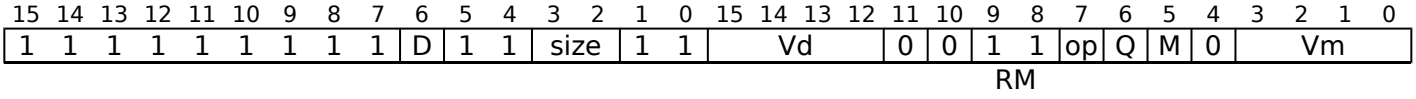
```
VCVTM{<q>}.<dt>.<dt2> <Dd>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VCVTM{<q>}.<dt>.<dt2> <Qd>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
rounding = FPDecodeRM(RM);  unsigned = (op == '1');
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



64-bit SIMD vector (Q == 0)

```
VCVTM{<q>}.<dt>.<dt2> <Dd>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VCVTM{<q>}.<dt>.<dt2> <Qd>, <Qm>
```

```
if InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
rounding = FPDecodeRM(RM);  unsigned = (op == '1');
case size of
    when '01' esize = 16; elements = 4;
    when '10' esize = 32; elements = 2;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<q> See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the destination, encoded in “op:size”:

op	size	<dt>
0	01	S16
0	10	S32
1	01	U16
1	10	U32

<dt2> Is the data type for the elements of the source vector, encoded in “size”:

size	<dt2>
01	F16
10	F32

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
EncodingSpecificOperations(); CheckAdvSIMDEnabled();
bits(esize) result;
for r = 0 to regs-1
    for e = 0 to elements-1
        Elem[D[d+r],e,esize] = FPToFixed(Elem[D[m+r],e,esize], 0, unsigned,
                                         StandardFPSCRValue(), rounding);
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

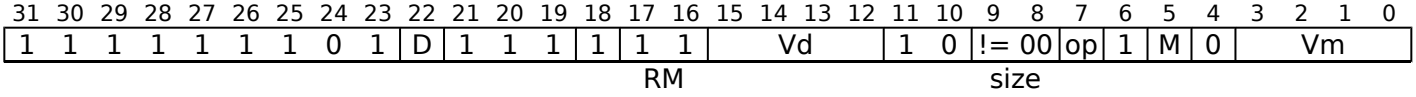
VCVTM (floating-point)

Convert floating-point to integer with Round towards -Infinity converts a value in a register from floating-point to a 32-bit integer using the Round towards -Infinity rounding mode, and places the result in a second register.

Depending on settings in the *CPACR*, *NSACR*, *HCPTR*, and *FPEXC* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



Half-precision scalar (size == 01)
(FEAT_FP16)

```
VCVTM{<q>}.<dt>.F16 <Sd>, <Sm>
```

Single-precision scalar (size == 10)

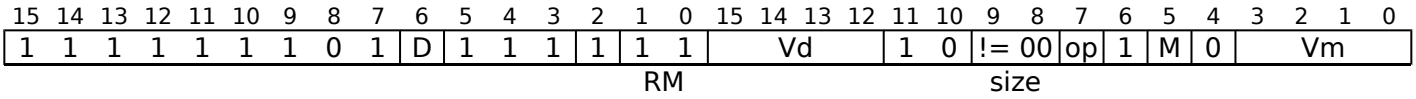
```
VCVTM{<q>}.<dt>.F32 <Sd>, <Sm>
```

Double-precision scalar (size == 11)

```
VCVTM{<q>}.<dt>.F64 <Sd>, <Dm>
```

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
rounding = FPDecodeRM(RM);  unsigned = (op == '0');
d = UInt(Vd:D);
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



Half-precision scalar (size == 01)
(FEAT_FP16)

```
VCVTM{<q>}.<dt>.F16 <Sd>, <Sm>
```

Single-precision scalar (size == 10)

```
VCVTM{<q>}.<dt>.F32 <Sd>, <Sm>
```

Double-precision scalar (size == 11)

```
VCVTM{<q>}.<dt>.F64 <Sd>, <Dm>
```

```
if InITBlock() then UNPREDICTABLE;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
rounding = FPDecodeRM(RM);  unsigned = (op == '0');
d = UInt(Vd:D);
case size of
    when '01' esize = 16; m = UInt(Vm:M);
    when '10' esize = 32; m = UInt(Vm:M);
    when '11' esize = 64; m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<q> See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the destination, encoded in “op”:

op	<dt>
0	U32
1	S32

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

<Sm> Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
case esize of
    when 16
        S[d] = FPToFixed(S[m]<15:0>, 0, unsigned, FPSCR[], rounding);
    when 32
        S[d] = FPToFixed(S[m], 0, unsigned, FPSCR[], rounding);
    when 64
        S[d] = FPToFixed(D[m], 0, unsigned, FPSCR[], rounding);
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VCVTN (Advanced SIMD)

Vector Convert floating-point to integer with Round to Nearest converts each element in a vector from floating-point to integer using the Round to Nearest rounding mode, and places the results in a second vector.

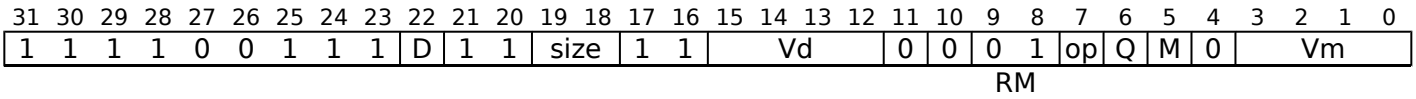
The operand vector elements are floating-point numbers.

The result vector elements are integers, and the same size as the operand vector elements. Signed and unsigned integers are distinct.

Depending on settings in the *CPACR*, *NSACR*, *HCPTR*, and *FPEXC* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



64-bit SIMD vector (Q == 0)

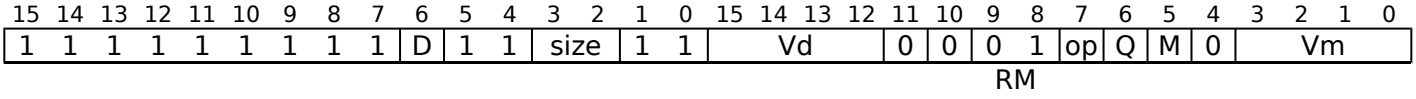
```
VCVTN{<q>}.<dt>.<dt2> <Dd>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VCVTN{<q>}.<dt>.<dt2> <Qd>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
rounding = FPDecodeRM(RM);  unsigned = (op == '1');
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



64-bit SIMD vector (Q == 0)

```
VCVTN{<q>}.<dt>.<dt2> <Dd>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VCVTN{<q>}.<dt>.<dt2> <Qd>, <Qm>
```

```
if InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
rounding = FPDecodeRM(RM);  unsigned = (op == '1');
case size of
    when '01' esize = 16; elements = 4;
    when '10' esize = 32; elements = 2;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<q> See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the destination, encoded in “op:size”:

op	size	<dt>
0	01	S16
0	10	S32
1	01	U16
1	10	U32

<dt2> Is the data type for the elements of the source vector, encoded in “size”:

size	<dt2>
01	F16
10	F32

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
EncodingSpecificOperations(); CheckAdvSIMDEnabled();
bits(esize) result;
for r = 0 to regs-1
    for e = 0 to elements-1
        Elem[D[d+r],e,esize] = FPToFixed(Elem[D[m+r],e,esize], 0, unsigned,
                                         StandardFPSCRValue(), rounding);
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

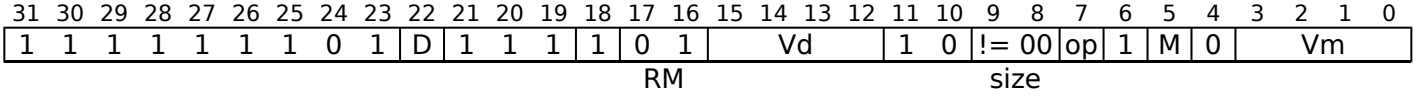
VCVTN (floating-point)

Convert floating-point to integer with Round to Nearest converts a value in a register from floating-point to a 32-bit integer using the Round to Nearest rounding mode, and places the result in a second register.

Depending on settings in the *CPACR*, *NSACR*, *HCPTR*, and *FPEXC* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



Half-precision scalar (size == 01)
(FEAT_FP16)

```
VCVTN{<q>}.<dt>.F16 <Sd>, <Sm>
```

Single-precision scalar (size == 10)

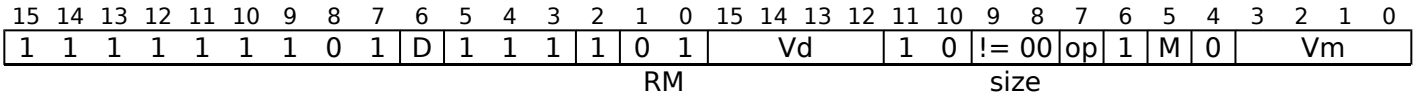
```
VCVTN{<q>}.<dt>.F32 <Sd>, <Sm>
```

Double-precision scalar (size == 11)

```
VCVTN{<q>}.<dt>.F64 <Sd>, <Dm>
```

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
rounding = FPDecodeRM(RM);  unsigned = (op == '0');
d = UInt(Vd:D);
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



Half-precision scalar (size == 01)
(FEAT_FP16)

```
VCVTN{<q>}.<dt>.F16 <Sd>, <Sm>
```

Single-precision scalar (size == 10)

```
VCVTN{<q>}.<dt>.F32 <Sd>, <Sm>
```

Double-precision scalar (size == 11)

```
VCVTN{<q>}.<dt>.F64 <Sd>, <Dm>
```

```
if InITBlock() then UNPREDICTABLE;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
rounding = FPDecodeRM(RM);  unsigned = (op == '0');
d = UInt(Vd:D);
case size of
    when '01' esize = 16; m = UInt(Vm:M);
    when '10' esize = 32; m = UInt(Vm:M);
    when '11' esize = 64; m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<q> See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the destination, encoded in “op”:

op	<dt>
0	U32
1	S32

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

<Sm> Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
case esize of
    when 16
        S[d] = FPToFixed(S[m]<15:0>, 0, unsigned, FPSCR[], rounding);
    when 32
        S[d] = FPToFixed(S[m], 0, unsigned, FPSCR[], rounding);
    when 64
        S[d] = FPToFixed(D[m], 0, unsigned, FPSCR[], rounding);
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VCVTP (Advanced SIMD)

Vector Convert floating-point to integer with Round towards +Infinity converts each element in a vector from floating-point to integer using the Round towards +Infinity rounding mode, and places the results in a second vector.

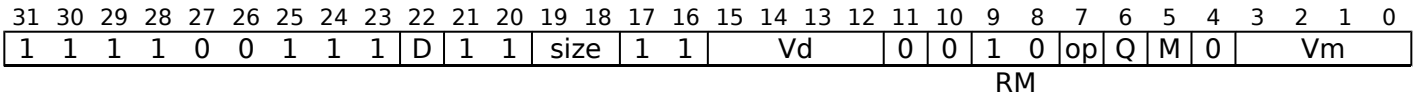
The operand vector elements are floating-point numbers.

The result vector elements are integers, and the same size as the operand vector elements. Signed and unsigned integers are distinct.

Depending on settings in the *CPACR*, *NSACR*, *HCPTR*, and *FPEXC* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



64-bit SIMD vector (Q == 0)

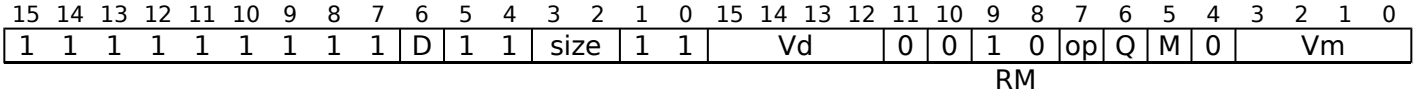
```
VCVTP{<q>}.<dt>.<dt2> <Dd>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VCVTP{<q>}.<dt>.<dt2> <Qd>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
rounding = FPDecodeRM(RM);  unsigned = (op == '1');
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



64-bit SIMD vector (Q == 0)

```
VCVTP{<q>}.<dt>.<dt2> <Dd>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VCVTP{<q>}.<dt>.<dt2> <Qd>, <Qm>
```

```
if InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
rounding = FPDecodeRM(RM);  unsigned = (op == '1');
case size of
    when '01' esize = 16; elements = 4;
    when '10' esize = 32; elements = 2;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<q> See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the destination, encoded in “op:size”:

op	size	<dt>
0	01	S16
0	10	S32
1	01	U16
1	10	U32

<dt2> Is the data type for the elements of the source vector, encoded in “size”:

size	<dt2>
01	F16
10	F32

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
EncodingSpecificOperations(); CheckAdvSIMDEnabled();
bits(esize) result;
for r = 0 to regs-1
    for e = 0 to elements-1
        Elem[D[d+r],e,esize] = FPToFixed(Elem[D[m+r],e,esize], 0, unsigned,
                                         StandardFPSCRValue(), rounding);
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

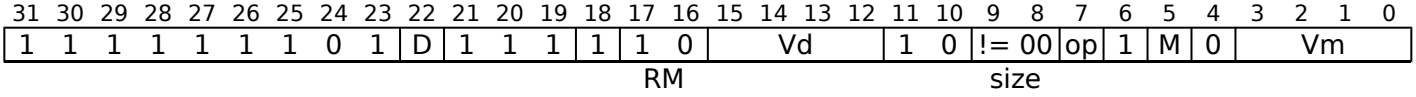
VCVTP (floating-point)

Convert floating-point to integer with Round towards +Infinity converts a value in a register from floating-point to a 32-bit integer using the Round towards +Infinity rounding mode, and places the result in a second register.

Depending on settings in the *CPACR*, *NSACR*, *HCPTR*, and *FPEXC* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



Half-precision scalar (size == 01)
(FEAT_FP16)

```
VCVTP{<q>}.<dt>.F16 <Sd>, <Sm>
```

Single-precision scalar (size == 10)

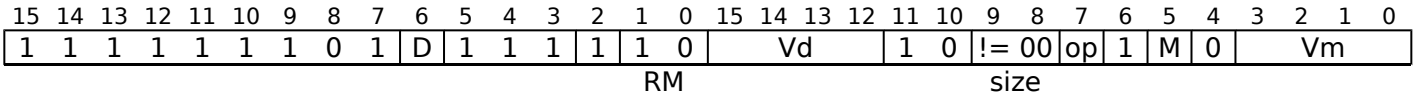
```
VCVTP{<q>}.<dt>.F32 <Sd>, <Sm>
```

Double-precision scalar (size == 11)

```
VCVTP{<q>}.<dt>.F64 <Sd>, <Dm>
```

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
rounding = FPDecodeRM(RM);  unsigned = (op == '0');
d = UInt(Vd:D);
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



Half-precision scalar (size == 01)
(FEAT_FP16)

```
VCVTP{<q>}.<dt>.F16 <Sd>, <Sm>
```

Single-precision scalar (size == 10)

```
VCVTP{<q>}.<dt>.F32 <Sd>, <Sm>
```

Double-precision scalar (size == 11)

```
VCVTP{<q>}.<dt>.F64 <Sd>, <Dm>
```

```
if InITBlock() then UNPREDICTABLE;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
rounding = FPDecodeRM(RM);  unsigned = (op == '0');
d = UInt(Vd:D);
case size of
    when '01' esize = 16; m = UInt(Vm:M);
    when '10' esize = 32; m = UInt(Vm:M);
    when '11' esize = 64; m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<q> See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the destination, encoded in “op”:

op	<dt>
0	U32
1	S32

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

<Sm> Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
case esize of
    when 16
        S[d] = FPToFixed(S[m]<15:0>, 0, unsigned, FPSCR[], rounding);
    when 32
        S[d] = FPToFixed(S[m], 0, unsigned, FPSCR[], rounding);
    when 64
        S[d] = FPToFixed(D[m], 0, unsigned, FPSCR[], rounding);
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VCVTR

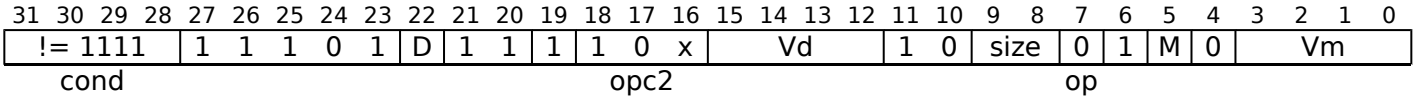
Convert floating-point to integer converts a value in a register from floating-point to a 32-bit integer, using the rounding mode specified by the *FPSCR* and places the result in a second register.

VCVT (between floating-point and fixed-point, floating-point) describes conversions between floating-point and 16-bit integers.

Depending on settings in the *CPACR*, *NSACR*, *HCPTR*, and *FPEXC* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



Half-precision scalar (opc2 == 100 && size == 01)
(FEAT_FP16)

```
VCVTR{<c>}{<q>}.U32.F16 <Sd>, <Sm>
```

Half-precision scalar (opc2 == 101 && size == 01)
(FEAT_FP16)

```
VCVTR{<c>}{<q>}.S32.F16 <Sd>, <Sm>
```

Single-precision scalar (opc2 == 100 && size == 10)

```
VCVTR{<c>}{<q>}.U32.F32 <Sd>, <Sm>
```

Single-precision scalar (opc2 == 101 && size == 10)

```
VCVTR{<c>}{<q>}.S32.F32 <Sd>, <Sm>
```

Double-precision scalar (opc2 == 100 && size == 11)

```
VCVTR{<c>}{<q>}.U32.F64 <Sd>, <Dm>
```

Double-precision scalar (opc2 == 101 && size == 11)

```
VCVTR{<c>}{<q>}.S32.F64 <Sd>, <Dm>
```

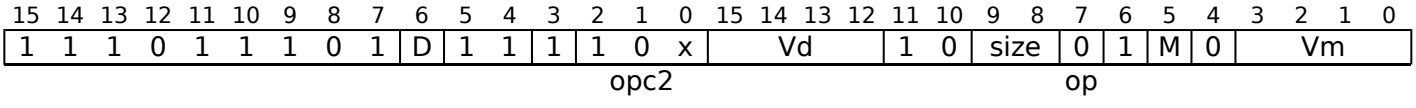
```
if opc2 != '000' && opc2 != '10x' then SEE "Related encodings";
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
to_integer = (opc2<2> == '1');
if to_integer then
    unsigned = (opc2<0> == '0');
    rounding = if op == '1' then FPRounding_ZERO else FPRoundingMode(FPSCR[]);
    d = UInt(Vd:D);
    case size of
        when '01' esize = 16; m = UInt(Vm:M);
        when '10' esize = 32; m = UInt(Vm:M);
        when '11' esize = 64; m = UInt(M:Vm);
else
    unsigned = (op == '0');
    rounding = FPRoundingMode(FPSCR[]);
    m = UInt(Vm:M);
    case size of
        when '01' esize = 16; d = UInt(Vd:D);
        when '10' esize = 32; d = UInt(Vd:D);
        when '11' esize = 64; d = UInt(D:Vd);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1



Half-precision scalar (opc2 == 100 && size == 01)
(FEAT_FP16)

```
VCVTR{<c>}{<q>}.U32.F16 <Sd>, <Sm>
```

Half-precision scalar (opc2 == 101 && size == 01)
(FEAT_FP16)

```
VCVTR{<c>}{<q>}.S32.F16 <Sd>, <Sm>
```

Single-precision scalar (opc2 == 100 && size == 10)

```
VCVTR{<c>}{<q>}.U32.F32 <Sd>, <Sm>
```

Single-precision scalar (opc2 == 101 && size == 10)

```
VCVTR{<c>}{<q>}.S32.F32 <Sd>, <Sm>
```

Double-precision scalar (opc2 == 100 && size == 11)

```
VCVTR{<c>}{<q>}.U32.F64 <Sd>, <Dm>
```

Double-precision scalar (opc2 == 101 && size == 11)

```
VCVTR{<c>}{<q>}.S32.F64 <Sd>, <Dm>
```

```
if opc2 != '000' && opc2 != '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[]);
    d = UInt(Vd:D);
    case size of
        when '01' esize = 16; m = UInt(Vm:M);
        when '10' esize = 32; m = UInt(Vm:M);
        when '11' esize = 64; m = UInt(M:Vm);
else
    unsigned = (op == '0');
    rounding = FPRoundingMode(FPSCR[]);
    m = UInt(Vm:M);
    case size of
        when '01' esize = 16; d = UInt(Vd:D);
        when '10' esize = 32; d = UInt(Vd:D);
        when '11' esize = 64; d = UInt(D:Vd);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Related encodings: See *Floating-point data-processing* for the T32 instruction set, or *Floating-point data-processing* for the A32 instruction set.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

<Sm> Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    if to_integer then
        case esize of
            when 16
                S[d] = FPToFixed(S[m]<15:0>, 0, unsigned, FPSCR[], rounding);
            when 32
                S[d] = FPToFixed(S[m], 0, unsigned, FPSCR[], rounding);
            when 64
                S[d] = FPToFixed(D[m], 0, unsigned, FPSCR[], rounding);
    else
        case esize of
            when 16
                bits(16) fp16 = FixedToFP(S[m], 0, unsigned, FPSCR[], rounding);
                S[d] = Zeros(16):fp16;
            when 32
                S[d] = FixedToFP(S[m], 0, unsigned, FPSCR[], rounding);
            when 64
                D[d] = FixedToFP(S[m], 0, unsigned, FPSCR[], rounding);
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VCVTT

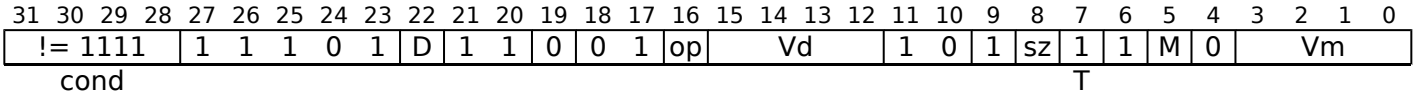
Convert to or from a half-precision value in the top half of a single-precision register does one of the following:

- Converts the half-precision value in the top half of a single-precision register to single-precision and writes the result to a single-precision register.
- Converts the half-precision value in the top half of a single-precision register to double-precision and writes the result to a double-precision register.
- Converts the single-precision value in a single-precision register to half-precision and writes the result into the top half of a single-precision register, preserving the other half of the destination register.
- Converts the double-precision value in a double-precision register to half-precision and writes the result into the top half of a single-precision register, preserving the other half of the destination register.

Depending on settings in the *CPACR*, *NSACR*, *HCPTR*, and *FPEXC* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



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)

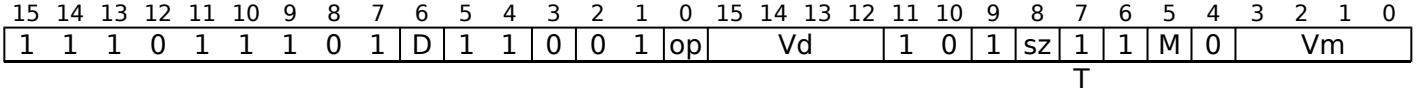
```
VCVTT{<c>}{<q>}.F16.F32 <Sd>, <Sm>
```

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



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(); CheckVFPEnabled(TRUE);
    bits(16) hp;
    if convert_from_half then
        hp = S[m]<lowbit+15:lowbit>;
        if uses_double then
            D[d] = FPConvert(hp, FPSCR[]);
        else
            S[d] = FPConvert(hp, FPSCR[]);
    else
        if uses_double then
            hp = FPConvert(D[m], FPSCR[]);
        else
            hp = FPConvert(S[m], FPSCR[]);
        S[d]<lowbit+15:lowbit> = hp;
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

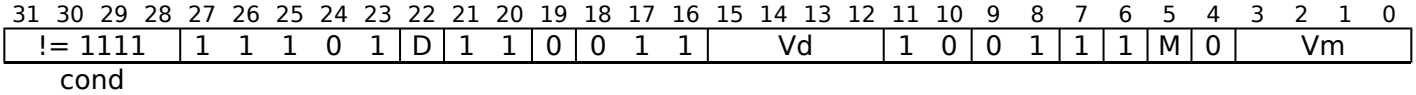
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VCVTT (BFloat16)

Converts the single-precision value in a single-precision register to BFloat16 format and writes the result in the top half of a single-precision register, preserving the bottom 16 bits of the register.
Unlike the BFloat16 multiplication instructions, this instruction honors all the control bits in the *FPSCR* that apply to single-precision arithmetic, including the rounding mode. This instruction can generate a floating-point exception which causes a cumulative exception bit in the *FPSCR* to be set, or a synchronous exception to be taken, depending on the enable bits in the *FPSCR*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1
(FEAT_AA32BF16)

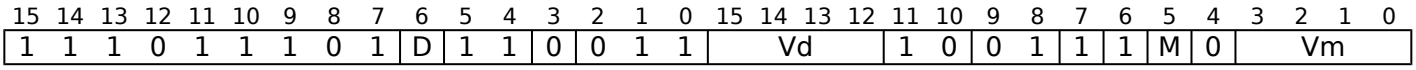


A1

```
VCVTT{<c>}{<q>}.BF16.F32 <Sd>, <Sm>
```

```
if !HaveAArch32BF16Ext() then UNDEFINED;
integer d = UInt(Vd:D);
integer m = UInt(Vm:M);
```

T1
(FEAT_AA32BF16)



T1

```
VCVTT{<c>}{<q>}.BF16.F32 <Sd>, <Sm>
```

```
if !HaveAArch32BF16Ext() then UNDEFINED;
integer d = UInt(Vd:D);
integer m = UInt(Vm:M);
```

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
<Sm>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEnabled(TRUE);

    S[d]<31:16> = FPConvertBF(S[m], FPSCR[]);
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

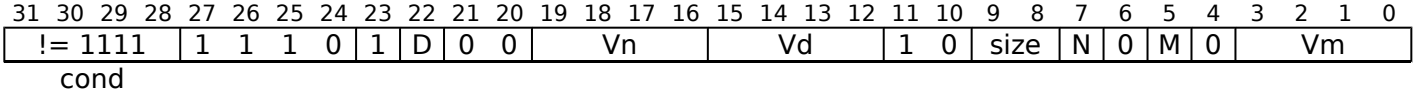
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VDIV

Divide divides one floating-point value by another floating-point value and writes the result to a third floating-point register.
Depending on settings in the *CPACR*, *NSACR*, *HCPTR*, and *FPEXC* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



Half-precision scalar (size == 01)
(FEAT_FP16)

```
VDIV{<c>}{<q>}.F16 {<Sd>,} <Sn>, <Sm>
```

Single-precision scalar (size == 10)

```
VDIV{<c>}{<q>}.F32 {<Sd>,} <Sn>, <Sm>
```

Double-precision scalar (size == 11)

```
VDIV{<c>}{<q>}.F64 {<Dd>,} <Dn>, <Dm>
```

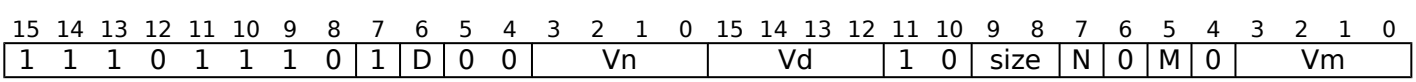
```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
case size of
    when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1



Half-precision scalar (size == 01)
(FEAT_FP16)

```
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_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

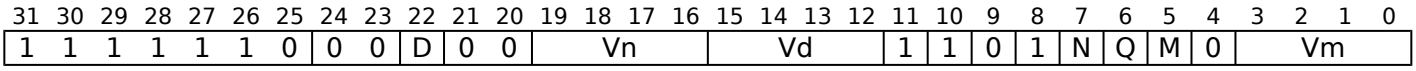
VDOT (vector)

BFloat16 floating-point (BF16) dot product (vector). This instruction delimits the source vectors into pairs of 16-bit BF16 elements. Within each pair, the elements in the first source vector are multiplied by the corresponding elements in the second source vector. The resulting single-precision products are then summed and added destructively to the single-precision element in the destination vector which aligns with the pair of BF16 values in the first source vector. The instruction does not update the *FPSCR* exception status.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

(FEAT_AA32BF16)



64-bit SIMD vector (Q == 0)

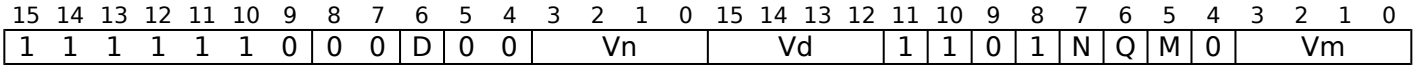
```
VDOT{<q>}.BF16 <Dd>, <Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VDOT{<q>}.BF16 <Qd>, <Qn>, <Qm>

if !HaveAArch32BF16Ext() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(M:Vm);
integer regs = if Q == '1' then 2 else 1;
```

T1

(FEAT_AA32BF16)



64-bit SIMD vector (Q == 0)

```
VDOT{<q>}.BF16 <Dd>, <Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VDOT{<q>}.BF16 <Qd>, <Qn>, <Qm>

if InITBlock() then UNPREDICTABLE;
if !HaveAArch32BF16Ext() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(M:Vm);
integer regs = if Q == '1' then 2 else 1;
```

Assembler Symbols

<q>	See *Standard assembler syntax fields*.
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qn>	Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
<Qm>	Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
bits(64) operand1;
bits(64) operand2;
bits(64) result;

CheckAdvSIMDEnabled();

for r = 0 to regs-1
    operand1 = Din[n+r];
    operand2 = Din[m+r];
    result = Din[d+r];
    for e = 0 to 1
        bits(16) elt1_a = Elem[operand1, 2 * e + 0, 16];
        bits(16) elt1_b = Elem[operand1, 2 * e + 1, 16];
        bits(16) elt2_a = Elem[operand2, 2 * e + 0, 16];
        bits(16) elt2_b = Elem[operand2, 2 * e + 1, 16];
        bits(32) sum = BFAdd(BFMul(elt1_a, elt2_a), BFMul(elt1_b, elt2_b));
        Elem[result, e, 32] = BFAdd(Elem[result, e, 32], sum);
    D[d+r] = result;
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

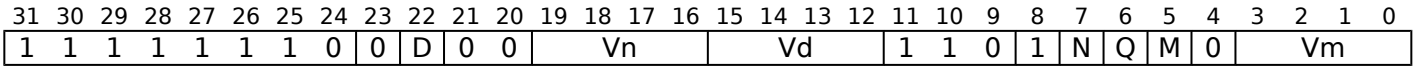
VDOT (by element)

BFloat16 floating-point indexed dot product (vector, by element). This instruction delimits the source vectors into pairs of 16-bit BF16 elements. Each pair of elements in the first source vector is multiplied by the indexed pair of elements in the second source vector. The resulting single-precision products are then summed and added destructively to the single-precision element in the destination vector which aligns with the pair of BFloat16 values in the first source vector. The instruction does not update the *FPSCR* exception status.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

(FEAT_AA32BF16)



64-bit SIMD vector (Q == 0)

```
VDOT{<q>}.BF16 <Dd>, <Dn>, <Dm>[<index>]
```

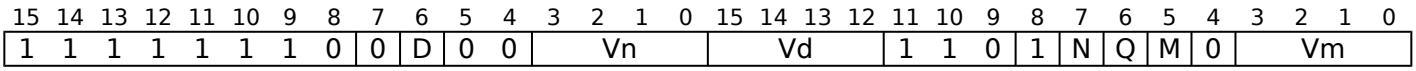
128-bit SIMD vector (Q == 1)

```
VDOT{<q>}.BF16 <Qd>, <Qn>, <Dm>[<index>]
```

```
if !HaveAArch32BF16Ext() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(Vm);
integer i = UInt(M);
integer regs = if Q == '1' then 2 else 1;
```

T1

(FEAT_AA32BF16)



64-bit SIMD vector (Q == 0)

```
VDOT{<q>}.BF16 <Dd>, <Dn>, <Dm>[<index>]
```

128-bit SIMD vector (Q == 1)

```
VDOT{<q>}.BF16 <Qd>, <Qn>, <Dm>[<index>]
```

```
if InITBlock() then UNPREDICTABLE;
if !HaveAArch32BF16Ext() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(Vm);
integer i = UInt(M);
integer regs = if Q == '1' then 2 else 1;
```

Assembler Symbols

<q>	See *Standard assembler syntax fields*.
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "Vm" field.

<index> Is the element index in the range 0 to 1, encoded in the "M" field.

Operation

```
bits(64) operand1;
bits(64) operand2;
bits(64) result;

CheckAdvSIMDEnabled();

operand2 = Din[m];
for r = 0 to regs-1
    operand1 = Din[n+r];
    result = Din[d+r];
    for e = 0 to 1
        bits(16) elt1_a = Elem[operand1, 2 * e + 0, 16];
        bits(16) elt1_b = Elem[operand1, 2 * e + 1, 16];
        bits(16) elt2_a = Elem[operand2, 2 * i + 0, 16];
        bits(16) elt2_b = Elem[operand2, 2 * i + 1, 16];
        bits(32) sum = BFAdd(BFMul(elt1_a, elt2_a), BFMul(elt1_b, elt2_b));
        Elem[result, e, 32] = BFAdd(Elem[result, e, 32], sum);
    D[d+r] = result;
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VDUP (general-purpose register)

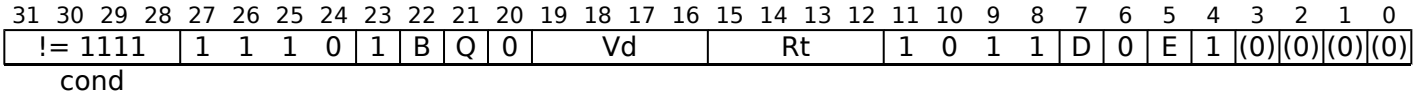
Duplicate general-purpose register to vector duplicates an element from a general-purpose register into every element of the destination vector.

The destination vector elements can be 8-bit, 16-bit, or 32-bit fields. The source element is the least significant 8, 16, or 32 bits of the general-purpose register. There is no distinction between data types.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

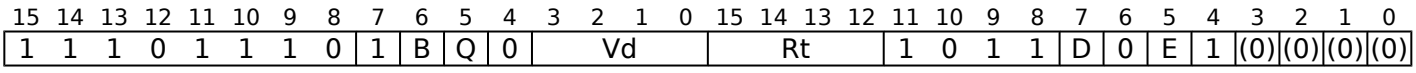


A1

```
VDUP{<c>}{<q>}.<size> <Qd>, <Rt> // (Encoded as Q = 1)

VDUP{<c>}{<q>}.<size> <Dd>, <Rt> // (Encoded as Q = 0)
```

```
if Q == '1' && Vd<0> == '1' then UNDEFINED;
d = UInt(D:Vd);  t = UInt(Rt);  regs = if Q == '0' then 1 else 2;
case B:E of
    when '00'  esize = 32;  elements = 2;
    when '01'  esize = 16;  elements = 4;
    when '10'  esize = 8;   elements = 8;
    when '11'  UNDEFINED;
if t == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

T1



T1

```
VDUP{<c>}{<q>}.<size> <Qd>, <Rt> // (Encoded as Q = 1)

VDUP{<c>}{<q>}.<size> <Dd>, <Rt> // (Encoded as Q = 0)
```

```
if Q == '1' && Vd<0> == '1' then UNDEFINED;
d = UInt(D:Vd);  t = UInt(Rt);  regs = if Q == '0' then 1 else 2;
case B:E of
    when '00'  esize = 32;  elements = 2;
    when '01'  esize = 16;  elements = 4;
    when '10'  esize = 8;   elements = 8;
    when '11'  UNDEFINED;
if t == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*. Arm strongly recommends that any VDUP instruction is unconditional, see *Conditional execution*.
<q>	See *Standard assembler syntax fields*.

<size> The data size for the elements of the destination vector. It must be one of:

8
Encoded as [b, e] = 0b10.

16
Encoded as [b, e] = 0b01.

32
Encoded as [b, e] = 0b00.

<Qd> The destination vector for a quadword operation.

<Dd> The destination vector for a doubleword operation.

<Rt> The Arm source register.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    scalar = R[t]<esize-1:0>;
    for r = 0 to regs-1
        for e = 0 to elements-1
            Elem[D[d+r],e,esize] = scalar;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VDUP (scalar)

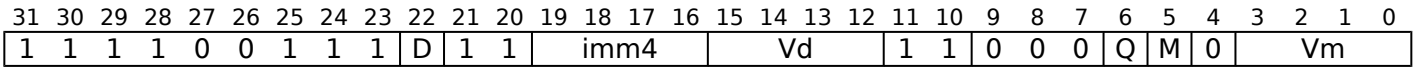
Duplicate vector element to vector duplicates a single element of a vector into every element of the destination vector. The scalar, and the destination vector elements, can be any one of 8-bit, 16-bit, or 32-bit fields. There is no distinction between data types.

For more information about scalars see *Advanced SIMD scalars*.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



(Q == 0)

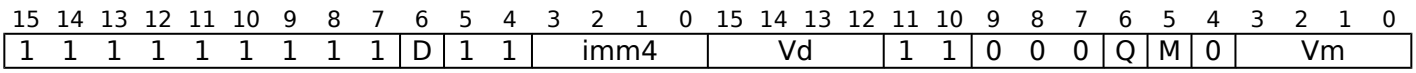
```
VDUP{<c>}{<q>}.<size> <Dd>, <Dm[x]>
```

(Q == 1)

```
VDUP{<c>}{<q>}.<size> <Qd>, <Dm[x]>
```

```
if imm4 == 'x000' then UNDEFINED;
if Q == '1' && Vd<0> == '1' then UNDEFINED;
case imm4 of
    when 'xxx1'  esize = 8;  elements = 8;  index = UInt(imm4<3:1>);
    when 'xx10'  esize = 16;  elements = 4;  index = UInt(imm4<3:2>);
    when 'x100'  esize = 32;  elements = 2;  index = UInt(imm4<3>);
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1



(Q == 0)

```
VDUP{<c>}{<q>}.<size> <Dd>, <Dm[x]>
```

(Q == 1)

```
VDUP{<c>}{<q>}.<size> <Qd>, <Dm[x]>
```

```
if imm4 == 'x000' then UNDEFINED;
if Q == '1' && Vd<0> == '1' then UNDEFINED;
case imm4 of
    when 'xxx1'  esize = 8;  elements = 8;  index = UInt(imm4<3:1>);
    when 'xx10'  esize = 16;  elements = 4;  index = UInt(imm4<3:2>);
    when 'x100'  esize = 32;  elements = 2;  index = UInt(imm4<3>);
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>	For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional. For encoding T1: see *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.

<size> The data size. It must be one of:

8
Encoded as imm4<0> = '1'. imm4<3:1> encodes the index[x] of the scalar.

16
Encoded as imm4<1:0> = '10'. imm4<3:2> encodes the index [x] of the scalar.

32
Encoded as imm4<2:0> = '100'. imm4<3> encodes the index [x] of the scalar.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm[x]> The scalar. For details of how [x] is encoded, see the description of <size>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    scalar = Elem[D[m],index,esize];
    for r = 0 to regs-1
        for e = 0 to elements-1
            Elem[D[d+r],e,esize] = scalar;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

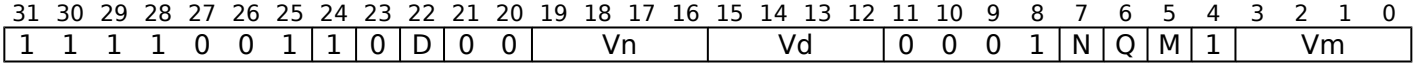
VEOR

Vector Bitwise Exclusive OR performs a bitwise Exclusive OR operation between two registers, and places the result in the destination register. The operand and result registers can be quadword or doubleword. They must all be the same size.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



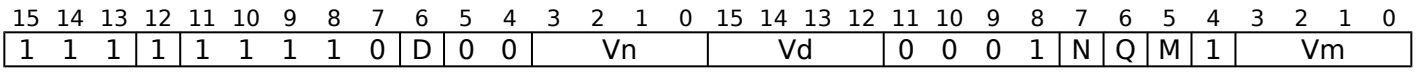
64-bit SIMD vector (Q == 0)

```
VEOR{<c>}{<q>}{.<dt>} {<Dd>,} <Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VEOR{<c>}{<q>}{.<dt>} {<Qd>,} <Qn>, <Qm>

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1



64-bit SIMD vector (Q == 0)

```
VEOR{<c>}{<q>}{.<dt>} {<Dd>,} <Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VEOR{<c>}{<q>}{.<dt>} {<Qd>,} <Qn>, <Qm>

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>	For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional. For encoding T1: see *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<dt>	An optional data type. It is ignored by assemblers, and does not affect the encoding.
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qn>	Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
<Qm>	Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        D[d+r] = D[n+r] EOR D[m+r];
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

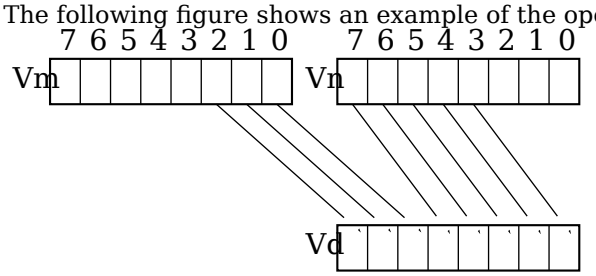
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VEXT (byte elements)

Vector Extract extracts elements from the bottom end of the second operand vector and the top end of the first, concatenates them and places the result in the destination vector.

The elements of the vectors are treated as being 8-bit fields. There is no distinction between data types.

The following figure shows an example of the operation of VEXT doubleword operation for imm = 3.



Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

This instruction is used by the pseudo-instruction VEXT (multibyte elements).

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	1	D	1	1	Vn				Vd				imm4				N	Q	M	0	Vm			

64-bit SIMD vector (Q == 0)

```
VEXT{<c>}{<q>}.8 {<Dd>,} <Dn>, <Dm>, #<imm>
```

128-bit SIMD vector (Q == 1)

```
VEXT{<c>}{<q>}.8 {<Qd>,} <Qn>, <Qm>, #<imm>
```

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if Q == '0' && imm4<3> == '1' then UNDEFINED;
quadword_operation = (Q == '1');  position = 8 * UInt(imm4);
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	1	1	D	1	1	Vn				Vd				imm4				N	Q	M	0	Vm			

64-bit SIMD vector (Q == 0)

```
VEXT{<c>}{<q>}.8 {<Dd>,} <Dn>, <Dm>, #<imm>
```

128-bit SIMD vector (Q == 1)

```
VEXT{<c>}{<q>}.8 {<Qd>,} <Qn>, <Qm>, #<imm>
```

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if Q == '0' && imm4<3> == '1' then UNDEFINED;
quadword_operation = (Q == '1');  position = 8 * UInt(imm4);
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

Assembler Symbols

<c>	For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional. For encoding T1: see *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qn>	Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
<Qm>	Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.
<imm>	For the 64-bit SIMD vector variant: is the location of the extracted result in the concatenation of the operands, as a number of bytes from the least significant end, in the range 0 to 7, encoded in the "imm4" field. For the 128-bit SIMD vector variant: is the location of the extracted result in the concatenation of the operands, as a number of bytes from the least significant end, in the range 0 to 15, encoded in the "imm4" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    if quadword_operation then
        Q[d>>1] = (Q[m>>1]:Q[n>>1])<position+127:position>;
    else
        D[d] = (D[m]:D[n])<position+63:position>;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VEXT (multibyte elements)

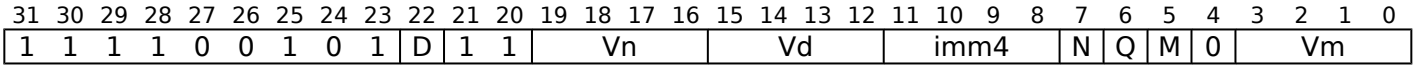
Vector Extract extracts elements from the bottom end of the second operand vector and the top end of the first, concatenates them and places the result in the destination vector

This is a pseudo-instruction of VEXT (byte elements). This means:

- The encodings in this description are named to match the encodings of VEXT (byte elements).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of VEXT (byte elements) gives the operational pseudocode for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



64-bit SIMD vector (Q == 0)

```
VEXT{<c>}{<q>}.<size> {<Dd>,} <Dn>, <Dm>, #<imm>
```

is equivalent to

```
VEXT{<c>}{<q>}.8 {<Dd>,} <Dn>, <Dm>, #<imm*(size/8)>
```

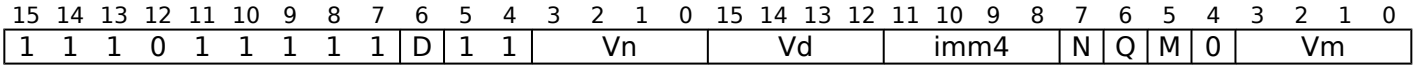
128-bit SIMD vector (Q == 1)

```
VEXT{<c>}{<q>}.<size> {<Qd>,} <Qn>, <Qm>, #<imm>
```

is equivalent to

```
VEXT{<c>}{<q>}.8 {<Qd>,} <Qn>, <Qm>, #<imm*(size/8)>
```

T1



64-bit SIMD vector (Q == 0)

```
VEXT{<c>}{<q>}.<size> {<Dd>,} <Dn>, <Dm>, #<imm>
```

is equivalent to

```
VEXT{<c>}{<q>}.8 {<Dd>,} <Dn>, <Dm>, #<imm*(size/8)>
```

128-bit SIMD vector (Q == 1)

```
VEXT{<c>}{<q>}.<size> {<Qd>,} <Qn>, <Qm>, #<imm>
```

is equivalent to

```
VEXT{<c>}{<q>}.8 {<Qd>,} <Qn>, <Qm>, #<imm*(size/8)>
```

Assembler Symbols

<c>	For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional. For encoding T1: see *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.

<size>	For the 64-bit SIMD vector variant: is the size of the operation, and can be one of 16 or 32. For the 128-bit SIMD vector variant: is the size of the operation, and can be one of 16, 32 or 64.
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qn>	Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
<Qm>	Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.
<imm>	For the 64-bit SIMD vector variant: is the location of the extracted result in the concatenation of the operands, as a number of bytes from the least significant end, in the range 0 to (128/<size>)-1. For the 128-bit SIMD vector variant: is the location of the extracted result in the concatenation of the operands, as a number of bytes from the least significant end, in the range 0 to (64/<size>)-1.

Operation

The description of VEXT (byte elements) gives the operational pseudocode for this instruction.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

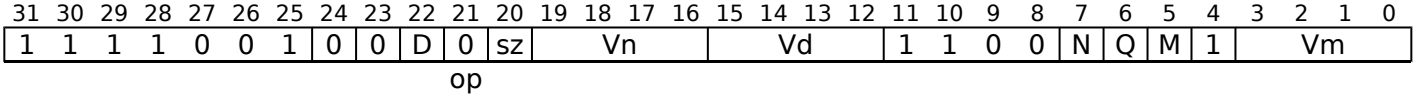
VFMA

Vector Fused Multiply Accumulate multiplies corresponding elements of two vectors, and accumulates the results into the elements of the destination vector. The instruction does not round the result of the multiply before the accumulation.

Depending on settings in the *CPACR*, *NSACR*, *HCPTR*, and *FPEXC* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2) .

A1



64-bit SIMD vector (Q == 0)

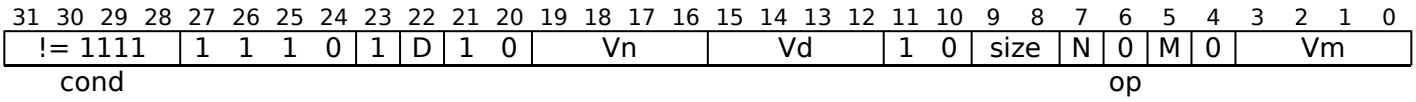
```
VFMA{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm>
```

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



Half-precision scalar (size == 01)
(FEAT_FP16)

```
VFMA{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>
```

Single-precision scalar (size == 10)

```
VFMA{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>
```

Double-precision scalar (size == 11)

```
VFMA{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>
```

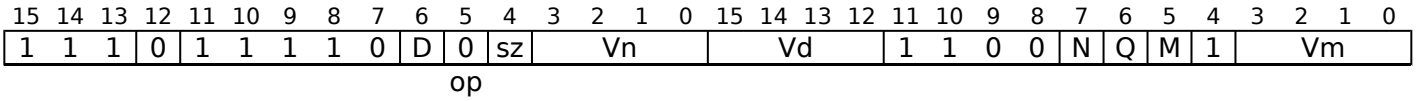
```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
advsimd = FALSE; op1_neg = (op == '1');
case size of
    when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1



64-bit SIMD vector (Q == 0)

```
VFMA{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VFMA{<c>}{<q>}.<dt> <Qd>, <Qn>, <Qm>
```

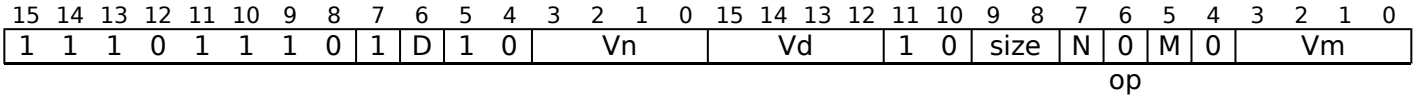
```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
advsimd = TRUE; op1_neg = (op == '1');
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
regs = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `sz == '1' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T2



Half-precision scalar (size == 01)
(FEAT_FP16)

```
VFMA{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>
```

Single-precision scalar (size == 10)

```
VFMA{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>
```

Double-precision scalar (size == 11)

```
VFMA{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>
```

```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock()  then UNPREDICTABLE;
advsimd = FALSE; op1_neg = (op == '1');
case size of
    when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.
For encoding A2, T1 and T2: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the vectors, encoded in “sz”:

sz	<dt>
0	F32
1	F16

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

<Sn> Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.

<Sm> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
    if advsimd then // Advanced SIMD instruction
        for r = 0 to regs-1
            for e = 0 to elements-1
                bits(esize) op1 = Elem[D[n+r],e,esize];
                if op1_neg then op1 = FPNeg(op1);
                Elem[D[d+r],e,esize] = FPMulAdd(Elem[D[d+r],e,esize],
                                        op1, Elem[D[m+r],e,esize], StandardFPSCRValue());

    else // VFP instruction
        case esize of
            when 16
                op16 = if op1_neg then FPNeg(S[n]<15:0>) else S[n]<15:0>;
                S[d] = Zeros(16) : FPMulAdd(S[d]<15:0>, op16, S[m]<15:0>, FPSCR[]);
            when 32
                op32 = if op1_neg then FPNeg(S[n]) else S[n];
                S[d] = FPMulAdd(S[d], op32, S[m], FPSCR[]);
            when 64
                op64 = if op1_neg then FPNeg(D[n]) else D[n];
                D[d] = FPMulAdd(D[d], op64, D[m], FPSCR[]);
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VFMAB, VFMAT (BFloat16, vector)

The Bfloat16 floating-point widening multiply-add long instruction widens the even-numbered (bottom) or odd-numbered (top) 16-bit elements in the first and second source vectors from Bfloat16 to single-precision format. The instruction then multiplies and adds these values to the overlapping single-precision elements of the destination vector.

Unlike other BFloat16 multiplication instructions, this performs a fused multiply-add, without intermediate rounding that uses the Round to Nearest rounding mode and can generate a floating-point exception that causes cumulative exception bits in the *FPSCR* to be set.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

(FEAT_AA32BF16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	0	0	0	D	1	1	Vn				Vd				1	0	0	0	N	Q	M	1	Vm			

A1

```
VFMA<bt>{<q>}.BF16 <Qd>, <Qn>, <Qm>

if !HaveAArch32BF16Ext() then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(M:Vm);
integer elements = 128 DIV 32;
integer sel = UInt(Q);
```

T1

(FEAT_AA32BF16)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	0	0	0	D	1	1	Vn				Vd				1	0	0	0	N	Q	M	1	Vm			

T1

```
VFMA<bt>{<q>}.BF16 <Qd>, <Qn>, <Qm>

if InITBlock() then UNPREDICTABLE;
if !HaveAArch32BF16Ext() then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(M:Vm);
integer elements = 128 DIV 32;
integer sel = UInt(Q);
```

Assembler Symbols

<bt> Is the bottom or top element specifier, encoded in “Q”:

Q	<bt>
0	B
1	T

<q> See *Standard assembler syntax fields*.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
CheckAdvSIMDEnabled();
bits(128) operand1 = Q[n>>1];
bits(128) operand2 = Q[m>>1];
bits(128) operand3 = Q[d>>1];
bits(128) result;

for e = 0 to elements-1
    bits(32) element1 = Elem[operand1, 2 * e + sel, 16] : Zeros(16);
    bits(32) element2 = Elem[operand2, 2 * e + sel, 16] : Zeros(16);
    bits(32) addend = Elem[operand3, e, 32];
    Elem[result, e, 32] = FPMulAdd(addend, element1, element2,
                                   StandardFPSCRValue());

Q[d>>1] = result;
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VFMAB, VFMAT (BFloat16, by scalar)

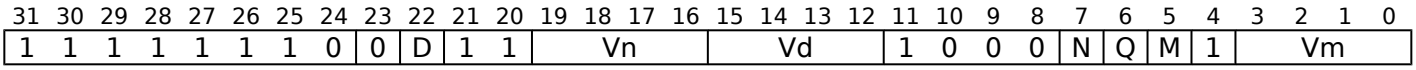
The BFloat16 floating-point widening multiply-add long instruction widens the even-numbered (bottom) or odd-numbered (top) 16-bit elements in the first source vector, and an indexed element in the second source vector from Bfloat16 to single-precision format. The instruction then multiplies and adds these values to the overlapping single-precision elements of the destination vector.

Unlike other BFloat16 multiplication instructions, this performs a fused multiply-add, without intermediate rounding that uses the Round to Nearest rounding mode and can generate a floating-point exception that causes cumulative exception bits in the *FPSCR* to be set.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

(FEAT_AA32BF16)



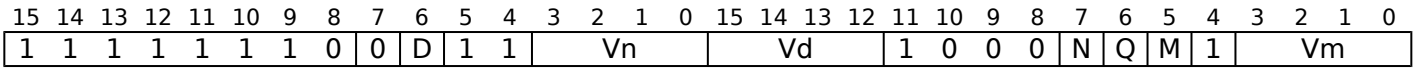
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0	0	D	1	1	Vn				Vd				1	0	0	0	N	Q	M	1	Vm			

A1

```
VFMA<bt>{<q>}.BF16 <Qd>, <Qn>, <Dm>[<index>]

if !HaveAArch32BF16Ext() then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' then UNDEFINED;
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(Vm<2:0>);
integer i = UInt(M:Vm<3>);
integer elements = 128 DIV 32;
integer sel = UInt(Q);
```

T1

(FEAT_AA32BF16)



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0	0	D	1	1	Vn				Vd				1	0	0	0	N	Q	M	1	Vm			

T1

```
VFMA<bt>{<q>}.BF16 <Qd>, <Qn>, <Dm>[<index>]

if InITBlock() then UNPREDICTABLE;
if !HaveAArch32BF16Ext() then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' then UNDEFINED;
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(Vm<2:0>);
integer i = UInt(M:Vm<3>);
integer elements = 128 DIV 32;
integer sel = UInt(Q);
```

Assembler Symbols

<bt> Is the bottom or top element specifier, encoded in “Q”:

Q	<bt>
0	B
1	T

<q> See *Standard assembler syntax fields*.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "Vm<2:0>" field.

<index> Is the element index in the range 0 to 3, encoded in the "M:Vm<3>" field.

Operation

```
CheckAdvSIMDEnabled();
bits(128) operand1 = Q[n>>1];
bits(64) operand2 = D[m];
bits(128) operand3 = Q[d>>1];
bits(128) result;

bits(32) element2 = Elem[operand2, i, 16] : Zeros(16);

for e = 0 to elements-1
    bits(32) element1 = Elem[operand1, 2 * e + sel, 16] : Zeros(16);
    bits(32) addend = Elem[operand3, e, 32];
    Elem[result, e, 32] = FPMulAdd(addend, element1, element2,
                                   StandardFPSCRValue());

Q[d>>1] = result;
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VFMAL (vector)

Vector Floating-point Multiply-Add Long to accumulator (vector). This instruction multiplies corresponding values in the vectors in the two source SIMD&FP registers, and accumulates the product to the corresponding vector element of the destination SIMD&FP register. The instruction does not round the result of the multiply before the accumulation.

Depending on settings in the *CPACR*, *NSACR*, *HCPTR*, and *FPEXC* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

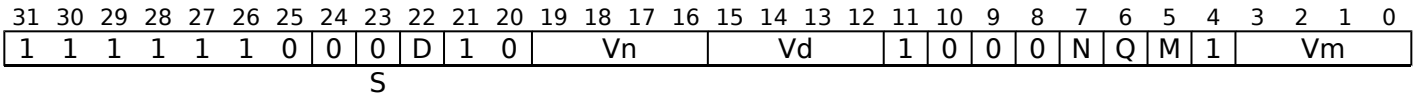
In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

ID_ISAR6.FHM indicates whether this instruction is supported.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

(FEAT_FHM)



64-bit SIMD vector (Q == 0)

```
VFMAL{<q>}.F16 <Dd>, <Sn>, <Sm>
```

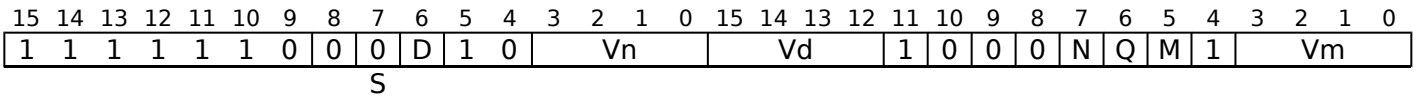
128-bit SIMD vector (Q == 1)

```
VFMAL{<q>}.F16 <Qd>, <Dn>, <Dm>
```

```
if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
if Q == '1' && Vd<0> == '1' then UNDEFINED;

integer d = UInt(D:Vd);
integer n = if Q == '1' then UInt(N:Vn) else UInt(Vn:N);
integer m = if Q == '1' then UInt(M:Vm) else UInt(Vm:M);
integer esize = 32;
integer regs = if Q=='1' then 2 else 1;
integer datasize = if Q=='1' then 64 else 32;
boolean sub_op = S=='1';
```

T1

(FEAT_FHM)



64-bit SIMD vector (Q == 0)

```
VFMAL{<q>}.F16 <Dd>, <Sn>, <Sm>
```

128-bit SIMD vector (Q == 1)

```
VFMAL{<q>}.F16 <Qd>, <Dn>, <Dm>
```

```
if InITBlock() then UNPREDICTABLE;
if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
if Q == '1' && Vd<0> == '1' then UNDEFINED;

integer d = UInt(D:Vd);
integer n = if Q == '1' then UInt(N:Vn) else UInt(Vn:N);
integer m = if Q == '1' then UInt(M:Vm) else UInt(Vm:M);
integer esize = 32;
integer regs = if Q=='1' then 2 else 1;
integer datasize = if Q=='1' then 64 else 32;
boolean sub_op = S=='1';
```

Assembler Symbols

<q>	See *Standard assembler syntax fields*.
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Sn>	Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.
<Sm>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.

Operation

```
CheckAdvSIMDEnabled();
bits(datasize) operand1 ;
bits(datasize) operand2 ;
bits(64) operand3;
bits(64) result;
bits(esize DIV 2) element1;
bits(esize DIV 2) element2;

if Q=='0' then
    operand1 = S[n]<datasize-1:0>;
    operand2 = S[m]<datasize-1:0>;
else
    operand1 = D[n]<datasize-1:0>;
    operand2 = D[m]<datasize-1:0>;
for r = 0 to regs-1
    operand3 = D[d+r];
    for e = 0 to 1
        element1 = Elem[operand1, 2*r+e, esize DIV 2];
        element2 = Elem[operand2, 2*r+e, esize DIV 2];
        if sub_op then element1 = FPNeg(element1);
        Elem[result, e, esize] = FPMulAddH(Elem[operand3, e, esize], element1, element2, StandardFPSCRVal
    D[d+r] = result;
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VFMAL (by scalar)

Vector Floating-point Multiply-Add Long to accumulator (by scalar). This instruction multiplies the vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, and accumulates the product to the corresponding vector element of the destination SIMD&FP register. The instruction does not round the result of the multiply before the accumulation.

Depending on settings in the *CPACR*, *NSACR*, *HCPTR*, and *FPEXC* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

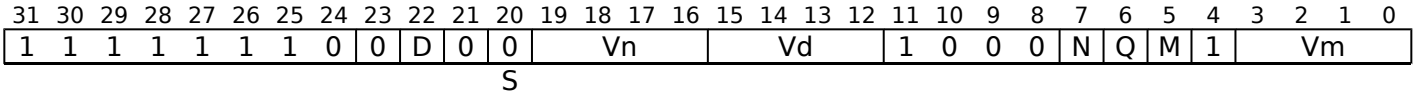
In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

ID_ISAR6.FHM indicates whether this instruction is supported.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

(FEAT_FHM)



64-bit SIMD vector (Q == 0)

```
VFMAL{<q>}.F16 <Dd>, <Sn>, <Sm>[<index>]
```

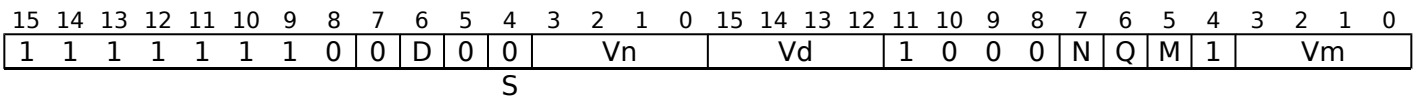
128-bit SIMD vector (Q == 1)

```
VFMAL{<q>}.F16 <Qd>, <Dn>, <Dm>[<index>]
```

```
if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
if Q == '1' && Vd<0> == '1' then UNDEFINED;

integer d = UInt(D:Vd);
integer n = if Q == '1' then UInt(N:Vn) else UInt(Vn:N);
integer m = if Q == '1' then UInt(Vm<2:0>) else UInt(Vm<2:0>:M);

integer index = if Q == '1' then UInt(M:Vm<3>) else UInt(Vm<3>);
integer esize = 32;
integer regs = if Q=='1' then 2 else 1;
integer datasize = if Q=='1' then 64 else 32;
boolean sub_op = S=='1';
```

T1

(FEAT_FHM)



64-bit SIMD vector (Q == 0)

```
VFMAL{<q>}.F16 <Dd>, <Sn>, <Sm>[<index>]
```

128-bit SIMD vector (Q == 1)

```
VFMAL{<q>}.F16 <Qd>, <Dn>, <Dm>[<index>]
```

```
if InITBlock() then UNPREDICTABLE;
if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
if Q == '1' && Vd<0> == '1' then UNDEFINED;

integer d = UInt(D:Vd);
integer n = if Q == '1' then UInt(N:Vn) else UInt(Vn:N);
integer m = if Q == '1' then UInt(Vm<2:0>) else UInt(Vm<2:0>:M);

integer index = if Q == '1' then UInt(M:Vm<3>) else UInt(Vm<3>);
integer esize = 32;
integer regs = if Q=='1' then 2 else 1;
integer datasize = if Q=='1' then 64 else 32;
boolean sub_op = S=='1';
```

Assembler Symbols

<q>	See *Standard assembler syntax fields*.
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "Vm<2:0>" field.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Sn>	Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.
<Sm>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm<2:0>:M" field.
<index>	For the 64-bit SIMD vector variant: is the element index in the range 0 to 1, encoded in the "Vm<3>" field. For the 128-bit SIMD vector variant: is the element index in the range 0 to 3, encoded in the "M:Vm<3>" field.

Operation

```
CheckAdvSIMDEnabled();
bits(datasize) operand1 ;
bits(datasize) operand2 ;
bits(64) operand3;
bits(64) result;
bits(esize DIV 2) element1;
bits(esize DIV 2) element2;

if Q=='0' then
    operand1 = S[n]<datasize-1:0>;
    operand2 = S[m]<datasize-1:0>;
else
    operand1 = D[n]<datasize-1:0>;
    operand2 = D[m]<datasize-1:0>;
element2 = Elem[operand2, index, esize DIV 2];
for r = 0 to regs-1
    operand3 = D[d+r];
    for e = 0 to 1
        element1 = Elem[operand1, 2*r+e, esize DIV 2];
        if sub_op then element1 = FPNeg(element1);
        Elem[result, e, esize] = FPMulAddH(Elem[operand3, e, esize], element1, element2, StandardFPSCRVal
    D[d+r] = result;
```

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

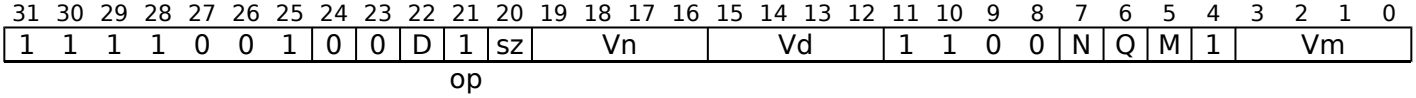
VFMS

Vector Fused Multiply Subtract negates the elements of one vector and multiplies them with the corresponding elements of another vector, adds the products to the corresponding elements of the destination vector, and places the results in the destination vector. The instruction does not round the result of the multiply before the addition.

Depending on settings in the *CPACR*, *NSACR*, *HCPTR*, and *FPEXC* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2) .

A1



64-bit SIMD vector (Q == 0)

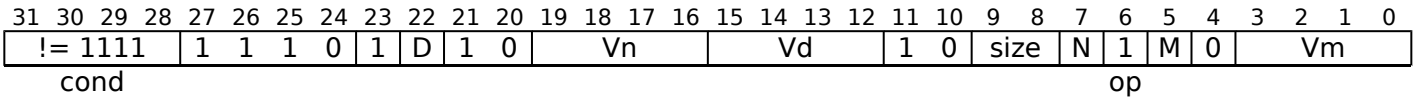
```
VFMS{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm>
```

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



Half-precision scalar (size == 01)
(FEAT_FP16)

```
VFMS{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>
```

Single-precision scalar (size == 10)

```
VFMS{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>
```

Double-precision scalar (size == 11)

```
VFMS{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>
```

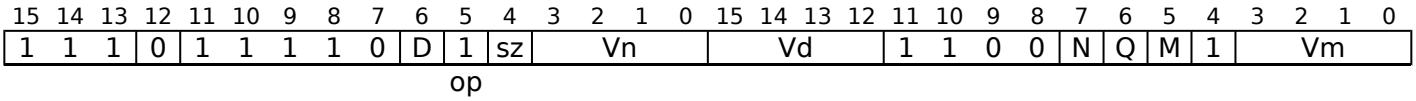
```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
advsimd = FALSE; op1_neg = (op == '1');
case size of
    when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1



64-bit SIMD vector (Q == 0)

```
VFMS{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VFMS{<c>}{<q>}.<dt> <Qd>, <Qn>, <Qm>
```

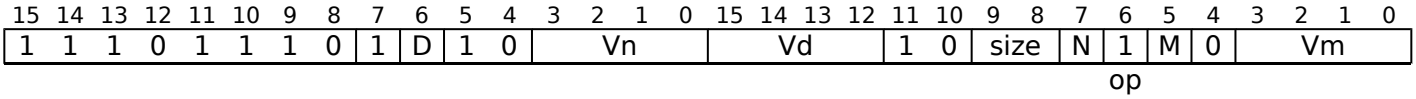
```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
advsimd = TRUE; op1_neg = (op == '1');
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
regs = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `sz == '1' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T2



Half-precision scalar (size == 01)
(FEAT_FP16)

```
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; 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> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.
For encoding A2, T1 and T2: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the vectors, encoded in “sz”:

sz	<dt>
0	F32
1	F16

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

<Sn> Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.

<Sm> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
    if advsimd then // Advanced SIMD instruction
        for r = 0 to regs-1
            for e = 0 to elements-1
                bits(esize) op1 = Elem[D[n+r],e,esize];
                if op1_neg then op1 = FPNeg(op1);
                Elem[D[d+r],e,esize] = FPMulAdd(Elem[D[d+r],e,esize],
                                        op1, Elem[D[m+r],e,esize], StandardFPSCRValue());

    else // VFP instruction
        case esize of
            when 16
                op16 = if op1_neg then FPNeg(S[n]<15:0>) else S[n]<15:0>;
                S[d] = Zeros(16) : FPMulAdd(S[d]<15:0>, op16, S[m]<15:0>, FPSCR[]);
            when 32
                op32 = if op1_neg then FPNeg(S[n]) else S[n];
                S[d] = FPMulAdd(S[d], op32, S[m], FPSCR[]);
            when 64
                op64 = if op1_neg then FPNeg(D[n]) else D[n];
                D[d] = FPMulAdd(D[d], op64, D[m], FPSCR[]);
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VFMSL (vector)

Vector Floating-point Multiply-Subtract Long from accumulator (vector). This instruction negates the values in the vector of one SIMD&FP register, multiplies these with the corresponding values in another vector, and accumulates the product to the corresponding vector element of the destination SIMD&FP register. The instruction does not round the result of the multiply before the accumulation.

Depending on settings in the *CPACR*, *NSACR*, *HCPTR*, and *FPEXC* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

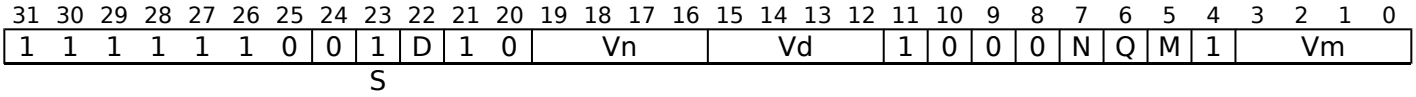
In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

ID_ISAR6.FHM indicates whether this instruction is supported.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

(FEAT_FHM)



64-bit SIMD vector (Q == 0)

```
VFMSL{<q>}.F16 <Dd>, <Sn>, <Sm>
```

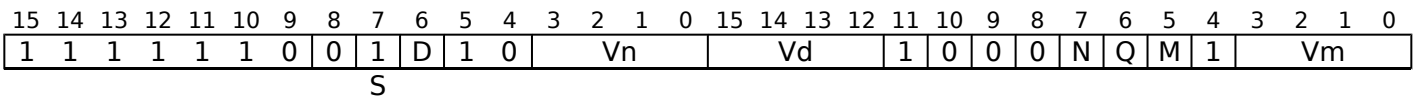
128-bit SIMD vector (Q == 1)

```
VFMSL{<q>}.F16 <Qd>, <Dn>, <Dm>
```

```
if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
if Q == '1' && Vd<0> == '1' then UNDEFINED;

integer d = UInt(D:Vd);
integer n = if Q == '1' then UInt(N:Vn) else UInt(Vn:N);
integer m = if Q == '1' then UInt(M:Vm) else UInt(Vm:M);
integer esize = 32;
integer regs = if Q=='1' then 2 else 1;
integer datasize = if Q=='1' then 64 else 32;
boolean sub_op = S=='1';
```

T1

(FEAT_FHM)



64-bit SIMD vector (Q == 0)

```
VFMSL{<q>}.F16 <Dd>, <Sn>, <Sm>
```

128-bit SIMD vector (Q == 1)

```
VFMSL{<q>}.F16 <Qd>, <Dn>, <Dm>
```

```
if InITBlock() then UNPREDICTABLE;
if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
if Q == '1' && Vd<0> == '1' then UNDEFINED;

integer d = UInt(D:Vd);
integer n = if Q == '1' then UInt(N:Vn) else UInt(Vn:N);
integer m = if Q == '1' then UInt(M:Vm) else UInt(Vm:M);
integer esize = 32;
integer regs = if Q=='1' then 2 else 1;
integer datasize = if Q=='1' then 64 else 32;
boolean sub_op = S=='1';
```

Assembler Symbols

<q>	See *Standard assembler syntax fields*.
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Sn>	Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.
<Sm>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.

Operation

```
CheckAdvSIMDEnabled();
bits(datasize) operand1 ;
bits(datasize) operand2 ;
bits(64) operand3;
bits(64) result;
bits(esize DIV 2) element1;
bits(esize DIV 2) element2;

if Q=='0' then
    operand1 = S[n]<datasize-1:0>;
    operand2 = S[m]<datasize-1:0>;
else
    operand1 = D[n]<datasize-1:0>;
    operand2 = D[m]<datasize-1:0>;
for r = 0 to regs-1
    operand3 = D[d+r];
    for e = 0 to 1
        element1 = Elem[operand1, 2*r+e, esize DIV 2];
        element2 = Elem[operand2, 2*r+e, esize DIV 2];
        if sub_op then element1 = FPNeg(element1);
        Elem[result, e, esize] = FPMulAddH(Elem[operand3, e, esize], element1, element2, StandardFPSCRVal
    D[d+r] = result;
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VFMSL (by scalar)

Vector Floating-point Multiply-Subtract Long from accumulator (by scalar). This instruction multiplies the negated vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, and accumulates the product to the corresponding vector element of the destination SIMD&FP register. The instruction does not round the result of the multiply before the accumulation.

Depending on settings in the *CPACR*, *NSACR*, *HCPTR*, and *FPEXC* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

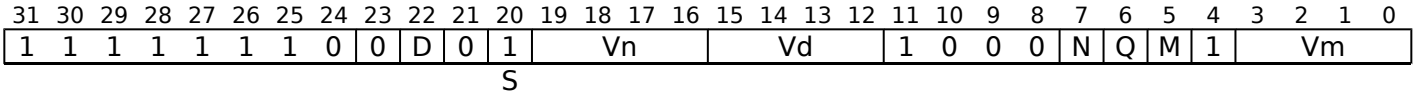
In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

ID_ISAR6.FHM indicates whether this instruction is supported.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

(FEAT_FHM)



64-bit SIMD vector (Q == 0)

```
VFMSL{<q>}.F16 <Dd>, <Sn>, <Sm>[<index>]
```

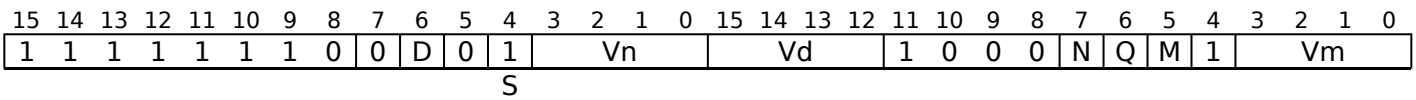
128-bit SIMD vector (Q == 1)

```
VFMSL{<q>}.F16 <Qd>, <Dn>, <Dm>[<index>]
```

```
if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
if Q == '1' && Vd<0> == '1' then UNDEFINED;

integer d = UInt(D:Vd);
integer n = if Q == '1' then UInt(N:Vn) else UInt(Vn:N);
integer m = if Q == '1' then UInt(Vm<2:0>) else UInt(Vm<2:0>:M);

integer index = if Q == '1' then UInt(M:Vm<3>) else UInt(Vm<3>);
integer esize = 32;
integer regs = if Q=='1' then 2 else 1;
integer datasize = if Q=='1' then 64 else 32;
boolean sub_op = S=='1';
```

T1

(FEAT_FHM)



64-bit SIMD vector (Q == 0)

```
VFMSL{<q>}.F16 <Dd>, <Sn>, <Sm>[<index>]
```

128-bit SIMD vector (Q == 1)

```
VFMSL{<q>}.F16 <Qd>, <Dn>, <Dm>[<index>]
```

```
if InITBlock() then UNPREDICTABLE;
if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
if Q == '1' && Vd<0> == '1' then UNDEFINED;

integer d = UInt(D:Vd);
integer n = if Q == '1' then UInt(N:Vn) else UInt(Vn:N);
integer m = if Q == '1' then UInt(Vm<2:0>) else UInt(Vm<2:0>:M);

integer index = if Q == '1' then UInt(M:Vm<3>) else UInt(Vm<3>);
integer esize = 32;
integer regs = if Q=='1' then 2 else 1;
integer datasize = if Q=='1' then 64 else 32;
boolean sub_op = S=='1';
```

Assembler Symbols

<q>	See *Standard assembler syntax fields*.
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "Vm<2:0>" field.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Sn>	Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.
<Sm>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm<2:0>:M" field.
<index>	For the 64-bit SIMD vector variant: is the element index in the range 0 to 1, encoded in the "Vm<3>" field. For the 128-bit SIMD vector variant: is the element index in the range 0 to 3, encoded in the "M:Vm<3>" field.

Operation

```
CheckAdvSIMDEnabled();
bits(datasize) operand1 ;
bits(datasize) operand2 ;
bits(64) operand3;
bits(64) result;
bits(esize DIV 2) element1;
bits(esize DIV 2) element2;

if Q=='0' then
    operand1 = S[n]<datasize-1:0>;
    operand2 = S[m]<datasize-1:0>;
else
    operand1 = D[n]<datasize-1:0>;
    operand2 = D[m]<datasize-1:0>;
element2 = Elem[operand2, index, esize DIV 2];
for r = 0 to regs-1
    operand3 = D[d+r];
    for e = 0 to 1
        element1 = Elem[operand1, 2*r+e, esize DIV 2];
        if sub_op then element1 = FPNeg(element1);
        Elem[result, e, esize] = FPMulAddH(Elem[operand3, e, esize], element1, element2, StandardFPSCRVal
    D[d+r] = result;
```

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

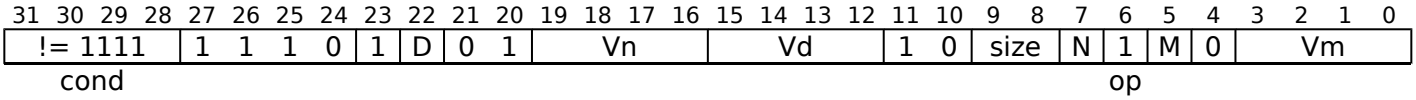
VFNMA

Vector Fused Negate Multiply Accumulate negates one floating-point register value and multiplies it by another floating-point register value, adds the negation of the floating-point value in the destination register to the product, and writes the result back to the destination register. The instruction does not round the result of the multiply before the addition.

Depending on settings in the *CPACR*, *NSACR*, *HCPTR*, and *FPEXC* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



Half-precision scalar (size == 01)
(FEAT_FP16)

```
VFNMA{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>
```

Single-precision scalar (size == 10)

```
VFNMA{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>
```

Double-precision scalar (size == 11)

```
VFNMA{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>
```

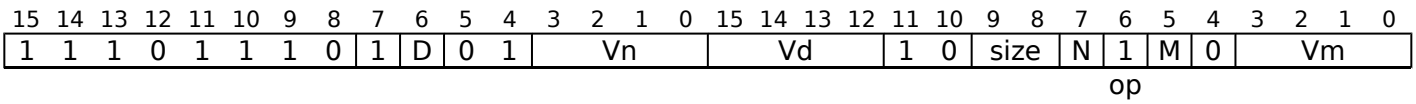
```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
op1_neg = (op == '1');
case size of
    when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    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' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1



Half-precision scalar (size == 01)
(FEAT_FP16)

```
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:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
<Sn>	Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.
<Sm>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
    case esize of
        when 16
            op16 = if op1_neg then FPNeg(S[n]<15:0>) else S[n]<15:0>;
            S[d] = Zeros(16) : FPMulAdd(FPNeg(S[d]<15:0>), op16, S[m]<15:0>, FPSCR[]);
        when 32
            op32 = if op1_neg then FPNeg(S[n]) else S[n];
            S[d] = FPMulAdd(FPNeg(S[d]), op32, S[m], FPSCR[]);
        when 64
            op64 = if op1_neg then FPNeg(D[n]) else D[n];
            D[d] = FPMulAdd(FPNeg(D[d]), op64, D[m], FPSCR[]);
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

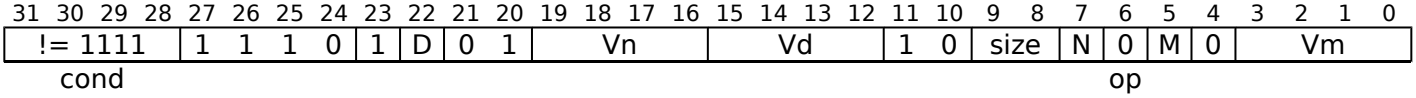
VFNMS

Vector Fused Negate Multiply Subtract multiplies together two floating-point register values, adds the negation of the floating-point value in the destination register to the product, and writes the result back to the destination register. The instruction does not round the result of the multiply before the addition.

Depending on settings in the *CPACR*, *NSACR*, *HCPTR*, and *FPEXC* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



Half-precision scalar (size == 01)
(FEAT_FP16)

```
VFNMS{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>
```

Single-precision scalar (size == 10)

```
VFNMS{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>
```

Double-precision scalar (size == 11)

```
VFNMS{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>
```

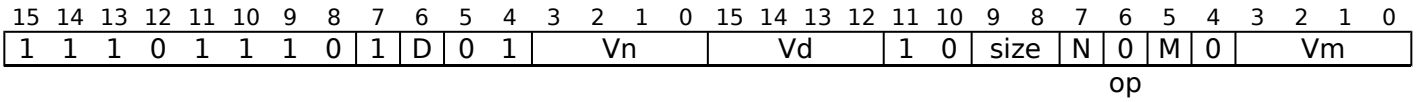
```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
op1_neg = (op == '1');
case size of
    when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    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' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1



Half-precision scalar (size == 01)
(FEAT_FP16)

```
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' && 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(); 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[]);
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VHADD

Vector Halving Add adds corresponding elements in two vectors of integers, shifts each result right one bit, and places the final results in the destination vector. The results of the halving operations are truncated. For rounded results, see *VRHADD*).

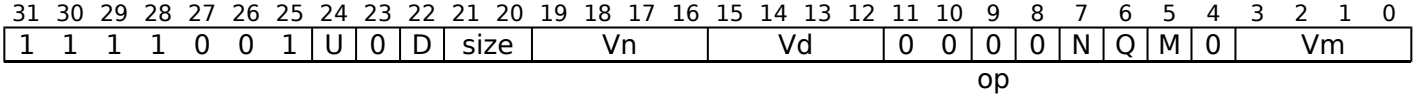
The operand and result elements are all the same type, and can be any one of:

- 8-bit, 16-bit, or 32-bit signed integers.
- 8-bit, 16-bit, or 32-bit unsigned integers.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



64-bit SIMD vector (Q == 0)

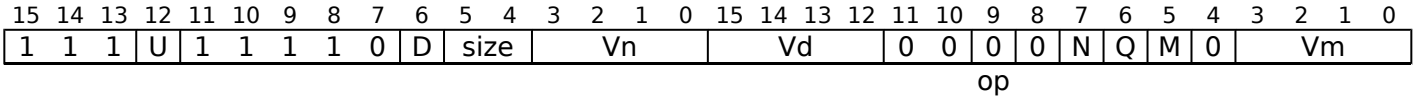
```
VHADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VHADD{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
add = (op == '0');  unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1



64-bit SIMD vector (Q == 0)

```
VHADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VHADD{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
add = (op == '0');  unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>	For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional. For encoding T1: see *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the operands, encoded in “U:size”:

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	00	U8
1	01	U16
1	10	U32

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Int(Elem[D[n+r],e,esize], unsigned);
            op2 = Int(Elem[D[m+r],e,esize], unsigned);
            result = if add then op1+op2 else op1-op2;
            Elem[D[d+r],e,esize] = result<esize:1>;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VHSUB

Vector Halving Subtract subtracts the elements of the second operand from the corresponding elements of the first operand, shifts each result right one bit, and places the final results in the destination vector. The results of the halving operations are truncated. There is no rounding version.

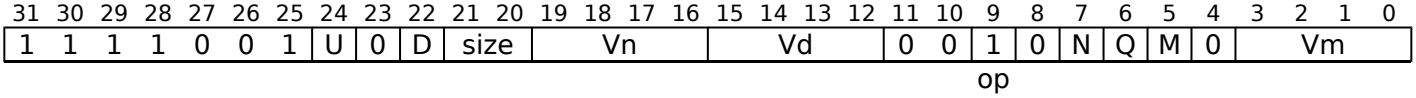
The operand and result elements are all the same type, and can be any one of:

- 8-bit, 16-bit, or 32-bit signed integers.
- 8-bit, 16-bit, or 32-bit unsigned integers.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



64-bit SIMD vector (Q == 0)

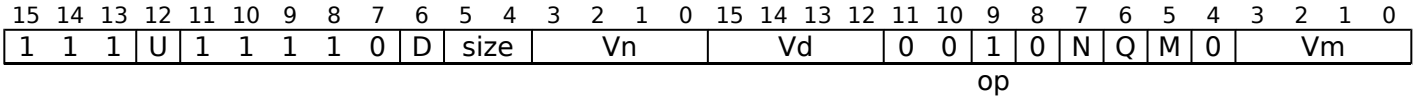
```
VHSUB{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VHSUB{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
add = (op == '0');  unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1



64-bit SIMD vector (Q == 0)

```
VHSUB{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VHSUB{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
add = (op == '0');  unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>	For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional. For encoding T1: see *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the operands, encoded in “U:size”:

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	00	U8
1	01	U16
1	10	U32

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Int(Elem[D[n+r],e,esize], unsigned);
            op2 = Int(Elem[D[m+r],e,esize], unsigned);
            result = if add then op1+op2 else op1-op2;
            Elem[D[d+r],e,esize] = result<esize:1>;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

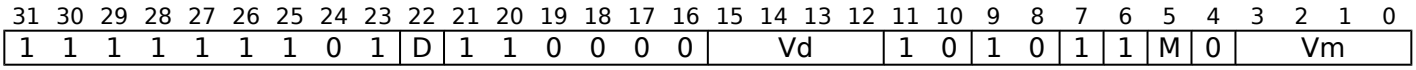
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VINS

Vector move Insertion. This instruction copies the lower 16 bits of the 32-bit source SIMD&FP register into the upper 16 bits of the 32-bit destination SIMD&FP register, while preserving the values in the remaining bits.
Depending on settings in the *CPACR*, *NSACR*, *HCPTR*, and *FPEXC* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

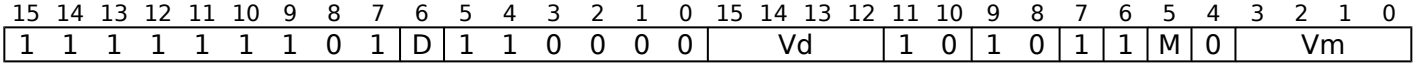
A1
(FEAT_FP16)



A1

```
VINS{<q>}.F16 <Sd>, <Sm>

if !HaveFP16Ext() then UNDEFINED;
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
d = UInt(Vd:D); m = UInt(Vm:M);
```

T1
(FEAT_FP16)



T1

```
VINS{<q>}.F16 <Sd>, <Sm>

if InITBlock() then UNPREDICTABLE;
if !HaveFP16Ext() then UNDEFINED;
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
d = UInt(Vd:D); m = UInt(Vm:M);
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<q>	See *Standard assembler syntax fields*.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
<Sm>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    S[d] = S[m]<15:0> : S[d]<15:0>;
```

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VJCVT

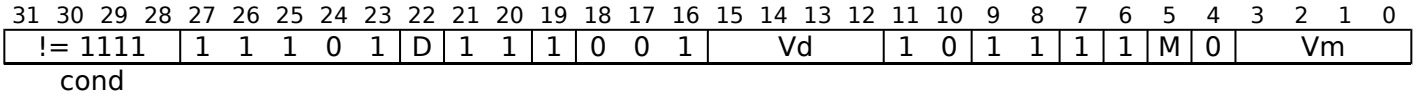
Javascript Convert to signed fixed-point, rounding toward Zero. This instruction converts the double-precision floating-point value in the SIMD&FP source register to a 32-bit signed integer using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register. If the result is too large to be accommodated as a signed 32-bit integer, then the result is the integer modulo 2^{32}, as held in a 32-bit signed integer.

This instruction can generate a floating-point exception. Depending on the settings in *FPSCR*, the exception results in either a flag being set or a synchronous exception being generated. For more information, see *Floating-point exceptions and exception traps*.

Depending on settings in the *CPACR*, *NSACR*, *HCPTR*, and *FPEXC* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1
(FEAT_JSCVT)

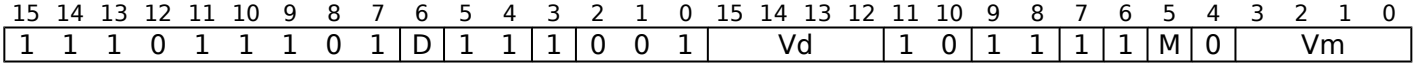


A1

```
VJCVT{<q>}.S32.F64 <Sd>, <Dm>
```

```
if !HaveFJCVTZSExt() then UNDEFINED;
if cond != '1110' then UNPREDICTABLE;
d = UInt(Vd:D);  m = UInt(M:Vm);
```

T1
(FEAT_JSCVT)



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();
CheckVFPEnabled(TRUE);
bits(64) fltval = D[m];
bits(32) intval;
bit      Z;
(intval, Z) = FPToFixedJS(fltval, FPSCR[], FALSE);
FPSCR<31:28> = '0':Z:'00';
S[d] = intval;
```

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

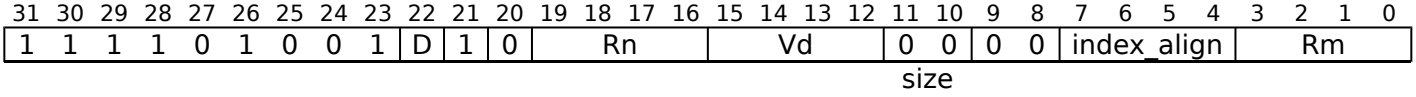
VLD1 (single element to one lane)

Load single 1-element structure to one lane of one register loads one element from memory into one element of a register. Elements of the register that are not loaded are unchanged. For details of the addressing mode see *Advanced SIMD addressing mode*.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1 , A2 and A3) and T32 (T1 , T2 and T3) .

A1



Offset (Rm == 1111)

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

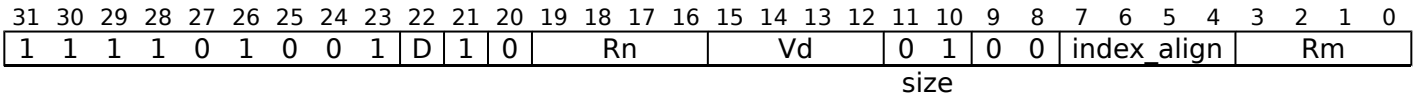
```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

```
if size == '11' then SEE "VLD1 (single element to all lanes)";
if index_align<0> != '0' then UNDEFINED;
ebytes = 1;  index = UInt(index_align<3:1>);  alignment = 1;
d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 then UNPREDICTABLE;
```

A2



Offset (Rm == 1111)

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

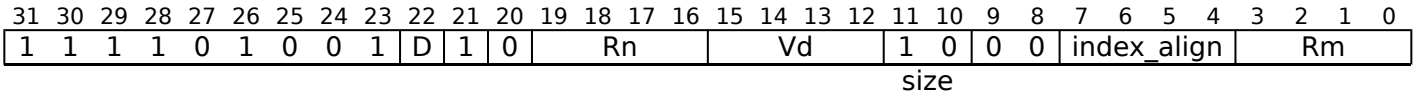
```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

```
if size == '11' then SEE "VLD1 (single element to all lanes)";
if index_align<1> != '0' then UNDEFINED;
ebytes = 2;  index = UInt(index_align<3:2>);
alignment = if index_align<0> == '0' then 1 else 2;
d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 then UNPREDICTABLE;
```

A3



Offset (Rm == 1111)

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

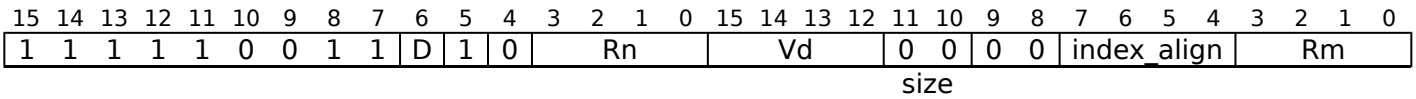
```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

```
if size == '11' then SEE "VLD1 (single element to all lanes)";
if index_align<2> != '0' then UNDEFINED;
if index_align<1:0> != '00' && index_align<1:0> != '11' then UNDEFINED;
ebytes = 4;  index = UInt(index_align<3>);
alignment = if index_align<1:0> == '00' then 1 else 4;
d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 then UNPREDICTABLE;
```

T1



Offset (Rm == 1111)

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

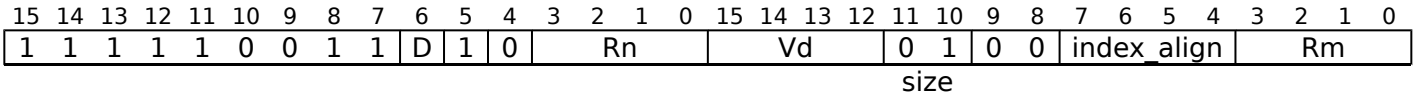
```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

```
if size == '11' then SEE "VLD1 (single element to all lanes)";
if index_align<0> != '0' then UNDEFINED;
ebytes = 1;  index = UInt(index_align<3:1>);  alignment = 1;
d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 then UNPREDICTABLE;
```

T2



Offset (Rm == 1111)

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

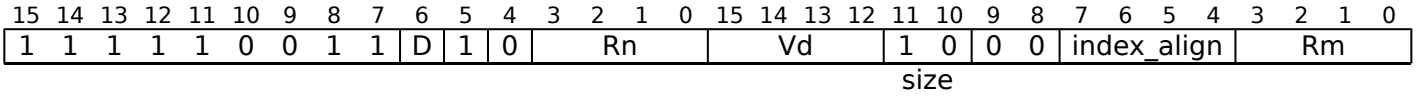
```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

```
if size == '11' then SEE "VLD1 (single element to all lanes)";
if index_align<1> != '0' then UNDEFINED;
ebytes = 2;  index = UInt(index_align<3:2>);
alignment = if index_align<0> == '0' then 1 else 2;
d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 then UNPREDICTABLE;
```

T3



Offset (Rm == 1111)

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

```
if size == '11' then SEE "VLD1 (single element to all lanes)";
if index_align<2> != '0' then UNDEFINED;
if index_align<1:0> != '00' && index_align<1:0> != '11' then UNDEFINED;
ebytes = 4;  index = UInt(index_align<3>);
alignment = if index_align<1:0> == '00' then 1 else 4;
d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 then UNPREDICTABLE;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c> For encoding A1, A2 and A3: see *Standard assembler syntax fields*. This encoding must be unconditional.

For encoding T1, T2 and T3: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<size> Is the data size, encoded in “size”:

size	<size>
00	8
01	16
10	32

<list> Is a list containing the single 64-bit name of the SIMD&FP register holding the element.
The list must be { <Dd>[<index>] }.
The register <Dd> is encoded in the "D:Vd" field.
The permitted values and encoding of <index> depend on <size>:

<size> == 8
<index> is in the range 0 to 7, encoded in the "index_align<3:1>" field.

<size> == 16
<index> is in the range 0 to 3, encoded in the "index_align<3:2>" field.

<size> == 32
<index> is 0 or 1, encoded in the "index_align<3>" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

<align> When <size> == 8, <align> must be omitted, otherwise it is the optional alignment.
Whenever <align> is omitted, the standard alignment is used, see *Unaligned data access*, and the encoding depends on <size>:

<size> == 8
Encoded in the "index_align<0>" field as 0.

<size> == 16
Encoded in the "index_align<1:0>" field as 0b00.

<size> == 32
Encoded in the "index_align<2:0>" field as 0b000.

Whenever <align> is present, the permitted values and encoding depend on <size>:

<size> == 16

<align> is 16, meaning 16-bit alignment, encoded in the "index_align<1:0>" field as 0b01.

<size> == 32

<align> is 32, meaning 32-bit alignment, encoded in the "index_align<2:0>" field as 0b011.

: is the preferred separator before the <align> value, but the alignment can be specified as @<align>, see *Advanced SIMD addressing mode*.

<Rm> Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see *Advanced SIMD addressing mode*.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    address = R[n];  iswrite = FALSE;
    - = AArch32.CheckAlignment(address, alignment, AccType_VEC, iswrite);
    Elem[D[d],index] = MemU[address,ebytes];
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + ebytes;
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

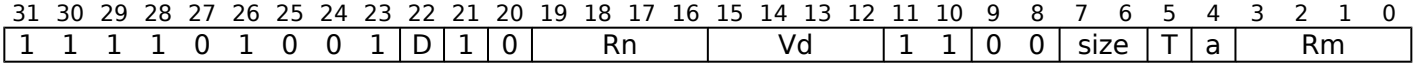
VLD1 (single element to all lanes)

Load single 1-element structure and replicate to all lanes of one register loads one element from memory into every element of one or two vectors. For details of the addressing mode see *Advanced SIMD addressing mode*.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



Offset (Rm == 1111)

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}],<Rm>
```

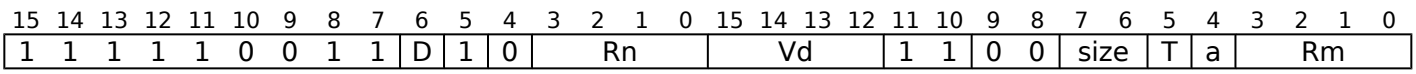
```
if size == '11' || (size == '00' && a == '1') then UNDEFINED;
ebytes = 1 << UInt(size);  regs = if T == '0' then 1 else 2;
alignment = if a == '0' then 1 else ebytes;
d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d+regs > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T1



Offset (Rm == 1111)

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

```
if size == '11' || (size == '00' && a == '1') then UNDEFINED;
ebytes = 1 << UInt(size);  regs = if T == '0' then 1 else 2;
alignment = if a == '0' then 1 else ebytes;
d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d+regs > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *VLD1 (single element to all lanes)*.

Assembler Symbols

<c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<size> Is the data size, encoded in “size”:

size	<size>
00	8
01	16
10	32
11	RESERVED

<list> Is a list containing the 64-bit names of the SIMD&FP registers.

The list must be one of:

{ <Dd>[] }

Encoded in the "T" field as 0.

{ <Dd>[], <Dd+1>[] }

Encoded in the "T" field as 1.

The register <Dd> is encoded in the "D:Vd" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

<align> When <size> == 8, <align> must be omitted, otherwise it is the optional alignment.

Whenever <align> is omitted, the standard alignment is used, see *Unaligned data access*, and is encoded in the "a" field as 0.

Whenever <align> is present, the permitted values and encoding depend on <size>:

<size> == 16

<align> is 16, meaning 16-bit alignment, encoded in the "a" field as 1.

<size> == 32
<align> is 32, meaning 32-bit alignment, encoded in the "a" field as 1.

: is the preferred separator before the <align> value, but the alignment can be specified as @<align>, see *Advanced SIMD addressing mode*.

<Rm> Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see *Advanced SIMD addressing mode*.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    address = R[n];  iswrite = FALSE;
    - = AArch32.CheckAlignment(address, alignment, AccType_VEC, iswrite);
    bits(64) replicated_element = Replicate(MemU[address,ebytes]);
    for r = 0 to regs-1
        D[d+r] = replicated_element;
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + ebytes;
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

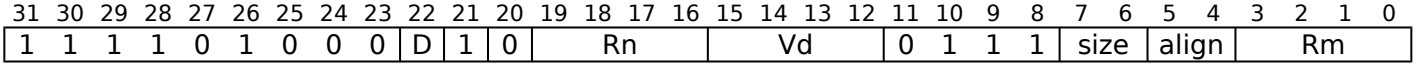
VLD1 (multiple single elements)

Load multiple single 1-element structures to one, two, three, or four registers loads elements from memory into one, two, three, or four registers, without de-interleaving. Every element of each register is loaded. For details of the addressing mode see *Advanced SIMD addressing mode*.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1 , A2 , A3 and A4) and T32 (T1 , T2 , T3 and T4) .

A1



Offset (Rm == 1111)

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

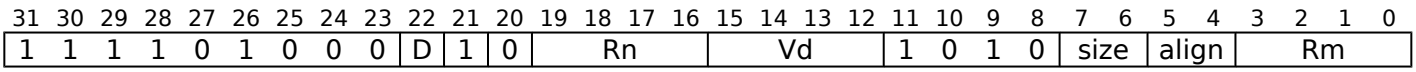
```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

```
regs = 1;  if align<1> == '1' then UNDEFINED;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 then UNPREDICTABLE;
```

A2



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>
```

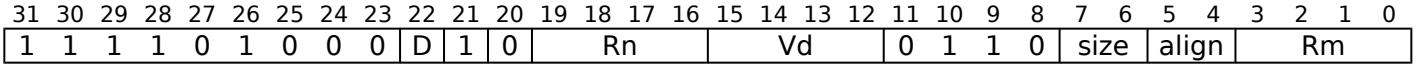
```
regs = 2;  if align == '11' then UNDEFINED;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UNPREDICTABLE;
```

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



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>
```

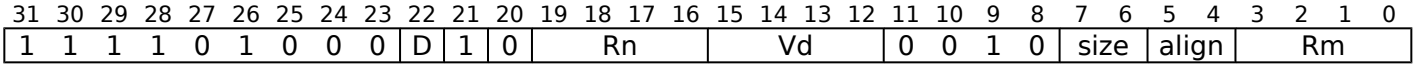
```
regs = 3;  if align<1> == '1' then UNDEFINED;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UNPREDICTABLE;
```

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



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>
```

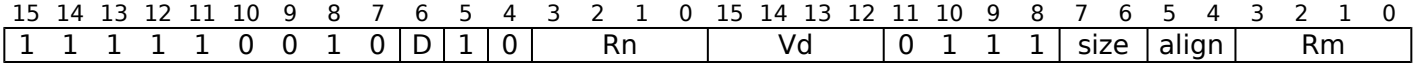
```
regs = 4;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UNPREDICTABLE;
```

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



Offset (Rm == 1111)

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

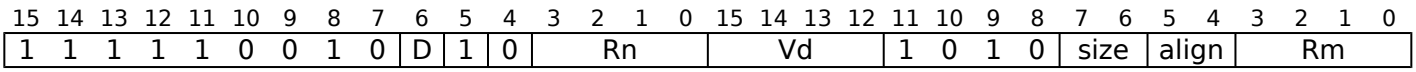
```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

```
regs = 1;  if align<1> == '1' then UNDEFINED;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 then UNPREDICTABLE;
```

T2



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>
```

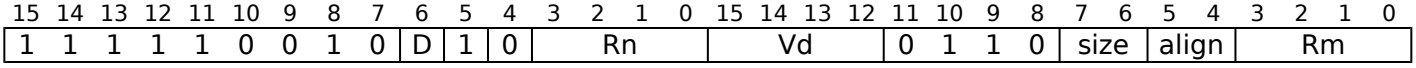
```
regs = 2;  if align == '11' then UNDEFINED;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UNPREDICTABLE;
```

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



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>
```

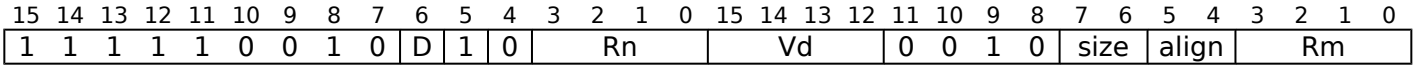
```
regs = 3;  if align<1> == '1' then UNDEFINED;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UNPREDICTABLE;
```

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



Offset (Rm == 1111)

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

```
regs = 4;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d+regs > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *VLD1 (multiple single elements)*.

Related encodings: See *Advanced SIMD element or structure load/store* for the T32 instruction set, or *Advanced SIMD element or structure load/store* for the A32 instruction set.

Assembler Symbols

<c> For encoding A1, A2, A3 and A4: see *Standard assembler syntax fields*. This encoding must be unconditional.

For encoding T1, T2, T3 and T4: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<size> Is the data size, encoded in “size”:

size	<size>
00	8
01	16
10	32
11	64

<list> Is a list containing the 64-bit names of the SIMD&FP registers.

The list must be one of:

{ <Dd> }
Single register. Selects the A1 and T1 encodings of the instruction.

{ <Dd>, <Dd+1> }
Two single-spaced registers. Selects the A2 and T2 encodings of the instruction.

{ <Dd>, <Dd+1>, <Dd+2> }
Three single-spaced registers. Selects the A3 and T3 encodings of the instruction.

{ <Dd>, <Dd+1>, <Dd+2>, <Dd+3> }
Four single-spaced registers. Selects the A4 and T4 encodings of the instruction.

The register <Dd> is encoded in the "D:Vd" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

<align> Is the optional alignment.

Whenever <align> is omitted, the standard alignment is used, see *Unaligned data access*, and is encoded in the "align" field as 0b00.

Whenever <align> is present, the permitted values are:

64

64-bit alignment, encoded in the "align" field as 0b01.

128

128-bit alignment, encoded in the "align" field as 0b10. Available only if <list> contains two or four registers.

256

256-bit alignment, encoded in the "align" field as 0b11. Available only if <list> contains four registers.

: is the preferred separator before the <align> value, but the alignment can be specified as @<align>, see *Advanced SIMD addressing mode*.

<Rm> Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see *Advanced SIMD addressing mode*.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    address = R[n];  iswrite = FALSE;
    - = AArch32.CheckAlignment(address, alignment, AccType_VEC, iswrite);
    for r = 0 to regs-1
        for e = 0 to elements-1
            bits(ebytes*8) data;
            if ebytes != 8 then
                data = MemU[address,ebytes];
            else
                - = AArch32.CheckAlignment(address, ebytes, AccType_NORMAL, iswrite);
                data<31:0> = if BigEndian(AccType_NORMAL) then MemU[address+4,4] else MemU[address,4];
                data<63:32> = if BigEndian(AccType_NORMAL) then MemU[address,4] else MemU[address+4,4];
            Elem[D[d+r],e] = data;
            address = address + ebytes;
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 8*regs;
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

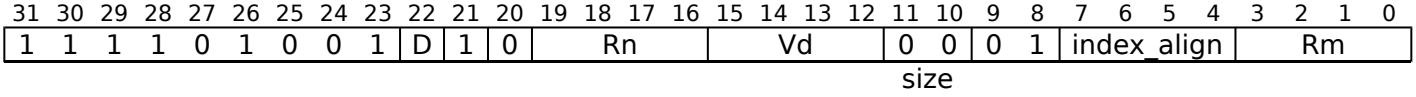
VLD2 (single 2-element structure to one lane)

Load single 2-element structure to one lane of two registers loads one 2-element structure from memory into corresponding elements of two registers. Elements of the registers that are not loaded are unchanged. For details of the addressing mode see *Advanced SIMD addressing mode*.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1 , A2 and A3) and T32 (T1 , T2 and T3) .

A1



Offset (Rm == 1111)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

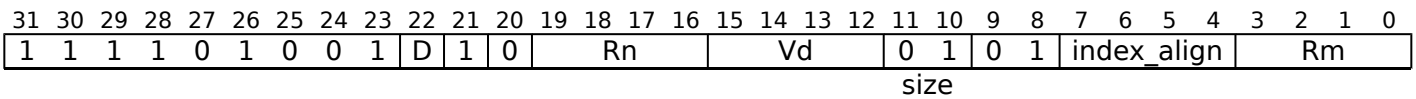
```
if size == '11' then SEE "VLD2 (single 2-element structure to all lanes)";
ebytes = 1;  index = UInt(index_align<3:1>);  inc = 1;
alignment = if index_align<0> == '0' then 1 else 2;
d = UInt(D:Vd);  d2 = d + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d2 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

A2



Offset (Rm == 1111)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

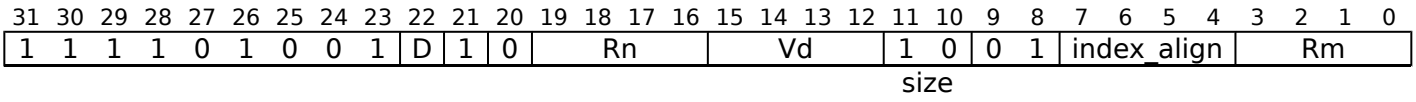
```
if size == '11' then SEE "VLD2 (single 2-element structure to all lanes)";
ebytes = 2;  index = UInt(index_align<3:2>);
inc = if index_align<1> == '0' then 1 else 2;
alignment = if index_align<0> == '0' then 1 else 4;
d = UInt(D:Vd);  d2 = d + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d2 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

A3



Offset (Rm == 1111)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

```
if size == '11' then SEE "VLD2 (single 2-element structure to all lanes)";
if index_align<1> != '0' then UNDEFINED;
ebytes = 4;  index = UInt(index_align<3>);
inc = if index_align<2> == '0' then 1 else 2;
alignment = if index_align<0> == '0' then 1 else 8;
d = UInt(D:Vd);  d2 = d + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UNPREDICTABLE;
```

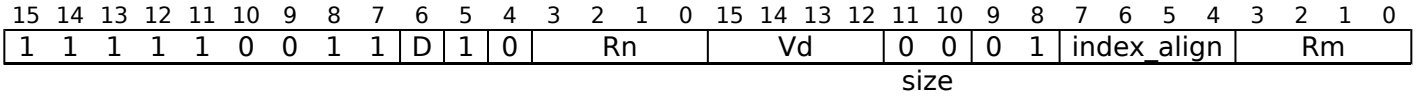
CONSTRAINED UNPREDICTABLE behavior

If `d2 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T1



Offset (Rm == 1111)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

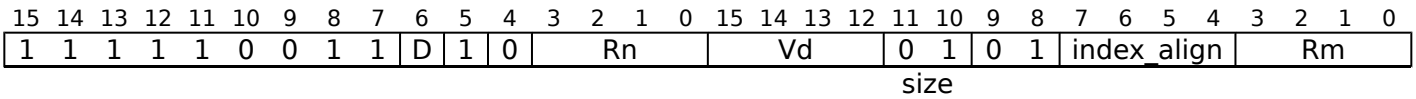
```
if size == '11' then SEE "VLD2 (single 2-element structure to all lanes)";
ebytes = 1;  index = UInt(index_align<3:1>);  inc = 1;
alignment = if index_align<0> == '0' then 1 else 2;
d = UInt(D:Vd);  d2 = d + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d2 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T2



Offset (Rm == 1111)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

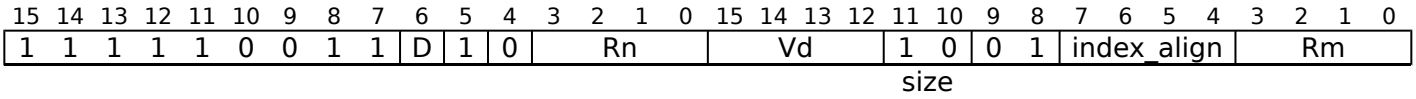
```
if size == '11' then SEE "VLD2 (single 2-element structure to all lanes)";
ebytes = 2;  index = UInt(index_align<3:2>);
inc = if index_align<1> == '0' then 1 else 2;
alignment = if index_align<0> == '0' then 1 else 4;
d = UInt(D:Vd);  d2 = d + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d2 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T3



Offset (Rm == 1111)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

```
if size == '11' then SEE "VLD2 (single 2-element structure to all lanes)";
if index_align<1> != '0' then UNDEFINED;
ebytes = 4;  index = UInt(index_align<3>);
inc = if index_align<2> == '0' then 1 else 2;
alignment = if index_align<0> == '0' then 1 else 8;
d = UInt(D:Vd);  d2 = d + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d2 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *VLD2 (single 2-element structure to one lane)*.

Assembler Symbols

<c> For encoding A1, A2 and A3: see *Standard assembler syntax fields*. This encoding must be unconditional.

For encoding T1, T2 and T3: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<size> Is the data size, encoded in “size”:

size	<size>
00	8
01	16
10	32

<list> Is a list containing the 64-bit names of the two SIMD&FP registers holding the element.

The list must be one of:

{ <Dd>[<index>], <Dd+1>[<index>] }
Single-spaced registers, encoded as "spacing" = 0.

{ <Dd>[<index>], <Dd+2>[<index>] }
Double-spaced registers, encoded as "spacing" = 1. Not permitted when <size> == 8.

The encoding of "spacing" depends on <size>:

<size> == 16
"spacing" is encoded in the "index_align<1>" field.

<size> == 32
"spacing" is encoded in the "index_align<2>" field.

The register <Dd> is encoded in the "D:Vd" field.

The permitted values and encoding of <index> depend on <size>:

<size> == 8
<index> is in the range 0 to 7, encoded in the "index_align<3:1>" field.

<size> == 16
<index> is in the range 0 to 3, encoded in the "index_align<3:2>" field.

<size> == 32
<index> is 0 or 1, encoded in the "index_align<3>" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

<align> Is the optional alignment.

Whenever <align> is omitted, the standard alignment is used, see *Unaligned data access*, and the encoding depends on <size>:

<size> == 8
Encoded in the "index_align<0>" field as 0.

<size> == 16
Encoded in the "index_align<0>" field as 0.

<size> == 32
Encoded in the "index_align<1:0>" field as 0b00.

Whenever <align> is present, the permitted values and encoding depend on <size>:

<size> == 8
<align> is 16, meaning 16-bit alignment, encoded in the "index_align<0>" field as 1.

<size> == 16
<align> is 32, meaning 32-bit alignment, encoded in the "index_align<0>" field as 1.

<size> == 32
<align> is 64, meaning 64-bit alignment, encoded in the "index_align<1:0>" field as 0b01.

: is the preferred separator before the <align> value, but the alignment can be specified as @<align>, see *Advanced SIMD addressing mode*.

<Rm> Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see *Advanced SIMD addressing mode*.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    address = R[n];  iswrite = FALSE;
    - = AArch32.CheckAlignment(address, alignment, AccType_VEC, iswrite);
    Elem[D[d], index] = MemU[address,ebytes];
    Elem[D[d2],index] = MemU[address+ebytes,ebytes];
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 2*ebytes;
```

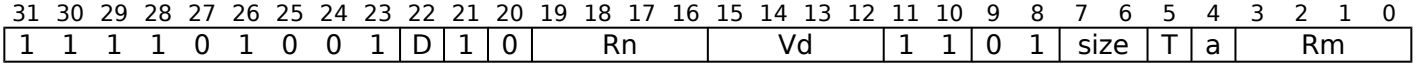
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VLD2 (single 2-element structure to all lanes)

Load single 2-element structure and replicate to all lanes of two registers loads one 2-element structure from memory into all lanes of two registers. For details of the addressing mode see *Advanced SIMD addressing mode*.
Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



Offset (Rm == 1111)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}],<Rm>
```

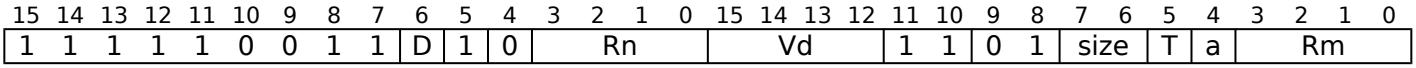
```
if size == '11' then UNDEFINED;
ebytes = 1 << UInt(size);
alignment = if a == '0' then 1 else 2*ebytes;
inc = if T == '0' then 1 else 2;
d = UInt(D:Vd);  d2 = d + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d2 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T1



Offset (Rm == 1111)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

```
if size == '11' then UNDEFINED;
ebytes = 1 << UInt(size);
alignment = if a == '0' then 1 else 2*ebytes;
inc = if T == '0' then 1 else 2;
d = UInt(D:Vd);  d2 = d + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d2 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *VLD2 (single 2-element structure to all lanes)*.

Assembler Symbols

<c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<size> Is the data size, encoded in “size”:

size	<size>
00	8
01	16
10	32
11	RESERVED

<list> Is a list containing the 64-bit names of two SIMD&FP registers.

The list must be one of:

{ <Dd>[], <Dd+1>[] }

Single-spaced registers, encoded in the "T" field as 0.

{ <Dd>[], <Dd+2>[] }

Double-spaced registers, encoded in the "T" field as 1.

The register <Dd> is encoded in the "D:Vd" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

<align> Is the optional alignment.

Whenever <align> is omitted, the standard alignment is used, see *Unaligned data access*, and is encoded in the "a" field as 0.

Whenever <align> is present, the permitted values and encoding depend on <size>:

<size> == 8

<align> is 16, meaning 16-bit alignment, encoded in the "a" field as 1.

<size> == 16
<align> is 32, meaning 32-bit alignment, encoded in the "a" field as 1.

<size> == 32
<align> is 64, meaning 64-bit alignment, encoded in the "a" field as 1.

: is the preferred separator before the <align> value, but the alignment can be specified as @<align>, see *Advanced SIMD addressing mode*.

<Rm> Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see *Advanced SIMD addressing mode*.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    address = R[n];  iswrite = FALSE;
    - = AArch32.CheckAlignment(address, alignment, AccType_VEC, iswrite);
    D[d] = Replicate(MemU[address,ebytes]);
    D[d2] = Replicate(MemU[address+ebytes,ebytes]);
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 2*ebytes;
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

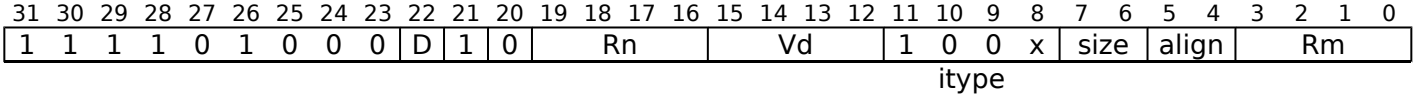
VLD2 (multiple 2-element structures)

Load multiple 2-element structures to two or four registers loads multiple 2-element structures from memory into two or four registers, with de-interleaving. For more information, see *Element and structure load/store instructions*. Every element of each register is loaded. For details of the addressing mode see *Advanced SIMD addressing mode*.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2) .

A1



Offset (Rm == 1111)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

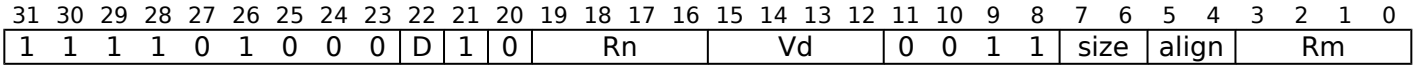
```
pairs = 1;  if align == '11' then UNDEFINED;
if size == '11' then UNDEFINED;
inc = if itype == '1001' then 2 else 1;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
d = UInt(D:Vd);  d2 = d + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d2+pairs > 32 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d2+pairs > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

A2



Offset (Rm == 1111)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

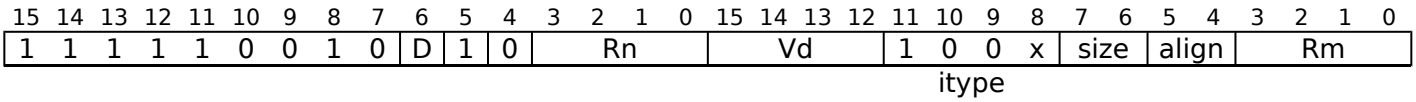
```
pairs = 2;  inc = 2;
if size == '11' then UNDEFINED;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
d = UInt(D:Vd);  d2 = d + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d2+pairs > 32 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d2+pairs > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T1



Offset (Rm == 1111)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

```
pairs = 1;  if align == '11' then UNDEFINED;
if size == '11' then UNDEFINED;
inc = if itype == '1001' then 2 else 1;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
d = UInt(D:Vd);  d2 = d + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d2+pairs > 32 then UNPREDICTABLE;
```

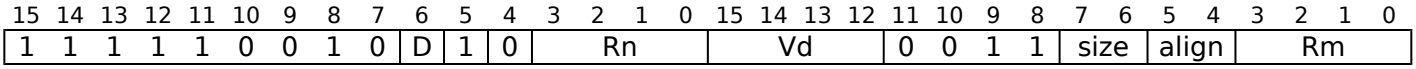
CONSTRAINED UNPREDICTABLE behavior

If `d2+pairs > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T2



Offset (Rm == 1111)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

```
pairs = 2;  inc = 2;
if size == '11' then UNDEFINED;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
d = UInt(D:Vd);  d2 = d + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d2+pairs > 32 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d2+pairs > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *VLD2 (multiple 2-element structures)*.

Related encodings: See *Advanced SIMD element or structure load/store* for the T32 instruction set, or *Advanced SIMD element or structure load/store* for the A32 instruction set.

Assembler Symbols

<c> For encoding A1 and A2: see *Standard assembler syntax fields*. This encoding must be unconditional.

For encoding T1 and T2: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<size> Is the data size, encoded in “size”:

size	<size>
00	8
01	16
10	32
11	RESERVED

<list> Is a list containing the 64-bit names of the SIMD&FP registers.

The list must be one of:

{ <Dd>, <Dd+1> }

Two single-spaced registers. Selects the A1 and T1 encodings of the instruction, and encoded in the "itype" field as 0b1000.

{ <Dd>, <Dd+2> }
Two double-spaced registers. Selects the A1 and T1 encodings of the instruction, and encoded in the "itype" field as 0b1001.

{ <Dd>, <Dd+1>, <Dd+2>, <Dd+3> }
Three single-spaced registers. Selects the A2 and T2 encodings of the instruction.

The register <Dd> is encoded in the "D:Vd" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

<align> Is the optional alignment.
Whenever <align> is omitted, the standard alignment is used, see *Unaligned data access*, and is encoded in the "align" field as 0b00.
Whenever <align> is present, the permitted values are:

64
64-bit alignment, encoded in the "align" field as 0b01.

128
128-bit alignment, encoded in the "align" field as 0b10.

256
256-bit alignment, encoded in the "align" field as 0b11. Available only if <list> contains four registers.

: is the preferred separator before the <align> value, but the alignment can be specified as @<align>, see *Advanced SIMD addressing mode*.

<Rm> Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see *Advanced SIMD addressing mode*.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    address = R[n];  iswrite = FALSE;
    - = AArch32.CheckAlignment(address, alignment, AccType_VEC, iswrite);
    for r = 0 to pairs-1
        for e = 0 to elements-1
            Elem[D[d+r], e] = MemU[address,ebytes];
            Elem[D[d2+r],e] = MemU[address+ebytes,ebytes];
            address = address + 2*ebytes;
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 16*pairs;
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

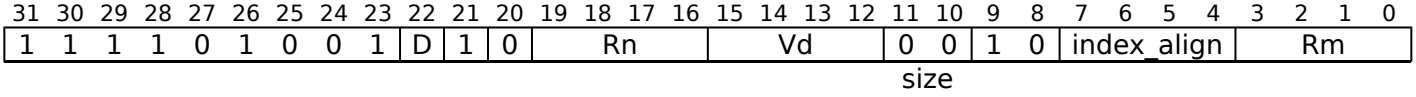
VLD3 (single 3-element structure to one lane)

Load single 3-element structure to one lane of three registers loads one 3-element structure from memory into corresponding elements of three registers. Elements of the registers that are not loaded are unchanged. For details of the addressing mode see *Advanced SIMD addressing mode*.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1 , A2 and A3) and T32 (T1 , T2 and T3) .

A1



Offset (Rm == 1111)

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>]
```

Post-indexed (Rm == 1101)

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>]!
```

Post-indexed (Rm != 11x1)

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>], <Rm>
```

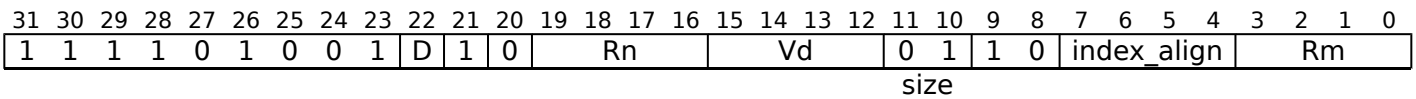
```
if size == '11' then SEE "VLD3 (single 3-element structure to all lanes)";
if index_align<0> != '0' then UNDEFINED;
ebytes = 1;  index = UInt(index_align<3:1>);  inc = 1;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d3 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

A2



Offset (Rm == 1111)

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>]
```

Post-indexed (Rm == 1101)

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>]!
```

Post-indexed (Rm != 11x1)

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>], <Rm>
```

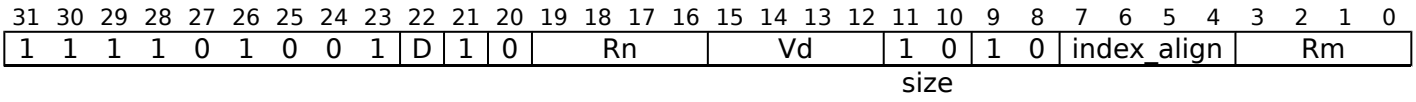
```
if size == '11' then SEE "VLD3 (single 3-element structure to all lanes)";
if index_align<0> != '0' then UNDEFINED;
ebytes = 2;  index = UInt(index_align<3:2>);
inc = if index_align<1> == '0' then 1 else 2;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d3 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

A3



Offset (Rm == 1111)

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>]
```

Post-indexed (Rm == 1101)

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>]!
```

Post-indexed (Rm != 11x1)

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>], <Rm>
```

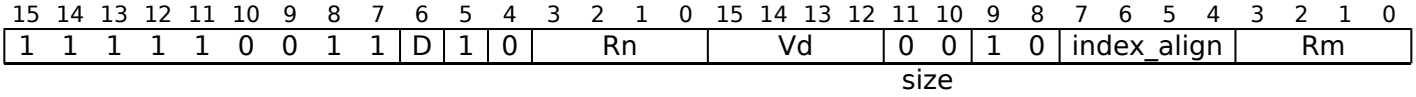
```
if size == '11' then SEE "VLD3 (single 3-element structure to all lanes)";
if index_align<1:0> != '00' then UNDEFINED;
ebytes = 4;  index = UInt(index_align<3>);
inc = if index_align<2> == '0' then 1 else 2;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d3 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T1



Offset (Rm == 1111)

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>]
```

Post-indexed (Rm == 1101)

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>]!
```

Post-indexed (Rm != 11x1)

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>], <Rm>
```

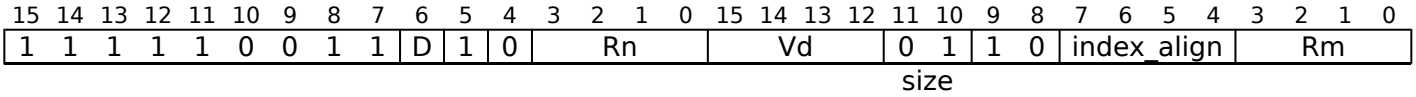
```
if size == '11' then SEE "VLD3 (single 3-element structure to all lanes)";
if index_align<0> != '0' then UNDEFINED;
ebytes = 1;  index = UInt(index_align<3:1>);  inc = 1;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d3 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T2



Offset (Rm == 1111)

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>]
```

Post-indexed (Rm == 1101)

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>]!
```

Post-indexed (Rm != 11x1)

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>], <Rm>
```

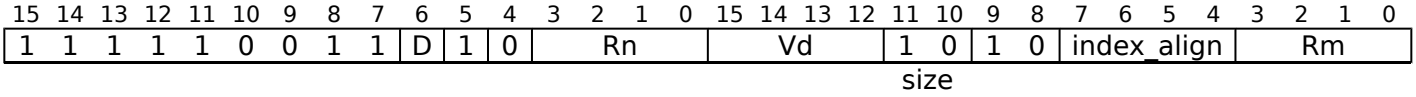
```
if size == '11' then SEE "VLD3 (single 3-element structure to all lanes)";
if index_align<0> != '0' then UNDEFINED;
ebytes = 2;  index = UInt(index_align<3:2>);
inc = if index_align<1> == '0' then 1 else 2;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d3 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T3



Offset (Rm == 1111)

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>]
```

Post-indexed (Rm == 1101)

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>]!
```

Post-indexed (Rm != 11x1)

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>], <Rm>
```

```
if size == '11' then SEE "VLD3 (single 3-element structure to all lanes)";
if index_align<1:0> != '00' then UNDEFINED;
ebytes = 4;  index = UInt(index_align<3>);
inc = if index_align<2> == '0' then 1 else 2;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d3 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *VLD3 (single 3-element structure to one lane)*.

Assembler Symbols

<c> For encoding A1, A2 and A3: see *Standard assembler syntax fields*. This encoding must be unconditional.

For encoding T1, T2 and T3: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<size> Is the data size, encoded in “size”:

size	<size>
00	8
01	16
10	32

<list> Is a list containing the 64-bit names of the three SIMD&FP registers holding the element. The list must be one of:

{ <Dd>[<index>], <Dd+1>[<index>], <Dd+2>[<index>] }
Single-spaced registers, encoded as "spacing" = 0.

{ <Dd>[<index>], <Dd+2>[<index>], <Dd+4>[<index>] }
Double-spaced registers, encoded as "spacing" = 1. Not permitted when <size> == 8.

The encoding of "spacing" depends on <size>:

<size> == 8
"spacing" is encoded in the "index_align<0>" field.

<size> == 16
"spacing" is encoded in the "index_align<1>" field, and "index_align<0>" is set to 0.

<size> == 32
"spacing" is encoded in the "index_align<2>" field, and "index_align<1:0>" is set to 0b00.

The register <Dd> is encoded in the "D:Vd" field.

The permitted values and encoding of <index> depend on <size>:

<size> == 8
<index> is in the range 0 to 7, encoded in the "index_align<3:1>" field.

<size> == 16
<index> is in the range 0 to 3, encoded in the "index_align<3:2>" field.

<size> == 32
<index> is 0 or 1, encoded in the "index_align<3>" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

<Rm> Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see *Advanced SIMD addressing mode*.

Alignment

Standard alignment rules apply, see *Alignment support*.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    address = R[n];
    Elem[D[d], index] = MemU[address,ebytes];
    Elem[D[d2],index] = MemU[address+ebytes,ebytes];
    Elem[D[d3],index] = MemU[address+2*ebytes,ebytes];
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 3*ebytes;
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

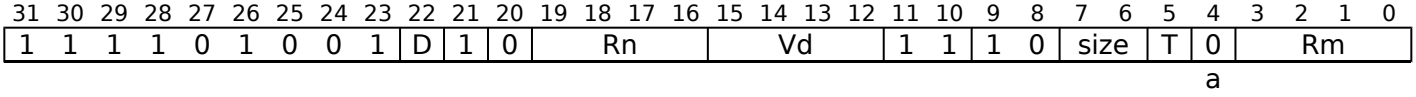
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VLD3 (single 3-element structure to all lanes)

Load single 3-element structure and replicate to all lanes of three registers loads one 3-element structure from memory into all lanes of three registers. For details of the addressing mode see *Advanced SIMD addressing mode*. Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



Offset (Rm == 1111)

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>]
```

Post-indexed (Rm == 1101)

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>]!
```

Post-indexed (Rm != 11x1)

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>], <Rm>
```

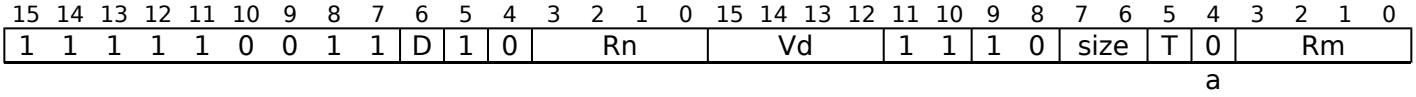
```
if size == '11' || a == '1' then UNDEFINED;
ebytes = 1 << UInt(size);
inc = if T == '0' then 1 else 2;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d3 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T1



Offset (Rm == 1111)

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>]
```

Post-indexed (Rm == 1101)

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>]!
```

Post-indexed (Rm != 11x1)

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>], <Rm>
```

```
if size == '11' || a == '1' then UNDEFINED;
ebytes = 1 << UInt(size);
inc = if T == '0' then 1 else 2;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d3 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *VLD3 (single 3-element structure to all lanes)*.

Assembler Symbols

<c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<size> Is the data size, encoded in “size”:

size	<size>
00	8
01	16
10	32
11	RESERVED

<list> Is a list containing the 64-bit names of three SIMD&FP registers.

The list must be one of:

{ <Dd>[], <Dd+1>[], <Dd+2>[] }

Single-spaced registers, encoded in the "T" field as 0.

{ <Dd>[], <Dd+2>[], <Dd+4>[] }

Double-spaced registers, encoded in the "T" field as 1.

The register <Dd> is encoded in the "D:Vd" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

<Rm> Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see *Advanced SIMD addressing mode*.

Alignment

Standard alignment rules apply, see *Alignment support*.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    address = R[n];
    D[d] = Replicate(MemU[address,ebytes]);
    D[d2] = Replicate(MemU[address+ebytes,ebytes]);
    D[d3] = Replicate(MemU[address+2*ebytes,ebytes]);
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 3*ebytes;
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

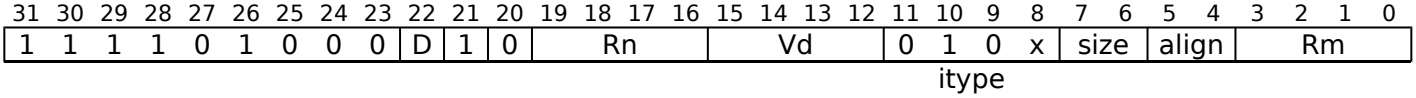
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VLD3 (multiple 3-element structures)

Load multiple 3-element structures to three registers loads multiple 3-element structures from memory into three registers, with de-interleaving. For more information, see *Element and structure load/store instructions*. Every element of each register is loaded. For details of the addressing mode see *Advanced SIMD addressing mode*.
Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



Offset (Rm == 1111)

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

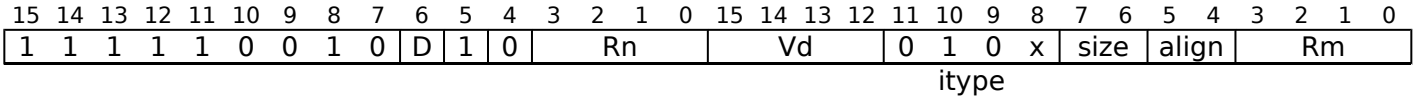
```
case itype of
    when '0100'
        inc = 1;
    when '0101'
        inc = 2;
    otherwise
        SEE "Related encodings";
if size == '11' || align<1> == '1' then UNDEFINED;
alignment = if align<0> == '0' then 1 else 8;
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d3 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T1



Offset (Rm == 1111)

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

```
case itype of
    when '0100'
        inc = 1;
    when '0101'
        inc = 2;
    otherwise
        SEE "Related encodings";
if size == '11' || align<1> == '1' then UNDEFINED;
alignment = if align<0> == '0' then 1 else 8;
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d3 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *VLD3 (multiple 3-element structures)*.

Related encodings: See *Advanced SIMD element or structure load/store* for the T32 instruction set, or *Advanced SIMD element or structure load/store* for the A32 instruction set.

Assembler Symbols

<c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<size> Is the data size, encoded in “size”:

size	<size>
00	8
01	16
10	32
11	RESERVED

<list> Is a list containing the 64-bit names of the SIMD&FP registers.

The list must be one of:

{ <Dd>, <Dd+1>, <Dd+2> }

Single-spaced registers, encoded in the "itype" field as 0b0100.

{ <Dd>, <Dd+2>, <Dd+4> }

Double-spaced registers, encoded in the "itype" field as 0b0101.

The register <Dd> is encoded in the "D:Vd" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

<align> Is the optional alignment.

Whenever <align> is omitted, the standard alignment is used, see *Unaligned data access*, and is encoded in the "align" field as 0b00.

Whenever <align> is present, the only permitted values is 64, meaning 64-bit alignment, encoded in the "align" field as 0b01.

: is the preferred separator before the <align> value, but the alignment can be specified as @<align>, see *Advanced SIMD addressing mode*.

<Rm> Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about <Rn>, !, and <Rm>, see *Advanced SIMD addressing mode*.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    address = R[n];  iswrite = FALSE;
    - = AArch32.CheckAlignment(address, alignment, AccType_VEC, iswrite);
    for e = 0 to elements-1
        Elem[D[d], e] = MemU[address,ebytes];
        Elem[D[d2],e] = MemU[address+ebytes,ebytes];
        Elem[D[d3],e] = MemU[address+2*ebytes,ebytes];
        address = address + 3*ebytes;
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 24;
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

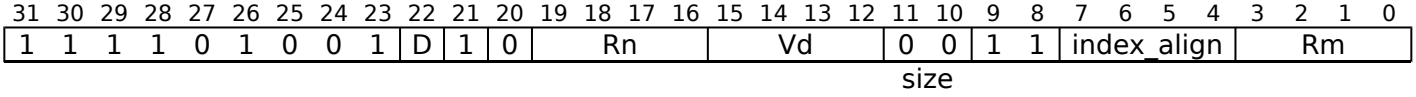
VLD4 (single 4-element structure to one lane)

Load single 4-element structure to one lane of four registers loads one 4-element structure from memory into corresponding elements of four registers. Elements of the registers that are not loaded are unchanged. For details of the addressing mode see *Advanced SIMD addressing mode*.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1 , A2 and A3) and T32 (T1 , T2 and T3) .

A1



Offset (Rm == 1111)

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

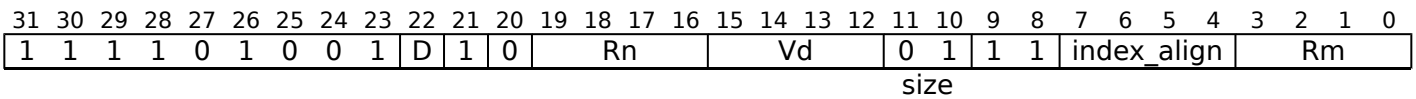
```
if size == '11' then SEE "VLD4 (single 4-element structure to all lanes)";
ebytes = 1;  index = UInt(index_align<3:1>);  inc = 1;
alignment = if index_align<0> == '0' then 1 else 4;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  d4 = d3 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d4 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

A2



Offset (Rm == 1111)

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

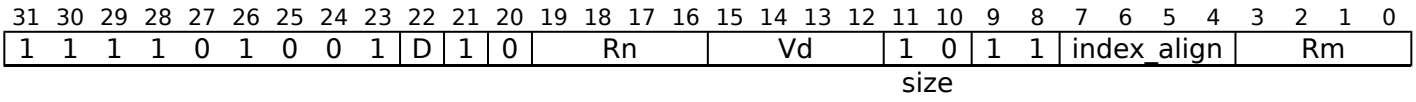
```
if size == '11' then SEE "VLD4 (single 4-element structure to all lanes)";
ebytes = 2;  index = UInt(index_align<3:2>);
inc = if index_align<1> == '0' then 1 else 2;
alignment = if index_align<0> == '0' then 1 else 8;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  d4 = d3 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d4 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

A3



Offset (Rm == 1111)

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

```
if size == '11' then SEE "VLD4 (single 4-element structure to all lanes)";
if index_align<1:0> == '11' then UNDEFINED;
ebytes = 4;  index = UInt(index_align<3>);
inc = if index_align<2> == '0' then 1 else 2;
alignment = if index_align<1:0> == '00' then 1 else 4 << UInt(index_align<1:0>);
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  d4 = d3 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UNPREDICTABLE;
```

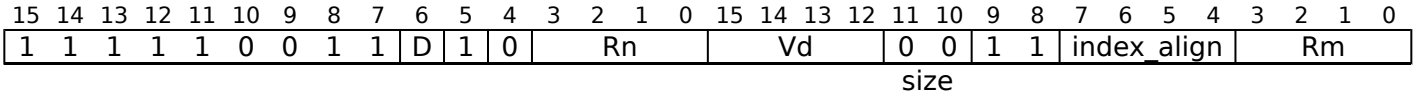
CONSTRAINED UNPREDICTABLE behavior

If `d4 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T1



Offset (Rm == 1111)

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

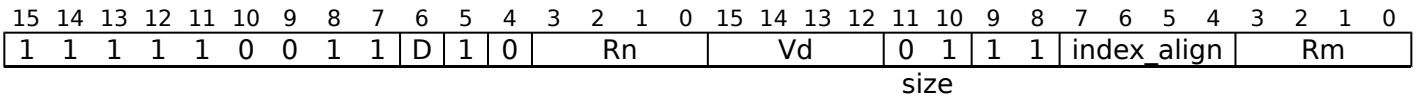
```
if size == '11' then SEE "VLD4 (single 4-element structure to all lanes)";
ebytes = 1;  index = UInt(index_align<3:1>);  inc = 1;
alignment = if index_align<0> == '0' then 1 else 4;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  d4 = d3 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d4 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T2



Offset (Rm == 1111)

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

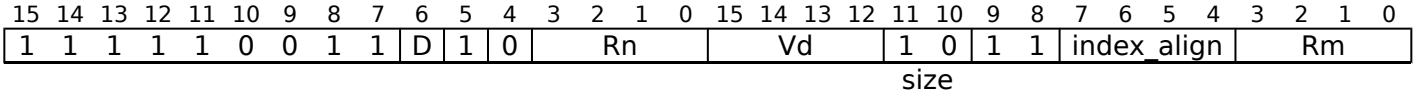
```
if size == '11' then SEE "VLD4 (single 4-element structure to all lanes)";
ebytes = 2;  index = UInt(index_align<3:2>);
inc = if index_align<1> == '0' then 1 else 2;
alignment = if index_align<0> == '0' then 1 else 8;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  d4 = d3 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d4 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T3



Offset (Rm == 1111)

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

```
if size == '11' then SEE "VLD4 (single 4-element structure to all lanes)";
if index_align<1:0> == '11' then UNDEFINED;
ebytes = 4;  index = UInt(index_align<3>);
inc = if index_align<2> == '0' then 1 else 2;
alignment = if index_align<1:0> == '00' then 1 else 4 << UInt(index_align<1:0>);
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  d4 = d3 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d4 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *VLD4 (single 4-element structure to one lane)*.

Assembler Symbols

<c> For encoding A1, A2 and A3: see *Standard assembler syntax fields*. This encoding must be unconditional.

For encoding T1, T2 and T3: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<size> Is the data size, encoded in “size”:

size	<size>
00	8
01	16
10	32

<list> Is a list containing the 64-bit names of the four SIMD&FP registers holding the element.

The list must be one of:

{ <Dd>[<index>], <Dd+1>[<index>], <Dd+2>[<index>], <Dd+3>[<index>] }
Single-spaced registers, encoded as "spacing" = 0.

{ <Dd>[<index>], <Dd+2>[<index>], <Dd+4>[<index>], <Dd+6>[<index>] }
Double-spaced registers, encoded as "spacing" = 1. Not permitted when <size> == 8.

The encoding of "spacing" depends on <size>:

<size> == 16
"spacing" is encoded in the "index_align<1>" field.

<size> == 32
"spacing" is encoded in the "index_align<2>" field.

The register <Dd> is encoded in the "D:Vd" field.

The permitted values and encoding of <index> depend on <size>:

<size> == 8
<index> is in the range 0 to 7, encoded in the "index_align<3:1>" field.

<size> == 16
<index> is in the range 0 to 3, encoded in the "index_align<3:2>" field.

<size> == 32
<index> is 0 or 1, encoded in the "index_align<3>" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

<align> Is the optional alignment.

Whenever <align> is omitted, the standard alignment is used, see *Unaligned data access*, and the encoding depends on <size>:

<size> == 8
Encoded in the "index_align<0>" field as 0.

<size> == 16
Encoded in the "index_align<0>" field as 0.

<size> == 32
Encoded in the "index_align<1:0>" field as 0b00.

Whenever <align> is present, the permitted values and encoding depend on <size>:

<size> == 8
<align> is 32, meaning 32-bit alignment, encoded in the "index_align<0>" field as 1.

<size> == 16
<align> is 64, meaning 64-bit alignment, encoded in the "index_align<0>" field as 1.

<size> == 32
<align> can be 64 or 128. 64-bit alignment is encoded in the "index_align<1:0>" field as 0b01, and 128-bit alignment is encoded in the "index_align<1:0>" field as 0b10.

: is the preferred separator before the <align> value, but the alignment can be specified as @<align>, see *Advanced SIMD addressing mode*.

<Rm> Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see *Advanced SIMD addressing mode*.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    address = R[n];  iswrite = FALSE;
    - = AArch32.CheckAlignment(address, alignment, AccType_VEC, iswrite);
    Elem[D[d], index] = MemU[address,ebytes];
    Elem[D[d2],index] = MemU[address+ebytes,ebytes];
    Elem[D[d3],index] = MemU[address+2*ebytes,ebytes];
    Elem[D[d4],index] = MemU[address+3*ebytes,ebytes];
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 4*ebytes;
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

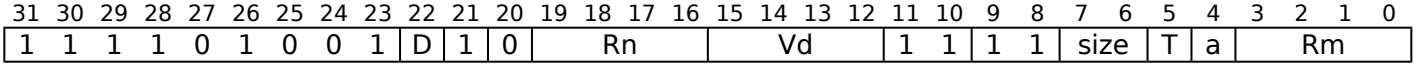
VLD4 (single 4-element structure to all lanes)

Load single 4-element structure and replicate to all lanes of four registers loads one 4-element structure from memory into all lanes of four registers. For details of the addressing mode see *Advanced SIMD addressing mode*.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	1	D	1	0	Rn				Vd				1	1	1	1	size		T	a	Rm			

Offset (Rm == 1111)

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}],<Rm>
```

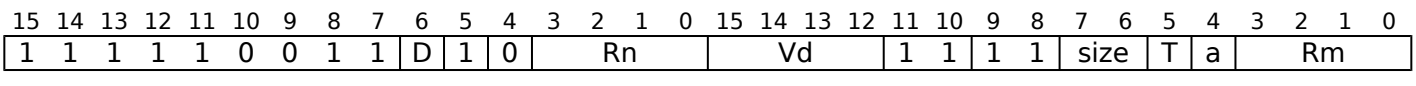
```
if size == '11' && a == '0' then UNDEFINED;
if size == '11' then
    ebytes = 4;  alignment = 16;
else
    ebytes = 1 << UInt(size);
    if size == '10' then
        alignment = if a == '0' then 1 else 8;
    else
        alignment = if a == '0' then 1 else 4*ebytes;
inc = if T == '0' then 1 else 2;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  d4 = d3 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d4 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T1



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	1	D	1	0	Rn				Vd				1	1	1	1	size		T	a	Rm			

Offset (Rm == 1111)

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

```
if size == '11' && a == '0' then UNDEFINED;
if size == '11' then
    ebytes = 4;  alignment = 16;
else
    ebytes = 1 << UInt(size);
    if size == '10' then
        alignment = if a == '0' then 1 else 8;
    else
        alignment = if a == '0' then 1 else 4*ebytes;
inc = if T == '0' then 1 else 2;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  d4 = d3 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d4 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *VLD4 (single 4-element structure to all lanes)*.

Assembler Symbols

<c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<size> Is the data size, encoded in “size”:

size	<size>
00	8
01	16
1x	32

<list> Is a list containing the 64-bit names of four SIMD&FP registers.

The list must be one of:

{ <Dd>[], <Dd+1>[], <Dd+2>[], <Dd+3>[] }

Single-spaced registers, encoded in the "T" field as 0.

{ <Dd>[], <Dd+2>[], <Dd+4>[], <Dd+6>[] }

Double-spaced registers, encoded in the "T" field as 1.

The register <Dd> is encoded in the "D:Vd" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

<align> Is the optional alignment.

Whenever <align> is omitted, the standard alignment is used, see *Unaligned data access*, and is encoded in the "a" field as 0.

Whenever <align> is present, the permitted values and encoding depend on <size>:

<size> == 8

<align> is 32, meaning 32-bit alignment, encoded in the "a" field as 1.

<size> == 16

<align> is 64, meaning 64-bit alignment, encoded in the "a" field as 1.

<size> == 32

<align> can be 64 or 128. 64-bit alignment is encoded in the "a:size<0>" field as 0b10, and 128-bit alignment is encoded in the "a:size<0>" field as 0b11.

: is the preferred separator before the <align> value, but the alignment can be specified as @<align>, see *Advanced SIMD addressing mode*.

<Rm> Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see *Advanced SIMD addressing mode*.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    address = R[n];  iswrite = FALSE;
    - = AArch32.CheckAlignment(address, alignment, AccType_VEC, iswrite);
    D[d] = Replicate(MemU[address,ebytes]);
    D[d2] = Replicate(MemU[address+ebytes,ebytes]);
    D[d3] = Replicate(MemU[address+2*ebytes,ebytes]);
    D[d4] = Replicate(MemU[address+3*ebytes,ebytes]);
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 4*ebytes;
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

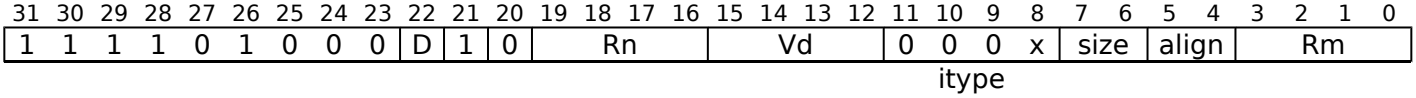
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VLD4 (multiple 4-element structures)

Load multiple 4-element structures to four registers loads multiple 4-element structures from memory into four registers, with de-interleaving. For more information, see *Element and structure load/store instructions*. Every element of each register is loaded. For details of the addressing mode see *Advanced SIMD addressing mode*.
Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



Offset (Rm == 1111)

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

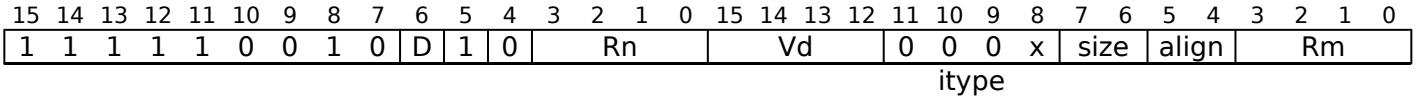
```
case itype of
    when '0000'
        inc = 1;
    when '0001'
        inc = 2;
    otherwise
        SEE "Related encodings";
if size == '11' then UNDEFINED;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  d4 = d3 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d4 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

T1



Offset (Rm == 1111)

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

```
case itype of
    when '0000'
        inc = 1;
    when '0001'
        inc = 2;
    otherwise
        SEE "Related encodings";
if size == '11' then UNDEFINED;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  d4 = d3 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d4 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- One or more of the SIMD and floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *VLD4 (multiple 4-element structures)*.

Related encodings: See *Advanced SIMD element or structure load/store* for the T32 instruction set, or *Advanced SIMD element or structure load/store* for the A32 instruction set.

Assembler Symbols

<c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<size> Is the data size, encoded in “size”:

size	<size>
00	8
01	16
10	32
11	RESERVED

<list> Is a list containing the 64-bit names of the SIMD&FP registers.

The list must be one of:

{ <Dd>, <Dd+1>, <Dd+2>, <Dd+3> }

Single-spaced registers, encoded in the "itype" field as 0b0000.

{ <Dd>, <Dd+2>, <Dd+4>, <Dd+6> }

Double-spaced registers, encoded in the "itype" field as 0b0001.

The register <Dd> is encoded in the "D:Vd" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

<align> Is the optional alignment.

Whenever <align> is omitted, the standard alignment is used, see *Unaligned data access*, and is encoded in the "align" field as 0b00.

Whenever <align> is present, the permitted values are:

64
64-bit alignment, encoded in the "align" field as 0b01.

128
128-bit alignment, encoded in the "align" field as 0b10.

256
256-bit alignment, encoded in the "align" field as 0b11.

: is the preferred separator before the <align> value, but the alignment can be specified as @<align>, see *Advanced SIMD addressing mode*.

<Rm> Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see *Advanced SIMD addressing mode*.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    address = R[n];  iswrite = FALSE;
    - = AArch32.CheckAlignment(address, alignment, AccType_VEC, iswrite);
    for e = 0 to elements-1
        Elem[D[d], e] = MemU[address,ebytes];
        Elem[D[d2],e] = MemU[address+ebytes,ebytes];
        Elem[D[d3],e] = MemU[address+2*ebytes,ebytes];
        Elem[D[d4],e] = MemU[address+3*ebytes,ebytes];
        address = address + 4*ebytes;
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 32;
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

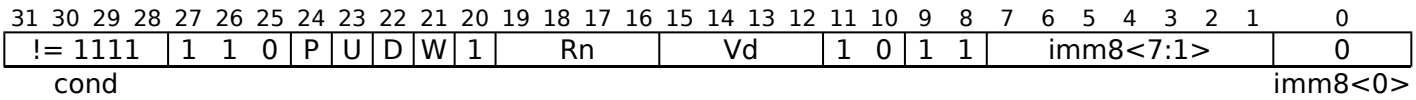
VLDM, VLDMDB, VLDMIA

Load Multiple SIMD&FP registers loads multiple registers from consecutive locations in the Advanced SIMD and floating-point register file using an address from a general-purpose register.
Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

This instruction is used by the alias VPOP.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2) .

A1



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

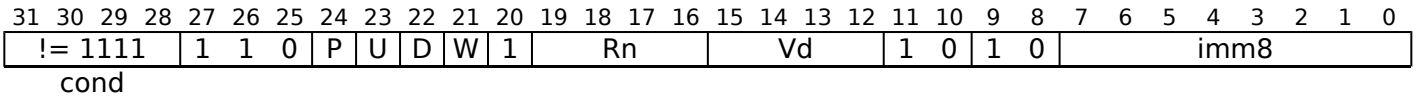
If `regs == 0`, then one of the following behaviors must occur:

- 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



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

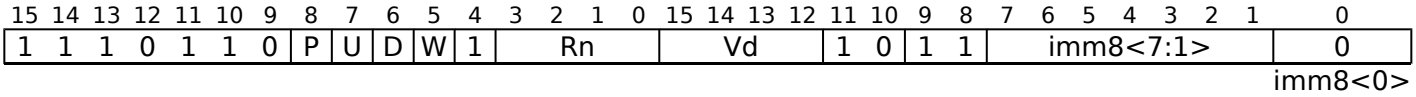
If `regs == 0`, then one of the following behaviors must occur:

- 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



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

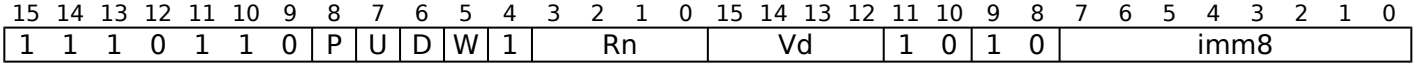
If `regs == 0`, then one of the following behaviors must occur:

- 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



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();  CheckVFPEnabled(TRUE);
    address = if add then R[n] else R[n]-imm32;
    for r = 0 to regs-1
        if single_regs then
            S[d+r] = MemA[address,4];  address = address+4;
        else
            word1 = MemA[address,4];  word2 = MemA[address+4,4];  address = address+8;
            // Combine the word-aligned words in the correct order for current endianness.
            D[d+r] = if BigEndian(AccType_ATOMIC) then word1:word2 else word2:word1;
    if wback then R[n] = if add then R[n]+imm32 else R[n]-imm32;
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

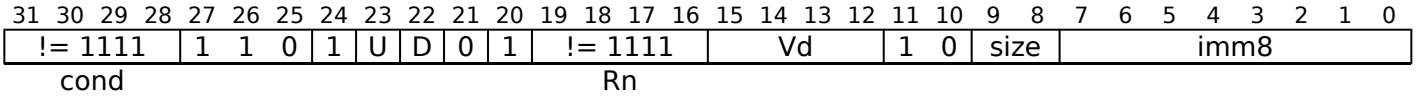
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VLDR (immediate)

Load SIMD&FP register (immediate) loads a single register from the Advanced SIMD and floating-point register file, using an address from a general-purpose register, with an optional offset.
Depending on settings in the *CPACR*, *NSACR*, *HCPTR*, and *FPEXC* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



Half-precision scalar (size == 01)
(FEAT_FP16)

```
VLDR{<c>}{<q>}.16 <Sd>, [<Rn> {, #{+/-}<imm>}]
```

Single-precision scalar (size == 10)

```
VLDR{<c>}{<q>}{.32} <Sd>, [<Rn> {, #{+/-}<imm>}]
```

Double-precision scalar (size == 11)

```
VLDR{<c>}{<q>}{.64} <Dd>, [<Rn> {, #{+/-}<imm>}]
```

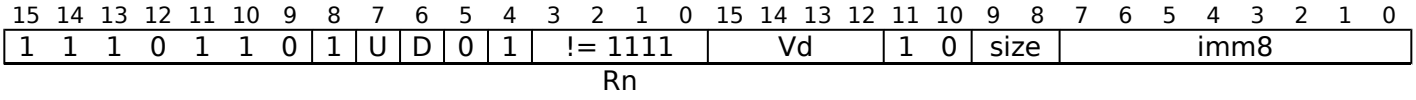
```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
esize = 8 << UInt(size);  add = (U == '1');
imm32 = if esize == 16 then ZeroExtend(imm8:'0', 32) else ZeroExtend(imm8:'00', 32);
case size of
    when '01' d = UInt(Vd:D);
    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



Half-precision scalar (size == 01)
(FEAT_FP16)

```
VLDR{<c>}{<q>}.16 <Sd>, [<Rn> {, #{+/-}<imm>}]
```

Single-precision scalar (size == 10)

```
VLDR{<c>}{<q>}{.32} <Sd>, [<Rn> {, #{+/-}<imm>}]
```

Double-precision scalar (size == 11)

```
VLDR{<c>}{<q>}{.64} <Dd>, [<Rn> {, #{+/-}<imm>}]
```

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock()  then UNPREDICTABLE;
esize = 8 << UInt(size);  add = (U == '1');
imm32 = if esize == 16 then ZeroExtend(imm8:'0', 32) else ZeroExtend(imm8:'00', 32);
case size of
    when '01' d = UInt(Vd:D);
    when '10' d = UInt(Vd:D);
    when '11' d = UInt(D:Vd);
n = UInt(Rn);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
.64	Is an optional data size specifier for 64-bit memory accesses that can be used in the assembler source code, but is otherwise ignored.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
.32	Is an optional data size specifier for 32-bit memory accesses that can be used in the assembler source code, but is otherwise ignored.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
<Rn>	Is the general-purpose base register, encoded in the "Rn" field.
+/-	Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<imm>	For the single-precision scalar or double-precision scalar variants: is the optional unsigned immediate byte offset, a multiple of 4, in the range 0 to 1020, defaulting to 0, and encoded in the "imm8" field as <imm>/4. For the half-precision scalar variant: is the optional unsigned immediate byte offset, a multiple of 2, in the range 0 to 510, defaulting to 0, and encoded in the "imm8" field as <imm>/2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    base = if n == 15 then Align(PC,4) else R[n];
    address = if add then (base + imm32) else (base - imm32);
    case esize of
        when 16
            S[d] = Zeros(16) : MemA[address,2];
        when 32
            S[d] = MemA[address,4];
        when 64
            word1 = MemA[address,4];  word2 = MemA[address+4,4];
            // Combine the word-aligned words in the correct order for current endianness.
            D[d] = if BigEndian(AccType_ATOMIC) then word1:word2 else word2:word1;
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

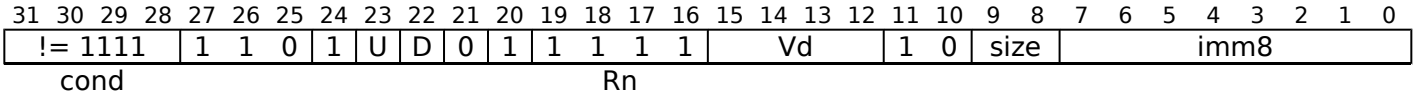
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VLDR (literal)

Load SIMD&FP register (literal) loads a single register from the Advanced SIMD and floating-point register file, using an address from the PC value and an immediate offset.
Depending on settings in the *CPACR*, *NSACR*, *HCPTR*, and *FPEXC* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



Half-precision scalar (size == 01)
(FEAT_FP16)

```
VLDR{<c>}{<q>}.16 <Sd>, <label>

VLDR{<c>}{<q>}.16 <Sd>, [PC, #{+/-}<imm>]
```

Single-precision scalar (size == 10)

```
VLDR{<c>}{<q>}{.32} <Sd>, <label>

VLDR{<c>}{<q>}{.32} <Sd>, [PC, #{+/-}<imm>]
```

Double-precision scalar (size == 11)

```
VLDR{<c>}{<q>}{.64} <Dd>, <label>

VLDR{<c>}{<q>}{.64} <Dd>, [PC, #{+/-}<imm>]
```

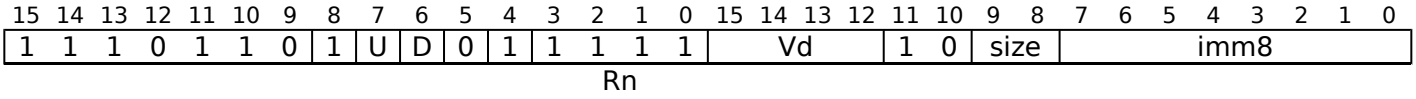
```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
esize = 8 << UInt(size);  add = (U == '1');
imm32 = if esize == 16 then ZeroExtend(imm8:'0', 32) else ZeroExtend(imm8:'00', 32);
case size of
    when '01' d = UInt(Vd:D);
    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



Half-precision scalar (size == 01)
(FEAT_FP16)

```
VLDR{<c>}{<q>}.16 <Sd>, <label>

VLDR{<c>}{<q>}.16 <Sd>, [PC, #{+/-}<imm>]
```

Single-precision scalar (size == 10)

```
VLDR{<c>}{<q>}{.32} <Sd>, <label>

VLDR{<c>}{<q>}{.32} <Sd>, [PC, #{+/-}<imm>]
```

Double-precision scalar (size == 11)

```
VLDR{<c>}{<q>}{.64} <Dd>, <label>

VLDR{<c>}{<q>}{.64} <Dd>, [PC, #{+/-}<imm>]
```

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock()  then UNPREDICTABLE;
esize = 8 << UInt(size);  add = (U == '1');
imm32 = if esize == 16 then ZeroExtend(imm8:'0', 32) else ZeroExtend(imm8:'00', 32);
case size of
    when '01' d = UInt(Vd:D);
    when '10' d = UInt(Vd:D);
    when '11' d = UInt(D:Vd);
n = UInt(Rn);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
.64	Is an optional data size specifier for 64-bit memory accesses that can be used in the assembler source code, but is otherwise ignored.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
.32	Is an optional data size specifier for 32-bit memory accesses that can be used in the assembler source code, but is otherwise ignored.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
<label>	The label of the literal data item to be loaded. For the single-precision scalar or double-precision scalar variants: the assembler calculates the required value of the offset from the Align(PC, 4) value of the instruction to this label. Permitted values are multiples of 4 in the range -1020 to 1020. For the half-precision scalar variant: the assembler calculates the required value of the offset from the Align(PC, 4) value of the instruction to this label. Permitted values are multiples of 2 in the range -510 to 510. If the offset is zero or positive, imm32 is equal to the offset and add == TRUE. If the offset is negative, imm32 is equal to minus the offset and add == FALSE.
+/-	Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<imm> For the single-precision scalar or double-precision scalar variants: is the optional unsigned immediate byte offset, a multiple of 4, in the range 0 to 1020, defaulting to 0, and encoded in the "imm8" field as <imm>/4.

For the half-precision scalar variant: is the optional unsigned immediate byte offset, a multiple of 2, in the range 0 to 510, defaulting to 0, and encoded in the "imm8" field as <imm>/2.

The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see *Use of labels in UAL instruction syntax*.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    base = if n == 15 then Align(PC,4) else R[n];
    address = if add then (base + imm32) else (base - imm32);
    case esize of
        when 16
            S[d] = Zeros(16) : MemA[address,2];
        when 32
            S[d] = MemA[address,4];
        when 64
            word1 = MemA[address,4];  word2 = MemA[address+4,4];
            // Combine the word-aligned words in the correct order for current endianness.
            D[d] = if BigEndian(AccType_ATOMIC) then word1:word2 else word2:word1;
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VMAX (floating-point)

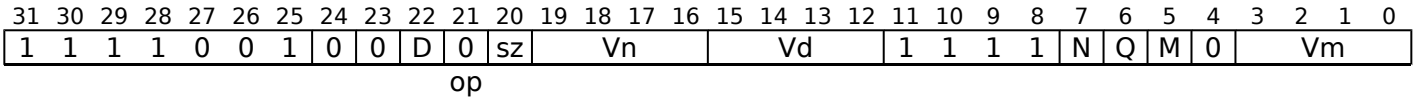
Vector Maximum compares corresponding elements in two vectors, and copies the larger of each pair into the corresponding element in the destination vector.

The operand vector elements are floating-point numbers.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



64-bit SIMD vector (Q == 0)

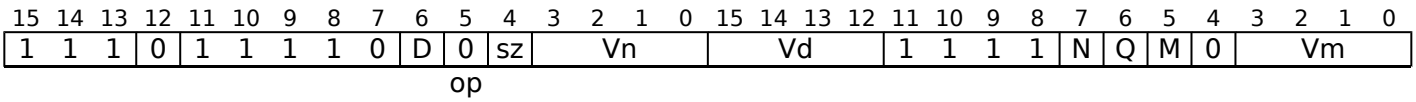
```
VMAX{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VMAX{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
maximum = (op == '0');
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1



64-bit SIMD vector (Q == 0)

```
VMAX{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VMAX{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
maximum = (op == '0');
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `sz == '1' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.

- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the vectors, encoded in “sz”:

sz	<dt>
0	F32
1	F16

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Floating-point maximum and minimum

- max(+0.0, -0.0) = +0.0
- If any input is a NaN, the corresponding result element is the default NaN.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[D[n+r],e,esize];  op2 = Elem[D[m+r],e,esize];
            if maximum then
                Elem[D[d+r],e,esize] = FPMax(op1, op2, StandardFPSCRValue());
            else
                Elem[D[d+r],e,esize] = FPMin(op1, op2, StandardFPSCRValue());
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VMAX (integer)

Vector Maximum compares corresponding elements in two vectors, and copies the larger of each pair into the corresponding element in the destination vector.

The operand vector elements can be any one of:

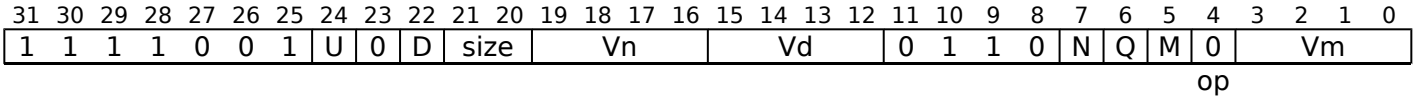
- 8-bit, 16-bit, or 32-bit signed integers.
- 8-bit, 16-bit, or 32-bit unsigned integers.

The result vector elements are the same size as the operand vector elements.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



64-bit SIMD vector (Q == 0)

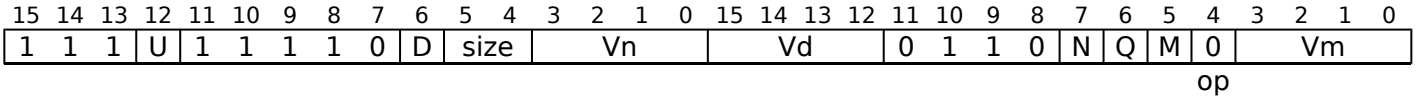
```
VMAX{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VMAX{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
maximum = (op == '0');  unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1



64-bit SIMD vector (Q == 0)

```
VMAX{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VMAX{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
maximum = (op == '0');  unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>	For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional. For encoding T1: see *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the operands, encoded in “U:size”:

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	00	U8
1	01	U16
1	10	U32

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Int(Elem[D[n+r],e,esize], unsigned);
            op2 = Int(Elem[D[m+r],e,esize], unsigned);
            result = if maximum then Max(op1,op2) else Min(op1,op2);
            Elem[D[d+r],e,esize] = result<esize-1:0>;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

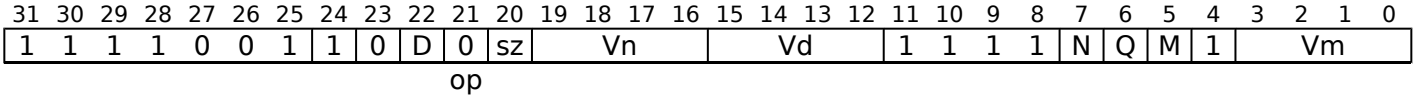
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VMAXNM

This instruction determines the floating-point maximum number.
It handles NaNs in consistence with the IEEE754-2008 specification. It returns the numerical operand when one operand is numerical and the other is a quiet NaN, but otherwise the result is identical to floating-point VMAX.
This instruction is not conditional.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2) .

A1



64-bit SIMD vector (Q == 0)

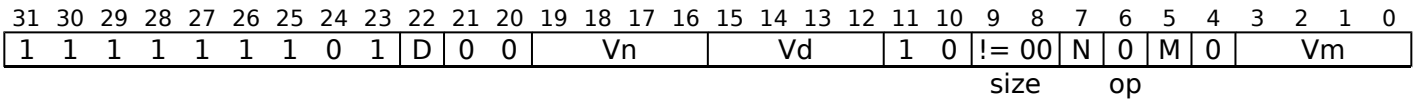
```
VMAXNM{<q>}.<dt> <Dd>, <Dn>, <Dm>
```

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



Half-precision scalar (size == 01)
(FEAT_FP16)

```
VMAXNM{<q>}.F16 <Sd>, <Sn>, <Sm> // (Cannot be conditional)
```

Single-precision scalar (size == 10)

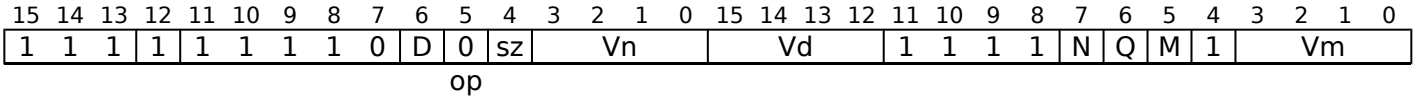
```
VMAXNM{<q>}.F32 <Sd>, <Sn>, <Sm> // (Cannot be conditional)
```

Double-precision scalar (size == 11)

```
VMAXNM{<q>}.F64 <Dd>, <Dn>, <Dm> // (Cannot be conditional)
```

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
advsimd = FALSE;
maximum = (op == '0');
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>
```

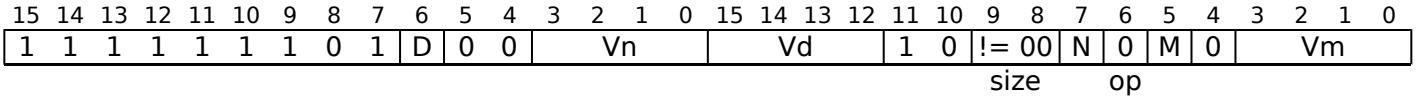
```
if InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
maximum = (op == '0');
advsimd = TRUE;
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd); 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)
(FEAT_FP16)

```
VMAXNM{<q>}.F16 <Sd>, <Sn>, <Sm> // (Not permitted in IT block)
```

Single-precision scalar (size == 10)

```
VMAXNM{<q>}.F32 <Sd>, <Sn>, <Sm> // (Not permitted in IT block)
```

Double-precision scalar (size == 11)

```
VMAXNM{<q>}.F64 <Dd>, <Dn>, <Dm> // (Not permitted in IT block)
```

```
if InITBlock() then UNPREDICTABLE;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
advsimd = FALSE;
maximum = (op == '0');
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();  CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
if advsimd then             // Advanced SIMD instruction
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[D[n+r], e, esize]; op2 = Elem[D[m+r], e, esize];
            if maximum then
                Elem[D[d+r], e, esize] = FPMaxNum(op1, op2, StandardFPSCRValue());
            else
                Elem[D[d+r], e, esize] = FPMinNum(op1, op2, StandardFPSCRValue());
else                        // VFP instruction
    case esize of
        when 16
            if maximum then
                S[d] = Zeros(16) : FPMaxNum(S[n]<15:0>, S[m]<15:0>, FPSCR[]);
            else
                S[d] = Zeros(16) : FPMinNum(S[n]<15:0>, S[m]<15:0>, FPSCR[]);
        when 32
            if maximum then
                S[d] = FPMaxNum(S[n], S[m], FPSCR[]);
            else
                S[d] = FPMinNum(S[n], S[m], FPSCR[]);
        when 64
            if maximum then
                D[d] = FPMaxNum(D[n], D[m], FPSCR[]);
            else
                D[d] = FPMinNum(D[n], D[m], FPSCR[]);
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VMIN (floating-point)

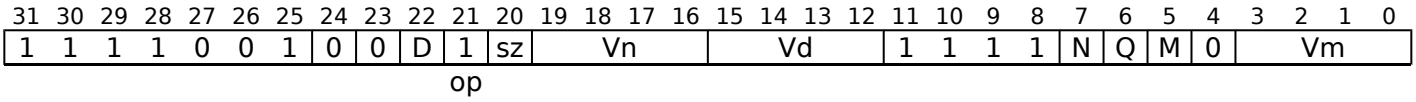
Vector Minimum compares corresponding elements in two vectors, and copies the smaller of each pair into the corresponding element in the destination vector.

The operand vector elements are floating-point numbers.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



64-bit SIMD vector (Q == 0)

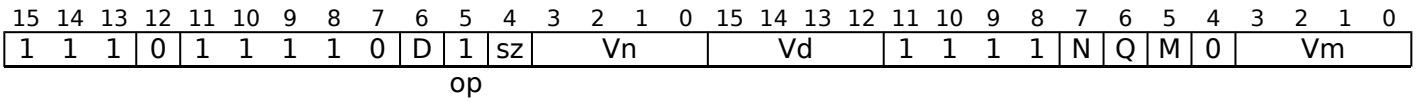
```
VMIN{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VMIN{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
maximum = (op == '0');
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1



64-bit SIMD vector (Q == 0)

```
VMIN{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VMIN{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
maximum = (op == '0');
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `sz == '1' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.

- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the vectors, encoded in “sz”:

sz	<dt>
0	F32
1	F16

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Floating-point minimum

- min(+0.0, -0.0) = -0.0
- If any input is a NaN, the corresponding result element is the default NaN.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[D[n+r],e,esize];  op2 = Elem[D[m+r],e,esize];
            if maximum then
                Elem[D[d+r],e,esize] = FPMax(op1, op2, StandardFPSCRValue());
            else
                Elem[D[d+r],e,esize] = FPMin(op1, op2, StandardFPSCRValue());
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VMIN (integer)

Vector Minimum compares corresponding elements in two vectors, and copies the smaller of each pair into the corresponding element in the destination vector.

The operand vector elements can be any one of:

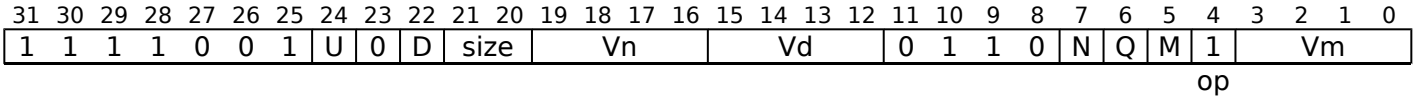
- 8-bit, 16-bit, or 32-bit signed integers.
- 8-bit, 16-bit, or 32-bit unsigned integers.

The result vector elements are the same size as the operand vector elements.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



64-bit SIMD vector (Q == 0)

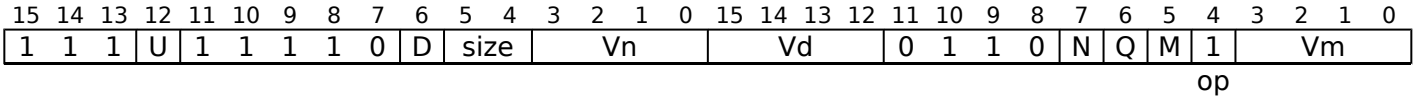
```
VMIN{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VMIN{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
maximum = (op == '0');  unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1



64-bit SIMD vector (Q == 0)

```
VMIN{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VMIN{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
maximum = (op == '0');  unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>	For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional. For encoding T1: see *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the operands, encoded in “U:size”:

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	00	U8
1	01	U16
1	10	U32

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Int(Elem[D[n+r],e,esize], unsigned);
            op2 = Int(Elem[D[m+r],e,esize], unsigned);
            result = if maximum then Max(op1,op2) else Min(op1,op2);
            Elem[D[d+r],e,esize] = result<esize-1:0>;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

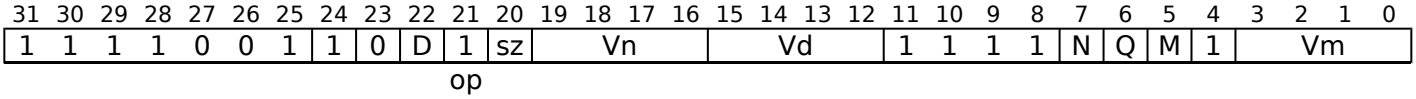
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VMINNM

This instruction determines the floating point minimum number.
It handles NaNs in consistence with the IEEE754-2008 specification. It returns the numerical operand when one operand is numerical and the other is a quiet NaN, but otherwise the result is identical to floating-point VMIN.
This instruction is not conditional.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2) .

A1



64-bit SIMD vector (Q == 0)

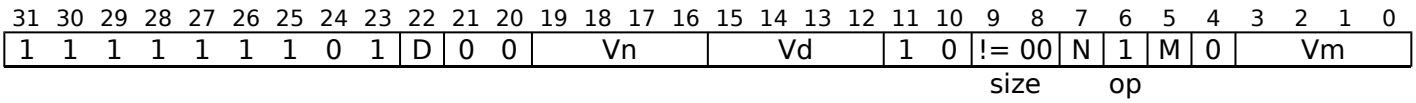
```
VMINNM{<q>}.<dt> <Dd>, <Dn>, <Dm>
```

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



Half-precision scalar (size == 01)
(FEAT_FP16)

```
VMINNM{<q>}.F16 <Sd>, <Sn>, <Sm> // (Cannot be conditional)
```

Single-precision scalar (size == 10)

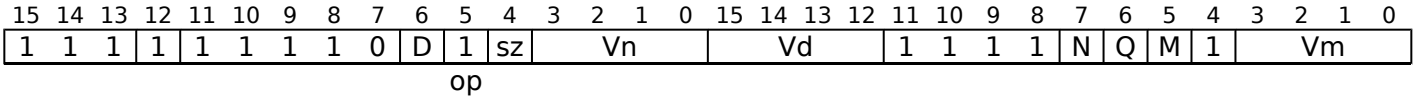
```
VMINNM{<q>}.F32 <Sd>, <Sn>, <Sm> // (Cannot be conditional)
```

Double-precision scalar (size == 11)

```
VMINNM{<q>}.F64 <Dd>, <Dn>, <Dm> // (Cannot be conditional)
```

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
advsimd = FALSE;
maximum = (op == '0');
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>
```

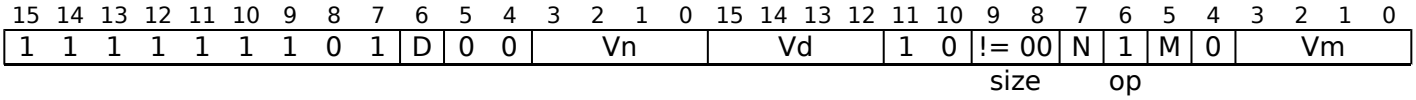
```
if InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
maximum = (op == '0');
advsimd = TRUE;
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd); 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)
(FEAT_FP16)

```
VMINNM{<q>}.F16 <Sd>, <Sn>, <Sm> // (Not permitted in IT block)
```

Single-precision scalar (size == 10)

```
VMINNM{<q>}.F32 <Sd>, <Sn>, <Sm> // (Not permitted in IT block)
```

Double-precision scalar (size == 11)

```
VMINNM{<q>}.F64 <Dd>, <Dn>, <Dm> // (Not permitted in IT block)
```

```
if InITBlock() then UNPREDICTABLE;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
advsimd = FALSE;
maximum = (op == '0');
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();  CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
if advsimd then             // Advanced SIMD instruction
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[D[n+r], e, esize]; op2 = Elem[D[m+r], e, esize];
            if maximum then
                Elem[D[d+r], e, esize] = FPMaxNum(op1, op2, StandardFPSCRValue());
            else
                Elem[D[d+r], e, esize] = FPMinNum(op1, op2, StandardFPSCRValue());
else                        // VFP instruction
    case esize of
        when 16
            if maximum then
                S[d] = Zeros(16) : FPMaxNum(S[n]<15:0>, S[m]<15:0>, FPSCR[]);
            else
                S[d] = Zeros(16) : FPMinNum(S[n]<15:0>, S[m]<15:0>, FPSCR[]);
        when 32
            if maximum then
                S[d] = FPMaxNum(S[n], S[m], FPSCR[]);
            else
                S[d] = FPMinNum(S[n], S[m], FPSCR[]);
        when 64
            if maximum then
                D[d] = FPMaxNum(D[n], D[m], FPSCR[]);
            else
                D[d] = FPMinNum(D[n], D[m], FPSCR[]);
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

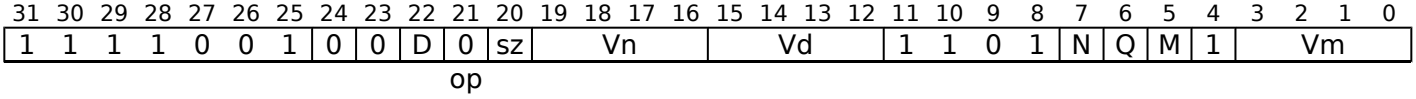
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VMLA (floating-point)

Vector Multiply Accumulate multiplies corresponding elements in two vectors, and accumulates the results into the elements of the destination vector.
Depending on settings in the *CPACR*, *NSACR*, *HCPTR*, and *FPEXC* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2) .

A1



64-bit SIMD vector (Q == 0)

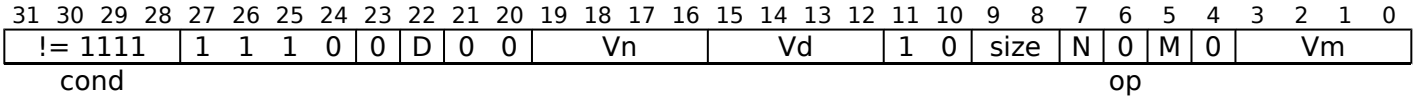
```
VMLA{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm>
```

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



Half-precision scalar (size == 01)
(FEAT_FP16)

```
VMLA{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>
```

Single-precision scalar (size == 10)

```
VMLA{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>
```

Double-precision scalar (size == 11)

```
VMLA{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>
```

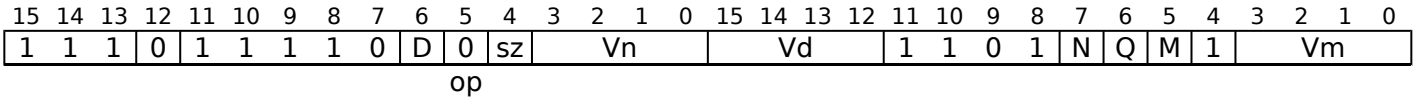
```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
advsimd = FALSE; add = (op == '0');
case size of
    when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    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>
```

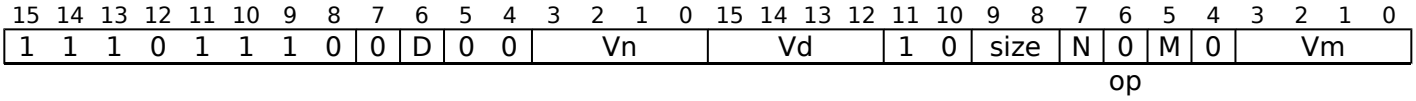
```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
advsimd = TRUE;  add = (op == '0');
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
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)
(FEAT_FP16)

```
VMLA{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>
```

Single-precision scalar (size == 10)

```
VMLA{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>
```

Double-precision scalar (size == 11)

```
VMLA{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>
```

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock()  then UNPREDICTABLE;
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
advsimd = FALSE; add = (op == '0');
case size of
    when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.
For encoding A2, T1 and T2: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the vectors, encoded in “sz”:

sz	<dt>
0	F32
1	F16

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

<Sn> Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.

<Sm> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
    if advsimd then  // Advanced SIMD instruction
        for r = 0 to regs-1
            for e = 0 to elements-1
                product = FPMul(Elem[D[n+r],e,esize], Elem[D[m+r],e,esize], StandardFPSCRValue());
                addend = if add then product else FPNeg(product);
                Elem[D[d+r],e,esize] = 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 FPNeg(FPMul(S[n]<15:0>
                S[d] = Zeros(16) : FPAdd(S[d]<15:0>, addend16, FPSCR[]);
            when 32
                addend32 = if add then FPMul(S[n], S[m], FPSCR[]) else FPNeg(FPMul(S[n], S[m], FPSCR[]));
                S[d] = FPAdd(S[d], addend32, FPSCR[]);
            when 64
                addend64 = if add then FPMul(D[n], D[m], FPSCR[]) else FPNeg(FPMul(D[n], D[m], FPSCR[]));
                D[d] = FPAdd(D[d], addend64, FPSCR[]);
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

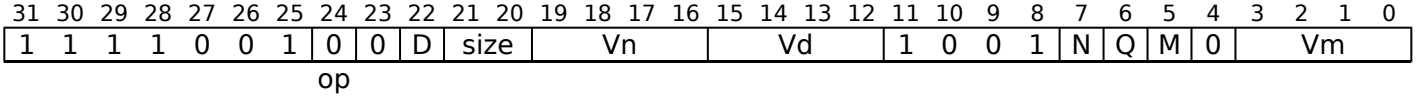
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VMLA (integer)

Vector Multiply Accumulate multiplies corresponding elements in two vectors, and adds the products to the corresponding elements of the destination vector.
Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



64-bit SIMD vector (Q == 0)

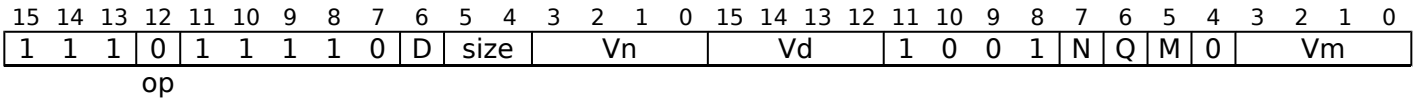
```
VMLA{<c>}{<q>}.<type><size> <Dd>, <Dn>, <Dm> // (Encoding T1/A1, encoded as Q = 0)
```

128-bit SIMD vector (Q == 1)

```
VMLA{<c>}{<q>}.<type><size> <Qd>, <Qn>, <Qm> // (Encoding T1/A1, encoded as Q = 1)
```

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
add = (op == '0');  long_destination = FALSE;
unsigned = FALSE;  // "Don't care" value: TRUE produces same functionality
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1



64-bit SIMD vector (Q == 0)

```
VMLA{<c>}{<q>}.<type><size> <Dd>, <Dn>, <Dm> // (Encoding T1/A1, encoded as Q = 0)
```

128-bit SIMD vector (Q == 1)

```
VMLA{<c>}{<q>}.<type><size> <Qd>, <Qn>, <Qm> // (Encoding T1/A1, encoded as Q = 1)
```

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
add = (op == '0');  long_destination = FALSE;
unsigned = FALSE;  // "Don't care" value: TRUE produces same functionality
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>	For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional. For encoding T1: see *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<type>	The data type for the elements of the operands. It must be one of:

S
Optional in encoding T1/A1. Encoded as U = 0 in encoding T2/A2.

U
Optional in encoding T1/A1. Encoded as U = 1 in encoding T2/A2.

I
Available only in encoding T1/A1.

<size> The data size for the elements of the operands. It must be one of:

8
Encoded as size = 0b00.

16
Encoded as size = 0b01.

32
Encoded as size = 0b10.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            product = Int(Elem[Din[n+r],e,esize],unsigned) * Int(Elem[Din[m+r],e,esize],unsigned);
            addend = if add then product else -product;
            if long_destination then
                Elem[Q[d>>1],e,2*esize] = Elem[Qin[d>>1],e,2*esize] + addend;
            else
                Elem[D[d+r],e,esize] = Elem[Din[d+r],e,esize] + addend;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VMLA (by scalar)

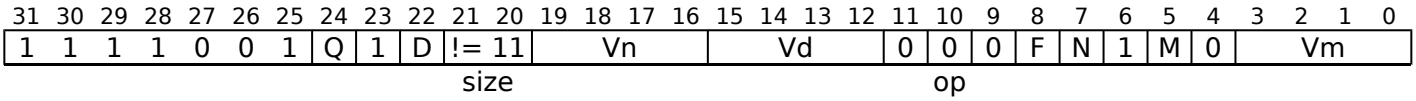
Vector Multiply Accumulate multiplies elements of a vector by a scalar, and adds the products to corresponding elements of the destination vector.

For more information about scalars see *Advanced SIMD scalars*.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



64-bit SIMD vector (Q == 0)

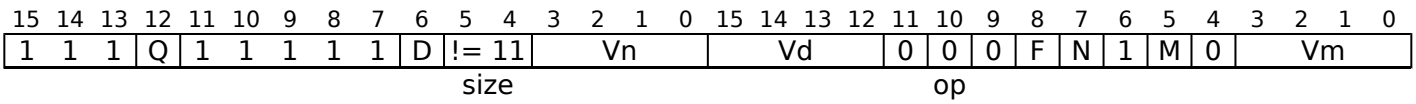
```
VMLA{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm[x]>
```

128-bit SIMD vector (Q == 1)

```
VMLA{<c>}{<q>}.<dt> <Qd>, <Qn>, <Dm[x]>
```

```
if size == '11' then SEE "Related encodings";
if size == '00' || (F == '1' && size == '01' && !HaveFP16Ext()) then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
unsigned = FALSE;  // "Don't care" value: TRUE produces same functionality
add = (op == '0');  floating_point = (F == '1');  long_destination = FALSE;
d = UInt(D:Vd);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```

T1



64-bit SIMD vector (Q == 0)

```
VMLA{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm[x]>
```

128-bit SIMD vector (Q == 1)

```
VMLA{<c>}{<q>}.<dt> <Qd>, <Qn>, <Dm[x]>
```

```
if size == '11' then SEE "Related encodings";
if size == '00' || (F == '1' && size == '01' && !HaveFP16Ext()) then UNDEFINED;
if F == '1' && size == '01' && InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
unsigned = FALSE;  // "Don't care" value: TRUE produces same functionality
add = (op == '0');  floating_point = (F == '1');  long_destination = FALSE;
d = UInt(D:Vd);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```

CONSTRAINED UNPREDICTABLE behavior

If `F == '1' && size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Related encodings: See *Advanced SIMD data-processing* for the T32 instruction set, or *Advanced SIMD data-processing* for the A32 instruction set.

Assembler Symbols

<c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<dt> Is the data type for the scalar and the elements of the operand vector, encoded in “F:size”:

F	size	<dt>
0	01	I16
0	10	I32
1	01	F16
1	10	F32

<Qd> Is the 128-bit name of the SIMD&FP register holding the accumulate vector, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Dd> Is the 64-bit name of the SIMD&FP register holding the accumulate vector, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm[x]> Is the 64-bit name of the second SIMD&FP source register holding the scalar. If <dt> is I16 or F16, Dm is restricted to D0-D7. Dm is encoded in "Vm<2:0>", and x is encoded in "M:Vm<3>". If <dt> is I32 or F32, Dm is restricted to D0-D15. Dm is encoded in "Vm", and x is encoded in "M".

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    op2 = Elem[Din[m],index,esize];  op2val = Int(op2, unsigned);
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[Din[n+r],e,esize];  op1val = Int(op1, unsigned);
            if floating_point then
                fp_addend = if add then FPMul(op1,op2,StandardFPSCRValue()) else FPNeg(FPMul(op1,op2,Star
                Elem[D[d+r],e,esize] = FPAdd(Elem[Din[d+r],e,esize], fp_addend, StandardFPSCRValue());
            else
                addend = if add then op1val*op2val else -op1val*op2val;
                if long_destination then
                    Elem[Q[d>>1],e,2*esize] = Elem[Qin[d>>1],e,2*esize] + addend;
                else
                    Elem[D[d+r],e,esize] = Elem[Din[d+r],e,esize] + addend;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check and is operating only on integer vector elements, then the following apply:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

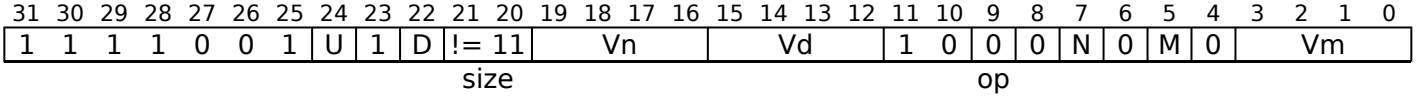
VMLAL (integer)

Vector Multiply Accumulate Long multiplies corresponding elements in two vectors, and add the products to the corresponding element of the destination vector. The destination vector element is twice as long as the elements that are multiplied.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

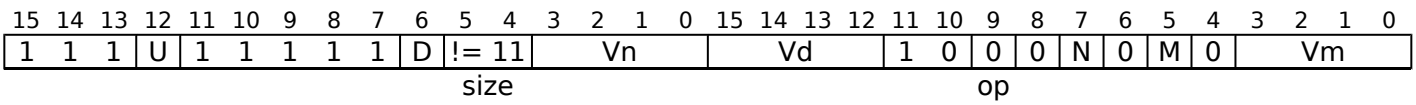
A1



A1

```
VMLAL{<c>}{<q>}.<type><size> <Qd>, <Dn>, <Dm> // (Encoding T2/A2)

if size == '11' then SEE "Related encodings";
if Vd<0> == '1' then UNDEFINED;
add = (op == '0');  long_destination = TRUE;  unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = 1;
```

T1



T1

```
VMLAL{<c>}{<q>}.<type><size> <Qd>, <Dn>, <Dm> // (Encoding T2/A2)

if size == '11' then SEE "Related encodings";
if Vd<0> == '1' then UNDEFINED;
add = (op == '0');  long_destination = TRUE;  unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = 1;
```

Related encodings: See *Advanced SIMD data-processing* for the T32 instruction set, or *Advanced SIMD data-processing* for the A32 instruction set.

Assembler Symbols

<c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<type> The data type for the elements of the operands. It must be one of:

S
Optional in encoding T1/A1. Encoded as U = 0 in encoding T2/A2.

U
Optional in encoding T1/A1. Encoded as U = 1 in encoding T2/A2.

I
Available only in encoding T1/A1.

<size> The data size for the elements of the operands. It must be one of:

8
Encoded as size = 0b00.

16
Encoded as size = 0b01.

32
Encoded as size = 0b10.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            product = Int(Elem[Din[n+r],e,esize],unsigned) * Int(Elem[Din[m+r],e,esize],unsigned);
            addend = if add then product else -product;
            if long_destination then
                Elem[Q[d>>1],e,2*esize] = Elem[Qin[d>>1],e,2*esize] + addend;
            else
                Elem[D[d+r],e,esize] = Elem[Din[d+r],e,esize] + addend;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VMLAL (by scalar)

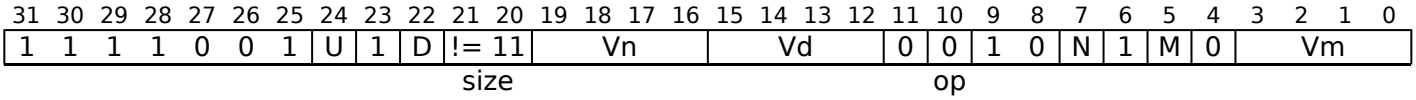
Vector Multiply Accumulate Long multiplies elements of a vector by a scalar, and adds the products to corresponding elements of the destination vector. The destination vector elements are twice as long as the elements that are multiplied.

For more information about scalars see *Advanced SIMD scalars*.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

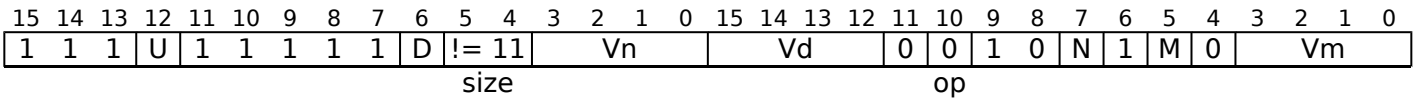


A1

```
VMLAL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm[x]>
```

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
unsigned = (U == '1');  add = (op == '0');  floating_point = FALSE;  long_destination = TRUE;
d = UInt(D:Vd);  n = UInt(N:Vn);  regs = 1;
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```

T1



T1

```
VMLAL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm[x]>
```

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
unsigned = (U == '1');  add = (op == '0');  floating_point = FALSE;  long_destination = TRUE;
d = UInt(D:Vd);  n = UInt(N:Vn);  regs = 1;
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```

Related encodings: See *Advanced SIMD data-processing* for the T32 instruction set, or *Advanced SIMD data-processing* for the A32 instruction set.

Assembler Symbols

<c>	For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional. For encoding T1: see *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<dt>	Is the data type for the scalar and the elements of the operand vector, encoded in “U:size”:

U	size	<dt>
0	01	S16
0	10	S32
1	01	U16
1	10	U32

<Qd> Is the 128-bit name of the SIMD&FP register holding the accumulate vector, encoded in the "D:Vd" field as <Qd>*2.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm[x]> Is the 64-bit name of the second SIMD&FP source register holding the scalar. If <dt> is S16 or U16, Dm is restricted to D0-D7. Dm is encoded in "Vm<2:0>", and x is encoded in "M:Vm<3>". If <dt> is S32 or U32, Dm is restricted to D0-D15. Dm is encoded in "Vm", and x is encoded in "M".

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    op2 = Elem[Din[m],index,esize];  op2val = Int(op2, unsigned);
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[Din[n+r],e,esize];  op1val = Int(op1, unsigned);
            if floating_point then
                fp_addend = if add then FPMul(op1,op2,StandardFPSCRValue()) else FPNeg(FPMul(op1,op2,Star
                Elem[D[d+r],e,esize] = FPAdd(Elem[Din[d+r],e,esize], fp_addend, StandardFPSCRValue());
            else
                addend = if add then op1val*op2val else -op1val*op2val;
                if long_destination then
                    Elem[Q[d>>1],e,2*esize] = Elem[Qin[d>>1],e,2*esize] + addend;
                else
                    Elem[D[d+r],e,esize] = Elem[Din[d+r],e,esize] + addend;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VMLS (floating-point)

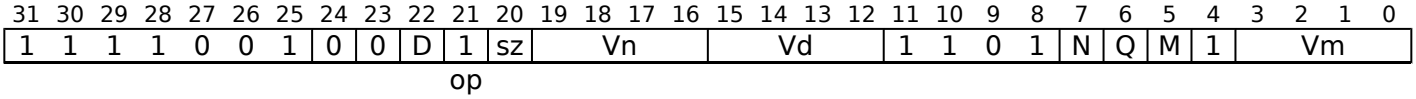
Vector Multiply Subtract multiplies corresponding elements in two vectors, subtracts the products from corresponding elements of the destination vector, and places the results in the destination vector.

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



64-bit SIMD vector (Q == 0)

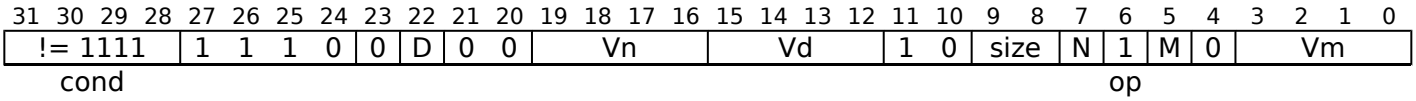
```
VMLS{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm>
```

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



Half-precision scalar (size == 01)
(FEAT_FP16)

```
VMLS{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>
```

Single-precision scalar (size == 10)

```
VMLS{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>
```

Double-precision scalar (size == 11)

```
VMLS{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>
```

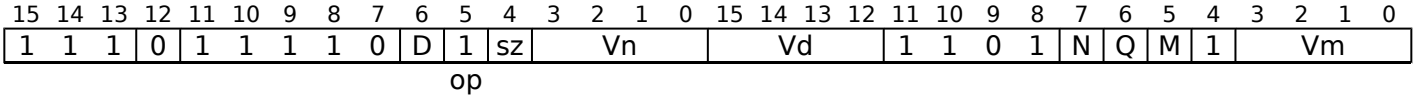
```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
advsimd = FALSE; add = (op == '0');
case size of
    when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1



64-bit SIMD vector (Q == 0)

```
VMLS{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VMLS{<c>}{<q>}.<dt> <Qd>, <Qn>, <Qm>
```

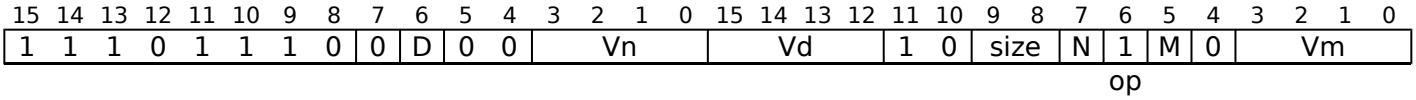
```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
advsimd = TRUE;  add = (op == '0');
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `sz == '1' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T2



Half-precision scalar (size == 01)
(FEAT_FP16)

```
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();  CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
    if advsimd then  // Advanced SIMD instruction
        for r = 0 to regs-1
            for e = 0 to elements-1
                product = FPMul(Elem[D[n+r],e,esize], Elem[D[m+r],e,esize], StandardFPSCRValue());
                addend = if add then product else FPNeg(product);
                Elem[D[d+r],e,esize] = 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 FPNeg(FPMul(S[n]<15:0>
                S[d] = Zeros(16) : FPAdd(S[d]<15:0>, addend16, FPSCR[]);
            when 32
                addend32 = if add then FPMul(S[n], S[m], FPSCR[]) else FPNeg(FPMul(S[n], S[m], FPSCR[]));
                S[d] = FPAdd(S[d], addend32, FPSCR[]);
            when 64
                addend64 = if add then FPMul(D[n], D[m], FPSCR[]) else FPNeg(FPMul(D[n], D[m], FPSCR[]));
                D[d] = FPAdd(D[d], addend64, FPSCR[]);
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

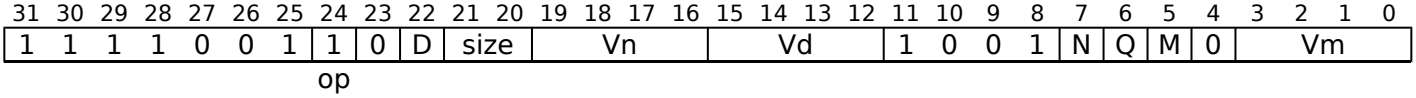
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VMLS (integer)

Vector Multiply Subtract multiplies corresponding elements in two vectors, and subtracts the products from the corresponding elements of the destination vector.
Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



64-bit SIMD vector (Q == 0)

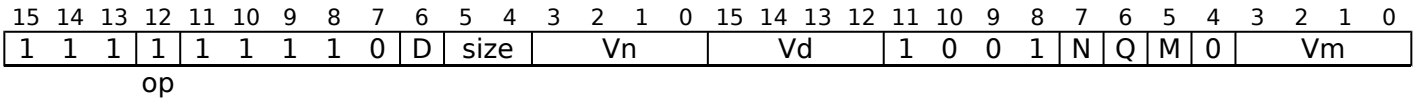
```
VMLS{<c>}{<q>}.<type><size> <Dd>, <Dn>, <Dm> // (Encoding T1/A1, encoded as Q = 0)
```

128-bit SIMD vector (Q == 1)

```
VMLS{<c>}{<q>}.<type><size> <Qd>, <Qn>, <Qm> // (Encoding T1/A1, encoded as Q = 1)
```

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
add = (op == '0');  long_destination = FALSE;
unsigned = FALSE;  // "Don't care" value: TRUE produces same functionality
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1



64-bit SIMD vector (Q == 0)

```
VMLS{<c>}{<q>}.<type><size> <Dd>, <Dn>, <Dm> // (Encoding T1/A1, encoded as Q = 0)
```

128-bit SIMD vector (Q == 1)

```
VMLS{<c>}{<q>}.<type><size> <Qd>, <Qn>, <Qm> // (Encoding T1/A1, encoded as Q = 1)
```

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
add = (op == '0');  long_destination = FALSE;
unsigned = FALSE;  // "Don't care" value: TRUE produces same functionality
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>	For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional. For encoding T1: see *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<type>	The data type for the elements of the operands. It must be one of:

S
Optional in encoding T1/A1. Encoded as U = 0 in encoding T2/A2.

U
Optional in encoding T1/A1. Encoded as U = 1 in encoding T2/A2.

I
Available only in encoding T1/A1.

<size> The data size for the elements of the operands. It must be one of:

8
Encoded as size = 0b00.

16
Encoded as size = 0b01.

32
Encoded as size = 0b10.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            product = Int(Elem[Din[n+r],e,esize],unsigned) * Int(Elem[Din[m+r],e,esize],unsigned);
            addend = if add then product else -product;
            if long_destination then
                Elem[Q[d>>1],e,2*esize] = Elem[Qin[d>>1],e,2*esize] + addend;
            else
                Elem[D[d+r],e,esize] = Elem[Din[d+r],e,esize] + addend;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VMLS (by scalar)

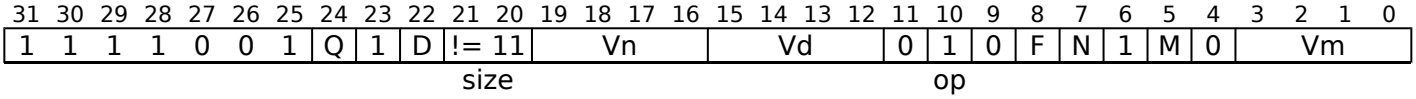
Vector Multiply Subtract multiplies elements of a vector by a scalar, and either subtracts the products from corresponding elements of the destination vector.

For more information about scalars see *Advanced SIMD scalars*.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



64-bit SIMD vector (Q == 0)

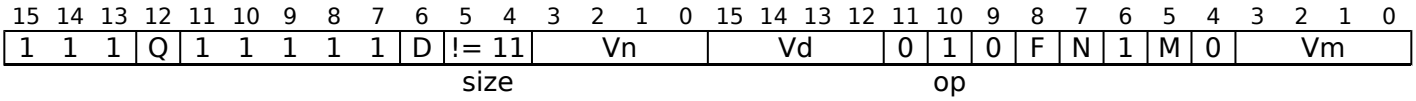
```
VMLS{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm[x]>
```

128-bit SIMD vector (Q == 1)

```
VMLS{<c>}{<q>}.<dt> <Qd>, <Qn>, <Dm[x]>
```

```
if size == '11' then SEE "Related encodings";
if size == '00' || (F == '1' && size == '01' && !HaveFP16Ext()) then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
unsigned = FALSE;  // "Don't care" value: TRUE produces same functionality
add = (op == '0');  floating_point = (F == '1');  long_destination = FALSE;
d = UInt(D:Vd);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```

T1



64-bit SIMD vector (Q == 0)

```
VMLS{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm[x]>
```

128-bit SIMD vector (Q == 1)

```
VMLS{<c>}{<q>}.<dt> <Qd>, <Qn>, <Dm[x]>
```

```
if size == '11' then SEE "Related encodings";
if size == '00' || (F == '1' && size == '01' && !HaveFP16Ext()) then UNDEFINED;
if F == '1' && size == '01' && InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
unsigned = FALSE;  // "Don't care" value: TRUE produces same functionality
add = (op == '0');  floating_point = (F == '1');  long_destination = FALSE;
d = UInt(D:Vd);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```

CONSTRAINED UNPREDICTABLE behavior

If `F == '1' && size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Related encodings: See *Advanced SIMD data-processing* for the T32 instruction set, or *Advanced SIMD data-processing* for the A32 instruction set.

Assembler Symbols

<c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<dt> Is the data type for the scalar and the elements of the operand vector, encoded in “F:size”:

F	size	<dt>
0	01	I16
0	10	I32
1	01	F16
1	10	F32

<Qd> Is the 128-bit name of the SIMD&FP register holding the accumulate vector, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Dd> Is the 64-bit name of the SIMD&FP register holding the accumulate vector, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm[x]> Is the 64-bit name of the second SIMD&FP source register holding the scalar. If <dt> is I16 or F16, Dm is restricted to D0-D7. Dm is encoded in "Vm<2:0>", and x is encoded in "M:Vm<3>". If <dt> is I32 or F32, Dm is restricted to D0-D15. Dm is encoded in "Vm", and x is encoded in "M".

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    op2 = Elem[Din[m],index,esize];  op2val = Int(op2, unsigned);
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[Din[n+r],e,esize];  op1val = Int(op1, unsigned);
            if floating_point then
                fp_addend = if add then FPMul(op1,op2,StandardFPSCRValue()) else FPNeg(FPMul(op1,op2,Star
                Elem[D[d+r],e,esize] = FPAdd(Elem[Din[d+r],e,esize], fp_addend, StandardFPSCRValue());
            else
                addend = if add then op1val*op2val else -op1val*op2val;
                if long_destination then
                    Elem[Q[d>>1],e,2*esize] = Elem[Qin[d>>1],e,2*esize] + addend;
                else
                    Elem[D[d+r],e,esize] = Elem[Din[d+r],e,esize] + addend;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check and is operating only on integer vector elements, then the following apply:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

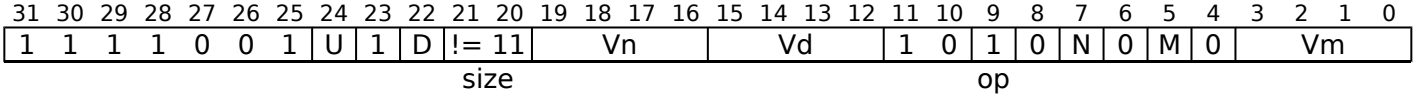
VMLSL (integer)

Vector Multiply Subtract Long multiplies corresponding elements in two vectors, and subtract the products from the corresponding elements of the destination vector. The destination vector element is twice as long as the elements that are multiplied.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

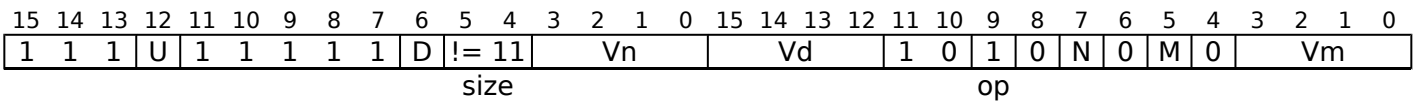
A1



A1

```
VMLSL{<c>}{<q>}.<type><size> <Qd>, <Dn>, <Dm> // (Encoding T2/A2)

if size == '11' then SEE "Related encodings";
if Vd<0> == '1' then UNDEFINED;
add = (op == '0');  long_destination = TRUE;  unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = 1;
```

T1



T1

```
VMLSL{<c>}{<q>}.<type><size> <Qd>, <Dn>, <Dm> // (Encoding T2/A2)

if size == '11' then SEE "Related encodings";
if Vd<0> == '1' then UNDEFINED;
add = (op == '0');  long_destination = TRUE;  unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = 1;
```

Related encodings: See *Advanced SIMD data-processing* for the T32 instruction set, or *Advanced SIMD data-processing* for the A32 instruction set.

Assembler Symbols

<c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<type> The data type for the elements of the operands. It must be one of:

S
Optional in encoding T1/A1. Encoded as U = 0 in encoding T2/A2.

U
Optional in encoding T1/A1. Encoded as U = 1 in encoding T2/A2.

I
Available only in encoding T1/A1.

<size> The data size for the elements of the operands. It must be one of:

8
Encoded as size = 0b00.

16
Encoded as size = 0b01.

32
Encoded as size = 0b10.

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            product = Int(Elem[Din[n+r],e,esize],unsigned) * Int(Elem[Din[m+r],e,esize],unsigned);
            addend = if add then product else -product;
            if long_destination then
                Elem[Q[d>>1],e,2*esize] = Elem[Qin[d>>1],e,2*esize] + addend;
            else
                Elem[D[d+r],e,esize] = Elem[Din[d+r],e,esize] + addend;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VMLSL (by scalar)

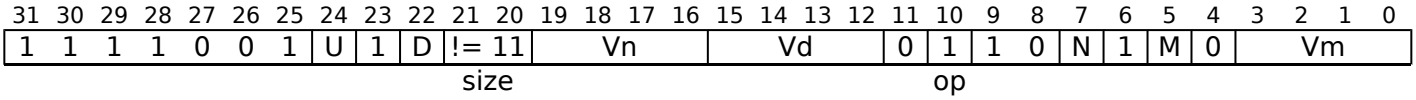
Vector Multiply Subtract Long multiplies elements of a vector by a scalar, and subtracts the products from corresponding elements of the destination vector. The destination vector elements are twice as long as the elements that are multiplied.

For more information about scalars see *Advanced SIMD scalars*.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

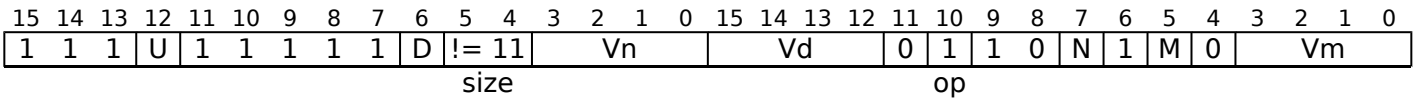


A1

```
VMLSL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm[x]>
```

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
unsigned = (U == '1');  add = (op == '0');  floating_point = FALSE;  long_destination = TRUE;
d = UInt(D:Vd);  n = UInt(N:Vn);  regs = 1;
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```

T1



T1

```
VMLSL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm[x]>
```

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
unsigned = (U == '1');  add = (op == '0');  floating_point = FALSE;  long_destination = TRUE;
d = UInt(D:Vd);  n = UInt(N:Vn);  regs = 1;
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```

Related encodings: See *Advanced SIMD data-processing* for the T32 instruction set, or *Advanced SIMD data-processing* for the A32 instruction set.

Assembler Symbols

<c>	For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional. For encoding T1: see *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<dt>	Is the data type for the scalar and the elements of the operand vector, encoded in “U:size”:

U	size	<dt>
0	01	S16
0	10	S32
1	01	U16
1	10	U32

<Qd> Is the 128-bit name of the SIMD&FP register holding the accumulate vector, encoded in the "D:Vd" field as <Qd>*2.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm[x]> Is the 64-bit name of the second SIMD&FP source register holding the scalar. If <dt> is S16 or U16, Dm is restricted to D0-D7. Dm is encoded in "Vm<2:0>", and x is encoded in "M:Vm<3>". If <dt> is S32 or U32, Dm is restricted to D0-D15. Dm is encoded in "Vm", and x is encoded in "M".

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    op2 = Elem[Din[m],index,esize];  op2val = Int(op2, unsigned);
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[Din[n+r],e,esize];  op1val = Int(op1, unsigned);
            if floating_point then
                fp_addend = if add then FPMul(op1,op2,StandardFPSCRValue()) else FPNeg(FPMul(op1,op2,Star
                Elem[D[d+r],e,esize] = FPAdd(Elem[Din[d+r],e,esize], fp_addend, StandardFPSCRValue());
            else
                addend = if add then op1val*op2val else -op1val*op2val;
                if long_destination then
                    Elem[Q[d>>1],e,2*esize] = Elem[Qin[d>>1],e,2*esize] + addend;
                else
                    Elem[D[d+r],e,esize] = Elem[Din[d+r],e,esize] + addend;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

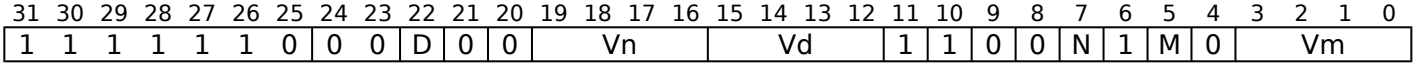
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VMMLA

BFloat16 floating-point matrix multiply-accumulate. This instruction multiplies the 2x4 matrix of BF16 values in the first 128-bit source vector by the 4x2 BF16 matrix in the second 128-bit source vector. The resulting 2x2 single-precision matrix product is then added destructively to the 2x2 single-precision matrix in the 128-bit destination vector. This is equivalent to performing a 4-way dot product per destination element. The instruction does not update the *FPSCR* exception status.
Arm expects that the VMMLA instruction will deliver a peak BF16 multiply throughput that is at least as high as can be achieved using two VDOT instructions, with a goal that it should have significantly higher throughput.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1
(FEAT_AA32BF16)

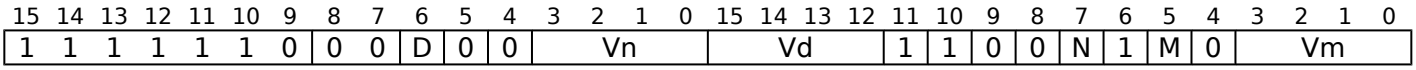


A1

```
VMMLA{<q>}.BF16 <Qd>, <Qn>, <Qm>
```

```
if !HaveAArch32BF16Ext() then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(M:Vm);
integer regs = 2;
```

T1
(FEAT_AA32BF16)



T1

```
VMMLA{<q>}.BF16 <Qd>, <Qn>, <Qm>
```

```
if InITBlock() then UNPREDICTABLE;
if !HaveAArch32BF16Ext() then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(M:Vm);
integer regs = 2;
```

Assembler Symbols

<q>	See *Standard assembler syntax fields*.
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qn>	Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
<Qm>	Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
CheckAdvSIMDEnabled();

bits(128) op1 = Q[n>>1];
bits(128) op2 = Q[m>>1];
bits(128) acc = Q[d>>1];

Q[d>>1] = BFMatMulAdd(acc, op1, op2);
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

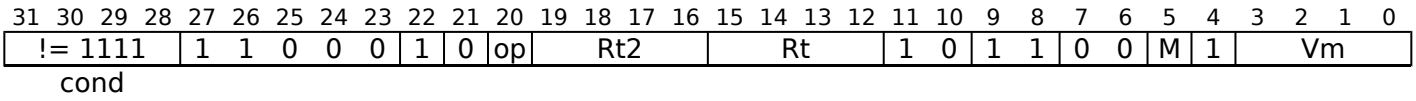
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VMOV (between two general-purpose registers and a doubleword floating-point register)

Copy two general-purpose registers to or from a SIMD&FP register copies two words from two general-purpose registers into a doubleword register in the Advanced SIMD and floating-point register file, or from a doubleword register in the Advanced SIMD and floating-point register file to two general-purpose registers.
Depending on settings in the *CPACR*, *NSACR*, *HCPTR*, and *FPEXC* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



From general-purpose registers (op == 0)

```
VMOV{<c>}{<q>} <Dm>, <Rt>, <Rt2>
```

To general-purpose registers (op == 1)

```
VMOV{<c>}{<q>} <Rt>, <Rt2>, <Dm>
```

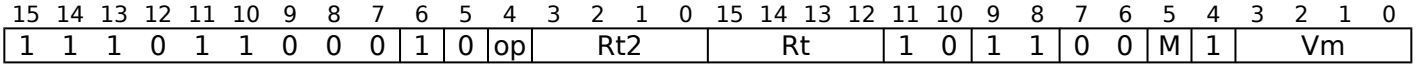
```
to_arm_registers = (op == '1');  t = UInt(Rt);  t2 = UInt(Rt2);  m = UInt(M:Vm);
if t == 15 || t2 == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
if to_arm_registers && t == t2 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `to_arm_registers && t == t2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The value in the destination register is UNKNOWN.

T1



From general-purpose registers (op == 0)

```
VMOV{<c>}{<q>} <Dm>, <Rt>, <Rt2>
```

To general-purpose registers (op == 1)

```
VMOV{<c>}{<q>} <Rt>, <Rt2>, <Dm>
```

```
to_arm_registers = (op == '1');  t = UInt(Rt);  t2 = UInt(Rt2);  m = UInt(M:Vm);
if t == 15 || t2 == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
if to_arm_registers && t == t2 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `to_arm_registers && t == t2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.

- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *VMOV (between two general-purpose registers and a doubleword floating-point register)*.

Assembler Symbols

<Dm>	Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "M:Vm" field.
<Rt2>	Is the second general-purpose register that <Dm>[63:32] will be transferred to or from, encoded in the "Rt2" field.
<Rt>	Is the first general-purpose register that <Dm>[31:0] will be transferred to or from, encoded in the "Rt" field.
<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    if to_arm_registers then
        R[t] = D[m]<31:0>;
        R[t2] = D[m]<63:32>;
    else
        D[m]<31:0> = R[t];
        D[m]<63:32> = R[t2];
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

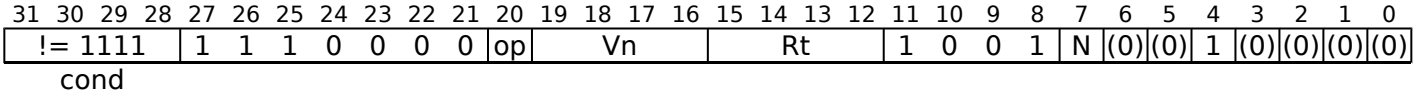
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VMOV (between general-purpose register and half-precision)

Copy 16 bits of a general-purpose register to or from a 32-bit SIMD&FP register. This instruction transfers the value held in the bottom 16 bits of a 32-bit SIMD&FP register to the bottom 16 bits of a general-purpose register, or the value held in the bottom 16 bits of a general-purpose register to the bottom 16 bits of a 32-bit SIMD&FP register.
Depending on settings in the *CPACR*, *NSACR*, *HCPTR*, and *FPEXC* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1
(FEAT_FP16)



From general-purpose register (op == 0)

```
VMOV{<c>}{<q>}.F16 <Sn>, <Rt>
```

To general-purpose register (op == 1)

```
VMOV{<c>}{<q>}.F16 <Rt>, <Sn>
```

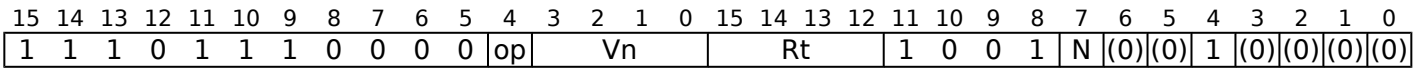
```
if !HaveFP16Ext() then UNDEFINED;
if cond != '1110' then UNPREDICTABLE;
to_arm_register = (op == '1');  t = UInt(Rt);  n = UInt(Vn:N);
if t == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

CONSTRAINED UNPREDICTABLE behavior

If `cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1
(FEAT_FP16)



From general-purpose register (op == 0)

```
VMOV{<c>}{<q>}.F16 <Sn>, <Rt>
```

To general-purpose register (op == 1)

```
VMOV{<c>}{<q>}.F16 <Rt>, <Sn>
```

```
if !HaveFP16Ext() then UNDEFINED;
if InITBlock() then UNPREDICTABLE;
to_arm_register = (op == '1');  t = UInt(Rt);  n = UInt(Vn:N);
if t == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<Rt>	Is the general-purpose register that <Sn> will be transferred to or from, encoded in the "Rt" field.
<Sn>	Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Vn:N" field.
<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    if to_arm_register then
        R[t] = Zeros(16) : S[n]<15:0>;
    else
        S[n] = Zeros(16) : R[t]<15:0>;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

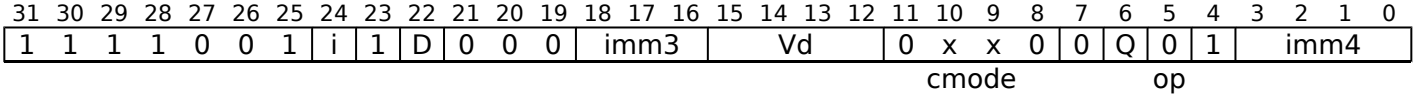
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VMOV (immediate)

Copy immediate value to a SIMD&FP register places an immediate constant into every element of the destination register.
Depending on settings in the *CPACR*, *NSACR*, *HCPTR*, and *FPEXC* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1 , A2 , A3 , A4 and A5) and T32 (T1 , T2 , T3 , T4 and T5) .

A1



64-bit SIMD vector (Q == 0)

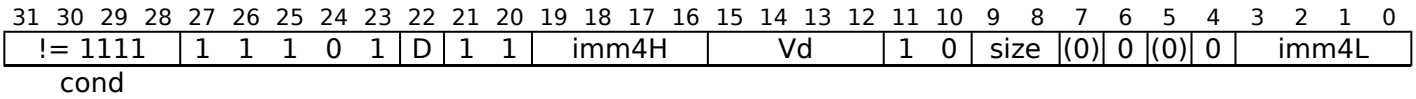
```
VMOV{<c>}{<q>}.I32 <Dd>, #<imm>
```

128-bit SIMD vector (Q == 1)

```
VMOV{<c>}{<q>}.I32 <Qd>, #<imm>
```

```
if op == '0' && cmode<0> == '1' && cmode<3:2> != '11' then SEE "VORR (immediate)";
if op == '1' && cmode != '1110' then SEE "Related encodings";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
single_register = FALSE;  advsimd = TRUE;  imm64 = AdvSIMDExpandImm(op, cmode, i:imm3:imm4);
d = UInt(D:Vd);  regs = if Q == '0' then 1 else 2;
```

A2



Half-precision scalar (size == 01)
(FEAT_FP16)

```
VMOV{<c>}{<q>}.F16 <Sd>, #<imm>
```

Single-precision scalar (size == 10)

```
VMOV{<c>}{<q>}.F32 <Sd>, #<imm>
```

Double-precision scalar (size == 11)

```
VMOV{<c>}{<q>}.F64 <Dd>, #<imm>
```

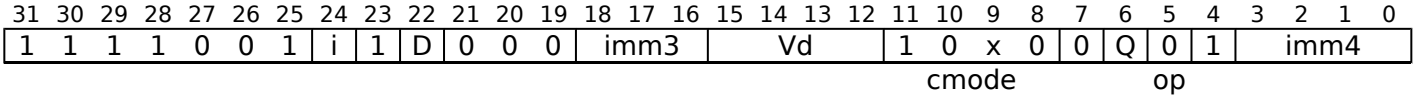
```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
single_register = (size != '11'); advsimd = FALSE;
bits(16) imm16;
bits(32) imm32;
bits(64) imm64;
case size of
    when '01' d = UInt(Vd:D);  imm16 = VFPExpandImm(imm4H:imm4L); imm32 = Zeros(16) : imm16;
    when '10' d = UInt(Vd:D);  imm32 = VFPExpandImm(imm4H:imm4L);
    when '11' d = UInt(D:Vd);  imm64 = VFPExpandImm(imm4H:imm4L);  regs = 1;
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

A3



64-bit SIMD vector (Q == 0)

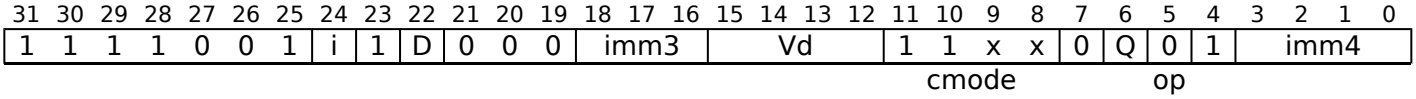
```
VMOV{<c>}{<q>}.I16 <Dd>, #<imm>
```

128-bit SIMD vector (Q == 1)

```
VMOV{<c>}{<q>}.I16 <Qd>, #<imm>
```

```
if op == '0' && cmode<0> == '1' && cmode<3:2> != '11' then SEE "VORR (immediate)";
if op == '1' && cmode != '1110' then SEE "Related encodings";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
single_register = FALSE;  advsimd = TRUE;  imm64 = AdvSIMDExpandImm(op, cmode, i:imm3:imm4);
d = UInt(D:Vd);  regs = if Q == '0' then 1 else 2;
```

A4



64-bit SIMD vector (Q == 0)

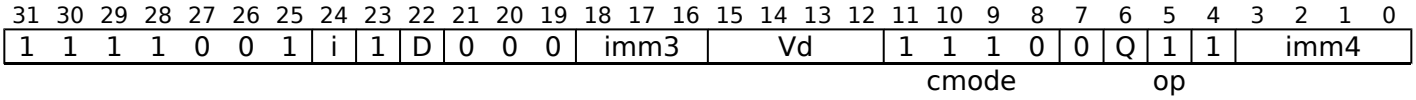
```
VMOV{<c>}{<q>}.<dt> <Dd>, #<imm>
```

128-bit SIMD vector (Q == 1)

```
VMOV{<c>}{<q>}.<dt> <Qd>, #<imm>
```

```
if op == '0' && cmode<0> == '1' && cmode<3:2> != '11' then SEE "VORR (immediate)";
if op == '1' && cmode != '1110' then SEE "Related encodings";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
single_register = FALSE;  advsimd = TRUE;  imm64 = AdvSIMDExpandImm(op, cmode, i:imm3:imm4);
d = UInt(D:Vd);  regs = if Q == '0' then 1 else 2;
```

A5



64-bit SIMD vector (Q == 0)

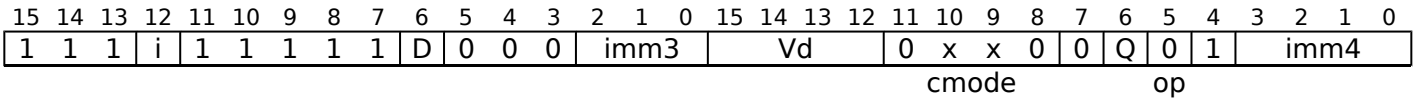
```
VMOV{<c>}{<q>}.I64 <Dd>, #<imm>
```

128-bit SIMD vector (Q == 1)

```
VMOV{<c>}{<q>}.I64 <Qd>, #<imm>
```

```
if op == '0' && cmode<0> == '1' && cmode<3:2> != '11' then SEE "VORR (immediate)";
if op == '1' && cmode != '1110' then SEE "Related encodings";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
single_register = FALSE;  advsimd = TRUE;  imm64 = AdvSIMDExpandImm(op, cmode, i:imm3:imm4);
d = UInt(D:Vd);  regs = if Q == '0' then 1 else 2;
```

T1



64-bit SIMD vector (Q == 0)

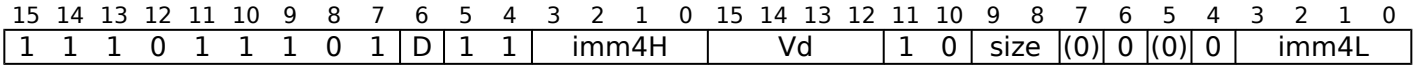
```
VMOV{<c>}{<q>}.I32 <Dd>, #<imm>
```

128-bit SIMD vector (Q == 1)

```
VMOV{<c>}{<q>}.I32 <Qd>, #<imm>
```

```
if op == '0' && cmode<0> == '1' && cmode<3:2> != '11' then SEE "VORR (immediate)";
if op == '1' && cmode != '1110' then SEE "Related encodings";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
single_register = FALSE;  advsimd = TRUE;  imm64 = AdvSIMDExpandImm(op, cmode, i:imm3:imm4);
d = UInt(D:Vd);  regs = if Q == '0' then 1 else 2;
```

T2



Half-precision scalar (size == 01)
(FEAT_FP16)

```
VMOV{<c>}{<q>}.F16 <Sd>, #<imm>
```

Single-precision scalar (size == 10)

```
VMOV{<c>}{<q>}.F32 <Sd>, #<imm>
```

Double-precision scalar (size == 11)

```
VMOV{<c>}{<q>}.F64 <Dd>, #<imm>
```

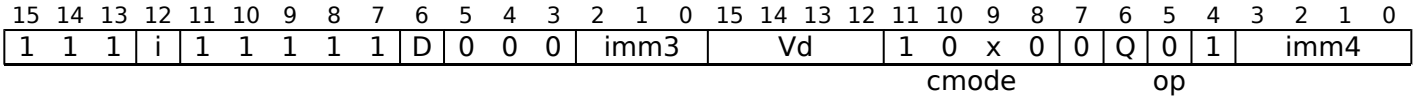
```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock()  then UNPREDICTABLE;
single_register = (size != '11'); advsimd = FALSE;
bits(16) imm16;
bits(32) imm32;
bits(64) imm64;
case size of
    when '01' d = UInt(Vd:D);  imm16 = VFPExpandImm(imm4H:imm4L); imm32 = Zeros(16) : imm16;
    when '10' d = UInt(Vd:D);  imm32 = VFPExpandImm(imm4H:imm4L);
    when '11' d = UInt(D:Vd);  imm64 = VFPExpandImm(imm4H:imm4L);  regs = 1;
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T3



64-bit SIMD vector (Q == 0)

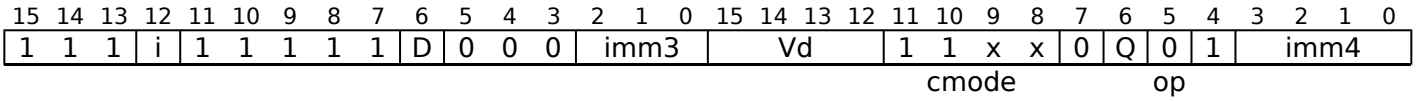
```
VMOV{<c>}{<q>}.I16 <Dd>, #<imm>
```

128-bit SIMD vector (Q == 1)

```
VMOV{<c>}{<q>}.I16 <Qd>, #<imm>
```

```
if op == '0' && cmode<0> == '1' && cmode<3:2> != '11' then SEE "VORR (immediate)";
if op == '1' && cmode != '1110' then SEE "Related encodings";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
single_register = FALSE;  advsimd = TRUE;  imm64 = AdvSIMDExpandImm(op, cmode, i:imm3:imm4);
d = UInt(D:Vd);  regs = if Q == '0' then 1 else 2;
```

T4



64-bit SIMD vector (Q == 0)

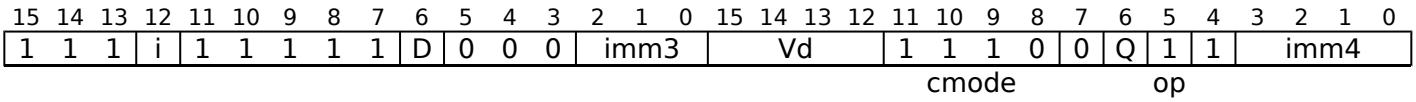
```
VMOV{<c>}{<q>}.<dt> <Dd>, #<imm>
```

128-bit SIMD vector (Q == 1)

```
VMOV{<c>}{<q>}.<dt> <Qd>, #<imm>
```

```
if op == '0' && cmode<0> == '1' && cmode<3:2> != '11' then SEE "VORR (immediate)";
if op == '1' && cmode != '1110' then SEE "Related encodings";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
single_register = FALSE;  advsimd = TRUE;  imm64 = AdvSIMDExpandImm(op, cmode, i:imm3:imm4);
d = UInt(D:Vd);  regs = if Q == '0' then 1 else 2;
```

T5



64-bit SIMD vector (Q == 0)

```
VMOV{<c>}{<q>}.I64 <Dd>, #<imm>
```

128-bit SIMD vector (Q == 1)

```
VMOV{<c>}{<q>}.I64 <Qd>, #<imm>
```

```
if op == '0' && cmode<0> == '1' && cmode<3:2> != '11' then SEE "VORR (immediate)";
if op == '1' && cmode != '1110' then SEE "Related encodings";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
single_register = FALSE;  advsimd = TRUE;  imm64 = AdvSIMDExpandImm(op, cmode, i:imm3:imm4);
d = UInt(D:Vd);  regs = if Q == '0' then 1 else 2;
```

Related encodings: See *Advanced SIMD one register and modified immediate* for the T32 instruction set, or *Advanced SIMD one register and modified immediate* for the A32 instruction set.

Assembler Symbols

<c> For encoding A1, A3, A4 and A5: see *Standard assembler syntax fields*. This encoding must be unconditional.

For encoding A2, T1, T2, T3, T4 and T5: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<dt> The data type, encoded in “cmode”:

cmode	<dt>
110x	I32
1110	I8
1111	F32

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

<imm> For encoding A1, A3, A4, A5, T1, T3, T4 and T5: is a constant of the specified type that is replicated to fill the destination register. For details of the range of constants available and the encoding of <imm>, see *Modified immediate constants in T32 and A32 Advanced SIMD instructions*.

For encoding A2 and T2: is a signed floating-point constant with 3-bit exponent and normalized 4 bits of precision, encoded in "imm4H:imm4L". For details of the range of constants available and the encoding of <imm>, see *Modified immediate constants in T32 and A32 floating-point instructions*.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
    if single_register then
        S[d] = imm32;
    else
        for r = 0 to regs-1
            D[d+r] = imm64;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

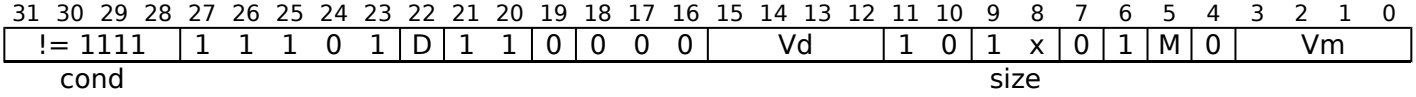
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VMOV (register)

Copy between FP registers copies the contents of one FP register to another.
Depending on settings in the *CPACR*, *NSACR*, *HCPTR*, and *FPEXC* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A2) and T32 (T2) .

A2



Single-precision scalar (size == 10)

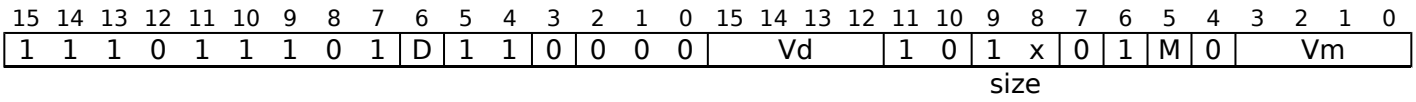
```
VMOV{<c>}{<q>}.F32 <Sd>, <Sm>
```

Double-precision scalar (size == 11)

```
VMOV{<c>}{<q>}.F64 <Dd>, <Dm>
```

```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
single_register = (size == '10');  advsimd = FALSE;
if single_register then
    d = UInt(Vd:D);  m = UInt(Vm:M);
else
    d = UInt(D:Vd);  m = UInt(M:Vm);  regs = 1;
```

T2



Single-precision scalar (size == 10)

```
VMOV{<c>}{<q>}.F32 <Sd>, <Sm>
```

Double-precision scalar (size == 11)

```
VMOV{<c>}{<q>}.F64 <Dd>, <Dm>
```

```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
single_register = (size == '10');  advsimd = FALSE;
if single_register then
    d = UInt(Vd:D);  m = UInt(Vm:M);
else
    d = UInt(D:Vd);  m = UInt(M:Vm);  regs = 1;
```

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
<Sm>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
    if single_register then
        S[d] = S[m];
    else
        for r = 0 to regs-1
            D[d+r] = D[m+r];
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VMOV (general-purpose register to scalar)

Copy a general-purpose register to a vector element copies a byte, halfword, or word from a general-purpose register into an Advanced SIMD scalar.

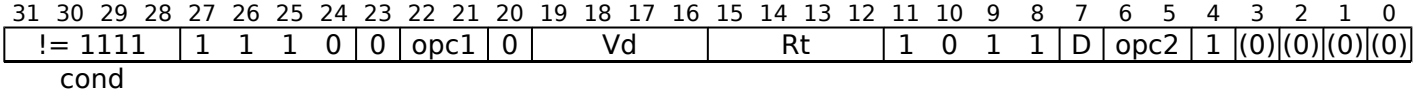
On a Floating-point-only system, this instruction transfers one word to the upper or lower half of a double-precision floating-point register from a general-purpose register. This is an identical operation to the Advanced SIMD single word transfer.

For more information about scalars see *Advanced SIMD scalars*.

Depending on settings in the *CPACR*, *NSACR*, *HCPTR*, and *FPEXC* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

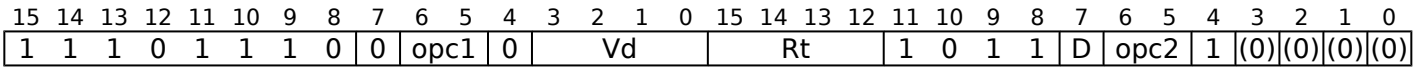


A1

```
VMOV{<c>}{<q>}{.<size>} <Dd[x]>, <Rt>
```

```
case opc1:opc2 of
    when '1xxx'  advsimd = TRUE;  esize = 8;  index = UInt(opc1<0>:opc2);
    when '0xx1'  advsimd = TRUE;  esize = 16;  index = UInt(opc1<0>:opc2<1>);
    when '0x00'  advsimd = FALSE;  esize = 32;  index = UInt(opc1<0>);
    when '0x10'  UNDEFINED;
d = UInt(D:Vd);  t = UInt(Rt);
if t == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

T1



T1

```
VMOV{<c>}{<q>}{.<size>} <Dd[x]>, <Rt>
```

```
case opc1:opc2 of
    when '1xxx'  advsimd = TRUE;  esize = 8;  index = UInt(opc1<0>:opc2);
    when '0xx1'  advsimd = TRUE;  esize = 16;  index = UInt(opc1<0>:opc2<1>);
    when '0x00'  advsimd = FALSE;  esize = 32;  index = UInt(opc1<0>);
    when '0x10'  UNDEFINED;
d = UInt(D:Vd);  t = UInt(Rt);
if t == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<size> The data size. It must be one of:

8

Encoded as opc1<1> = 1. [x] is encoded in opc1<0>, opc2.

16
Encoded as opc1<1> = 0, opc2<0> = 1. [x] is encoded in opc1<0>, opc2<1>.

32
Encoded as opc1<1> = 0, opc2 = 0b00. [x] is encoded in opc1<0>.

omitted
Equivalent to 32.

<Dd[x]> The scalar. The register <Dd> is encoded in D:Vd. For details of how [x] is encoded, see the description of <size>.

<Rt> The source general-purpose register.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
    Elem[D[d],index,esize] = R[t]<esize-1:0>;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

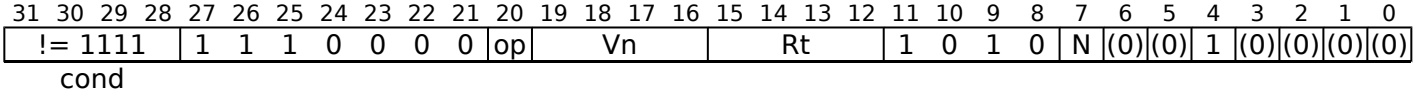
VMOV (between general-purpose register and single-precision)

Copy a general-purpose register to or from a 32-bit SIMD&FP register. This instruction transfers the value held in a 32-bit SIMD&FP register to a general-purpose register, or the value held in a general-purpose register to a 32-bit SIMD&FP register.

Depending on settings in the *CPACR*, *NSACR*, *HCPTR*, and *FPEXC* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



From general-purpose register (op == 0)

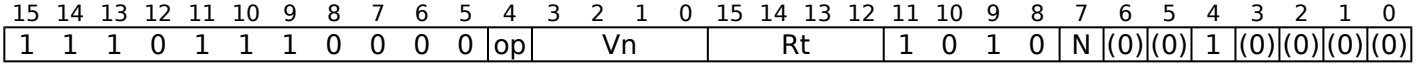
```
VMOV{<c>}{<q>} <Sn>, <Rt>
```

To general-purpose register (op == 1)

```
VMOV{<c>}{<q>} <Rt>, <Sn>
```

```
to_arm_register = (op == '1');  t = UInt(Rt);  n = UInt(Vn:N);
if t == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

T1



From general-purpose register (op == 0)

```
VMOV{<c>}{<q>} <Sn>, <Rt>
```

To general-purpose register (op == 1)

```
VMOV{<c>}{<q>} <Rt>, <Sn>
```

```
to_arm_register = (op == '1');  t = UInt(Rt);  n = UInt(Vn:N);
if t == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<Rt>	Is the general-purpose register that <Sn> will be transferred to or from, encoded in the "Rt" field.
<Sn>	Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Vn:N" field.
<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    if to_arm_register then
        R[t] = S[n];
    else
        S[n] = R[t];
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VMOV (scalar to general-purpose register)

Copy a vector element to a general-purpose register with sign or zero extension copies a byte, halfword, or word from an Advanced SIMD scalar to a general-purpose register. Bytes and halfwords can be either zero-extended or sign-extended.

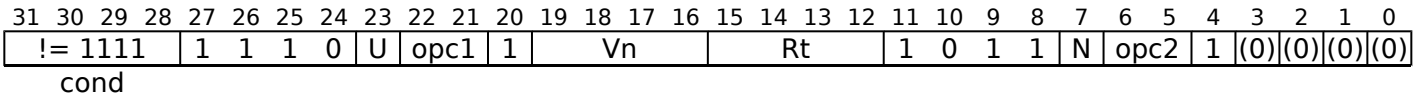
On a Floating-point-only system, this instruction transfers one word from the upper or lower half of a double-precision floating-point register to a general-purpose register. This is an identical operation to the Advanced SIMD single word transfer.

For more information about scalars see *Advanced SIMD scalars*.

Depending on settings in the *CPACR*, *NSACR*, *HCPTR*, and *FPEXC* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

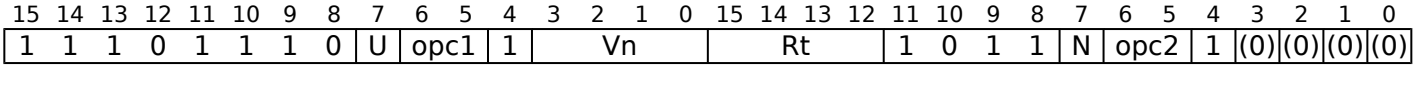


A1

```
VMOV{<c>}{<q>}{.<dt>} <Rt>, <Dn[x]>
```

```
case U:opc1:opc2 of
    when 'x1xxx'  advsimd = TRUE;  esize = 8;  index = UInt(opc1<0>:opc2);
    when 'x0xx1'  advsimd = TRUE;  esize = 16;  index = UInt(opc1<0>:opc2<1>);
    when '00x00'  advsimd = FALSE;  esize = 32;  index = UInt(opc1<0>);
    when '10x00'  UNDEFINED;
    when 'x0x10'  UNDEFINED;
t = UInt(Rt);  n = UInt(N:Vn);  unsigned = (U == '1');
if t == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

T1



T1

```
VMOV{<c>}{<q>}{.<dt>} <Rt>, <Dn[x]>
```

```
case U:opc1:opc2 of
    when 'x1xxx'  advsimd = TRUE;  esize = 8;  index = UInt(opc1<0>:opc2);
    when 'x0xx1'  advsimd = TRUE;  esize = 16;  index = UInt(opc1<0>:opc2<1>);
    when '00x00'  advsimd = FALSE;  esize = 32;  index = UInt(opc1<0>);
    when '10x00'  UNDEFINED;
    when 'x0x10'  UNDEFINED;
t = UInt(Rt);  n = UInt(N:Vn);  unsigned = (U == '1');
if t == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<dt>	The data type. It must be one of:

S8
Encoded as U = 0, opc1<1> = 1. [x] is encoded in opc1<0>, opc2.

S16
Encoded as U = 0, opc1<1> = 0, opc2<0> = 1. [x] is encoded in opc1<0>, opc2<1>.

U8
Encoded as U = 1, opc1<1> = 1. [x] is encoded in opc1<0>, opc2.

U16
Encoded as U = 1, opc1<1> = 0, opc2<0> = 1. [x] is encoded in opc1<0>, opc2<1>.

32
Encoded as U = 0, opc1<1> = 0, opc2 = 0b00. [x] is encoded in opc1<0>.

omitted
Equivalent to 32.

<Rt> The destination general-purpose register.

<Dn[x]> The scalar. For details of how [x] is encoded see the description of <dt>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
    if unsigned then
        R[t] = ZeroExtend(Elem[D[n],index,esize], 32);
    else
        R[t] = SignExtend(Elem[D[n],index,esize], 32);
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

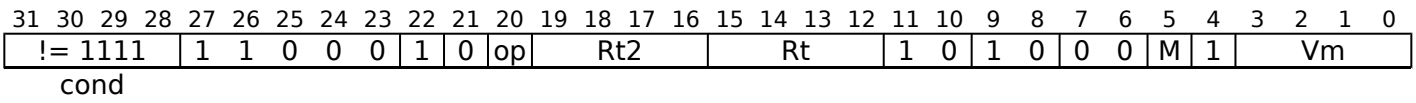
VMOV (between two general-purpose registers and two single-precision registers)

Copy two general-purpose registers to a pair of 32-bit SIMD&FP registers transfers the contents of two consecutively numbered single-precision Floating-point registers to two general-purpose registers, or the contents of two general-purpose registers to a pair of single-precision Floating-point registers. The general-purpose registers do not have to be contiguous.

Depending on settings in the *CPACR*, *NSACR*, *HCPTR*, and *FPEXC* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



From general-purpose registers (op == 0)

```
VMOV{<c>}{<q>} <Sm>, <Sm1>, <Rt>, <Rt2>
```

To general-purpose registers (op == 1)

```
VMOV{<c>}{<q>} <Rt>, <Rt2>, <Sm>, <Sm1>
```

```
to_arm_registers = (op == '1');  t = UInt(Rt);  t2 = UInt(Rt2);  m = UInt(Vm:M);
if t == 15 || t2 == 15 || m == 31 then UNPREDICTABLE;
if to_arm_registers && t == t2 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

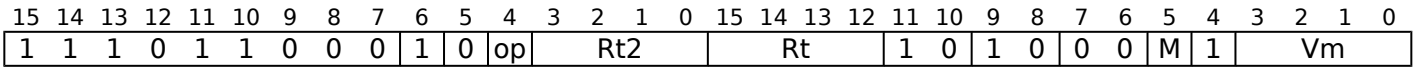
If `to_arm_registers && t == t2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The value in the destination register is UNKNOWN.

If `m == 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- One or more of the single-precision registers become UNKNOWN for a move to the single-precision register. The general-purpose registers listed in the instruction become UNKNOWN for a move from the single-precision registers. This behavior does not affect any other general-purpose registers.

T1



From general-purpose registers (op == 0)

```
VMOV{<c>}{<q>} <Sm>, <Sm1>, <Rt>, <Rt2>
```

To general-purpose registers (op == 1)

```
VMOV{<c>}{<q>} <Rt>, <Rt2>, <Sm>, <Sm1>
```

```
to_arm_registers = (op == '1');  t = UInt(Rt);  t2 = UInt(Rt2);  m = UInt(Vm:M);
if t == 15 || t2 == 15 || m == 31 then UNPREDICTABLE;
if to_arm_registers && t == t2 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `to_arm_registers && t == t2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

If `m == 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- One or more of the single-precision registers become UNKNOWN for a move to the single-precision register. The general-purpose registers listed in the instruction become UNKNOWN for a move from the single-precision registers. This behavior does not affect any other general-purpose registers.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *VMOV (between two general-purpose registers and two single-precision registers)*.

Assembler Symbols

<Rt2>	Is the second general-purpose register that <Sm1> will be transferred to or from, encoded in the "Rt2" field.
<Rt>	Is the first general-purpose register that <Sm> will be transferred to or from, encoded in the "Rt" field.
<Sm1>	Is the 32-bit name of the second SIMD&FP register to be transferred. This is the next SIMD&FP register after <Sm>.
<Sm>	Is the 32-bit name of the first SIMD&FP register to be transferred, encoded in the "Vm:M" field.
<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    if to_arm_registers then
        R[t] = S[m];
        R[t2] = S[m+1];
    else
        S[m] = R[t];
        S[m+1] = R[t2];
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VMOV (register, SIMD)

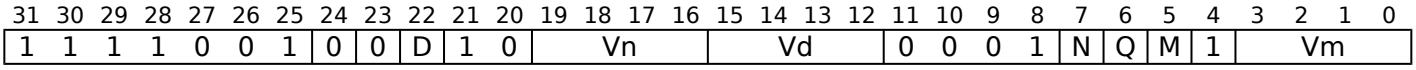
Copy between SIMD registers copies the contents of one SIMD register to another

This is an alias of VORR (register). This means:

- The encodings in this description are named to match the encodings of VORR (register).
- The description of VORR (register) gives the operational pseudocode for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



64-bit SIMD vector (Q == 0)

```
VMOV{<c>}{<q>}{.<dt>} <Dd>, <Dm>
```

is equivalent to

```
VORR{<c>}{<q>}{.<dt>} <Dd>, <Dm>, <Dm>
```

and is the preferred disassembly when `N:Vn == M:Vm`.

128-bit SIMD vector (Q == 1)

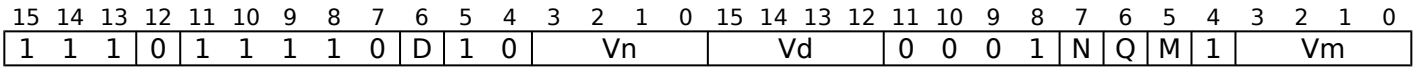
```
VMOV{<c>}{<q>}{.<dt>} <Qd>, <Qm>
```

is equivalent to

```
VORR{<c>}{<q>}{.<dt>} <Qd>, <Qm>, <Qm>
```

and is the preferred disassembly when `N:Vn == M:Vm`.

T1



64-bit SIMD vector (Q == 0)

```
VMOV{<c>}{<q>}{.<dt>} <Dd>, <Dm>
```

is equivalent to

```
VORR{<c>}{<q>}{.<dt>} <Dd>, <Dm>, <Dm>
```

and is the preferred disassembly when `N:Vn == M:Vm`.

128-bit SIMD vector (Q == 1)

```
VMOV{<c>}{<q>}{.<dt>} <Qd>, <Qm>
```

is equivalent to

```
VORR{<c>}{<q>}{.<dt>} <Qd>, <Qm>, <Qm>
```

and is the preferred disassembly when `N:Vn == M:Vm`.

Assembler Symbols

<c>	For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional. For encoding T1: see *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<dt>	An optional data type. <dt> must not be F64, but it is otherwise ignored.
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qm>	Is the 128-bit name of the SIMD&FP source register, encoded in the "N:Vn" and "M:Vm" field as <Qm>*2.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dm>	Is the 64-bit name of the SIMD&FP source register, encoded in the "N:Vn" and "M:Vm" field.

Operation

The description of VORR (register) gives the operational pseudocode for this instruction.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

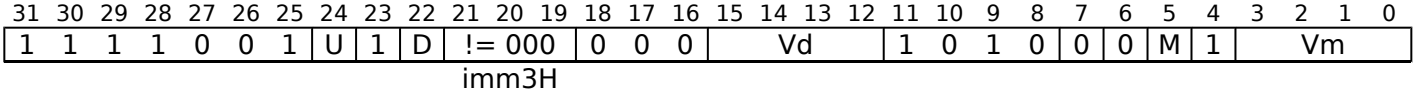
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VMOVL

Vector Move Long takes each element in a doubleword vector, sign or zero-extends them to twice their original length, and places the results in a quadword vector.
Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

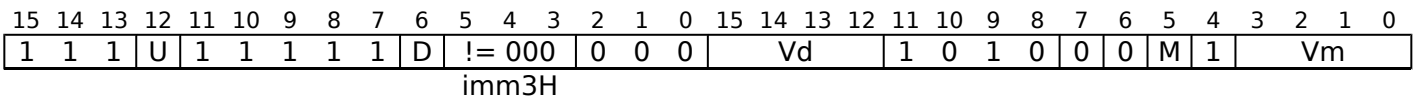
A1



A1

```
VMOVL{<c>}{<q>}.<dt> <Qd>, <Dm>

if imm3H == '000' then SEE "Related encodings";
if imm3H != '001' && imm3H != '010' && imm3H != '100' then SEE "VSHLL";
if Vd<0> == '1' then UNDEFINED;
esize = 8 * UInt(imm3H);
unsigned = (U == '1');  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);
```

T1



T1

```
VMOVL{<c>}{<q>}.<dt> <Qd>, <Dm>

if imm3H == '000' then SEE "Related encodings";
if imm3H != '001' && imm3H != '010' && imm3H != '100' then SEE "VSHLL";
if Vd<0> == '1' then UNDEFINED;
esize = 8 * UInt(imm3H);
unsigned = (U == '1');  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);
```

Related encodings: See *Advanced SIMD one register and modified immediate* for the T32 instruction set, or *Advanced SIMD one register and modified immediate* for the A32 instruction set.

Assembler Symbols

<c>	For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional. For encoding T1: see *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<dt>	Is the data type for the elements of the operand, encoded in “U:imm3H”:

U	imm3H	<dt>
0	001	S8
0	010	S16
0	100	S32
1	001	U8
1	010	U16
1	100	U32

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for e = 0 to elements-1
        result = Int(Elem[Din[m],e,esize], unsigned);
        Elem[Q[d>>1],e,2*esize] = result<2*esize-1:0>;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

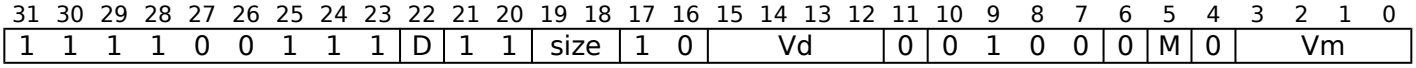
VMOVN

Vector Move and Narrow copies the least significant half of each element of a quadword vector into the corresponding elements of a doubleword vector.
The operand vector elements can be any one of 16-bit, 32-bit, or 64-bit integers. There is no distinction between signed and unsigned integers.
Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

This instruction is used by the pseudo-instructions VRSHRN (zero), and VSHRN (zero).

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

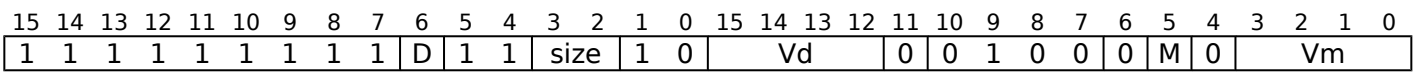
A1



A1

```
VMOVN{<c>}{<q>}.<dt> <Dd>, <Qm>

if size == '11' then UNDEFINED;
if Vm<0> == '1' then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);
```

T1



T1

```
VMOVN{<c>}{<q>}.<dt> <Dd>, <Qm>

if size == '11' then UNDEFINED;
if Vm<0> == '1' then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);
```

Assembler Symbols

<c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.
For encoding T1: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the operand, encoded in “size”:

size	<dt>
00	I16
01	I32
10	I64
11	RESERVED

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for e = 0 to elements-1
        Elem[D[d],e,esize] = Elem[Qin[m>>1],e,2*esize]<esize-1:0>;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

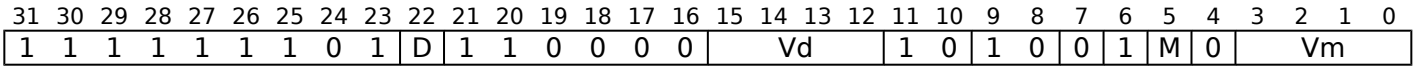
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VMOVX

Vector Move extraction. This instruction copies the upper 16 bits of the 32-bit source SIMD&FP register into the lower 16 bits of the 32-bit destination SIMD&FP register, while clearing the remaining bits to zero.
Depending on settings in the *CPACR*, *NSACR*, *HCPTR*, and *FPEXC* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

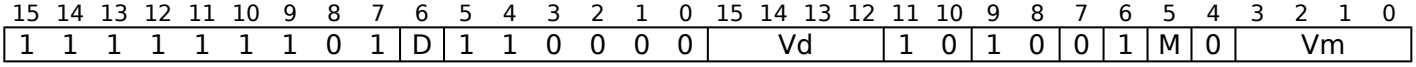
A1
(FEAT_FP16)



A1

```
VMOVX{<q>}.F16 <Sd>, <Sm>

if !HaveFP16Ext() then UNDEFINED;
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
d = UInt(Vd:D); m = UInt(Vm:M);
```

T1
(FEAT_FP16)



T1

```
VMOVX{<q>}.F16 <Sd>, <Sm>

if InITBlock() then UNPREDICTABLE;
if !HaveFP16Ext() then UNDEFINED;
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
d = UInt(Vd:D); m = UInt(Vm:M);
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<q>	See *Standard assembler syntax fields*.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
<Sm>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    S[d] = Zeros(16) : S[m]<31:16>;
```

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VMRS

Move SIMD&FP Special register to general-purpose register moves the value of an Advanced SIMD and floating-point System register to a general-purpose register. When the specified System register is the *FPSCR*, a form of the instruction transfers the *FPSCR*.{N, Z, C, V} condition flags to the *APSR*.{N, Z, C, V} condition flags.

Depending on settings in the *CPACR*, *NSACR*, *HCPTR*, and *FPEXC* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

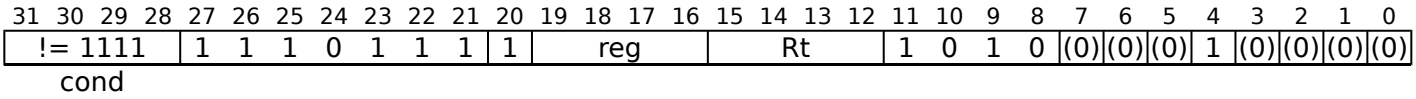
When these settings permit the execution of Advanced SIMD and floating-point instructions, if the specified floating-point System register is not the *FPSCR*, the instruction is UNDEFINED if executed in User mode.

In an implementation that includes EL2, when *HCR*.TID0 is set to 1, any VMRS access to *FPSID* from a Non-secure EL1 mode that would be permitted if *HCR*.TID0 was set to 0 generates a Hyp Trap exception. For more information, see *ID group 0, Primary device identification registers*.

For simplicity, the VMRS pseudocode does not show the possible trap to Hyp mode.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



A1

```
VMRS{<c>}{<q>} <Rt>, <spec_reg>
```

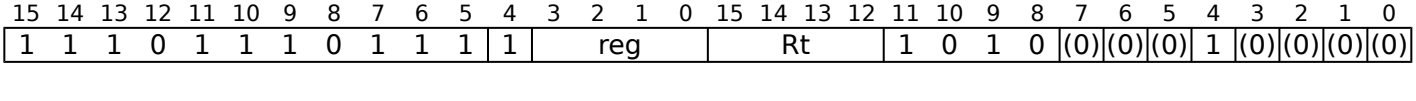
```
t = UInt(Rt);
if !(reg IN {'000x', '0101', '011x', '1000'}) then UNPREDICTABLE;
if t == 15 && reg != '0001' then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

CONSTRAINED UNPREDICTABLE behavior

If `!(reg IN {'000x', '0101', '011x', '1000'})`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction transfers an UNKNOWN value to the specified target register. When the Rt field holds the value 0b1111, the specified target register is the *APSR*.{N, Z, C, V} bits, and these bits become UNKNOWN. Otherwise, the specified target register is the register specified by the Rt field, R0 - R14.

T1



T1

```
VMRS{<c>}{<q>} <Rt>, <spec_reg>
```

```
t = UInt(Rt);
if !(reg IN {'000x', '0101', '011x', '1000'}) then UNPREDICTABLE;
if t == 15 && reg != '0001' then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

CONSTRAINED UNPREDICTABLE behavior

If `!(reg IN {'000x', '0101', '011x', '1000'})`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

- The instruction transfers an UNKNOWN value to the specified target register. When the Rt field holds the value 0b1111, the specified target register is the *APSR*.{N, Z, C, V} bits, and these bits become UNKNOWN. Otherwise, the specified target register is the register specified by the Rt field, R0 - R14.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<Rt> Is the general-purpose destination register, encoded in the "Rt" field. Is one of:

R0-R14
General-purpose register.

APSR_nzcv
Permitted only when <spec_reg> is FPSCR. Encoded as 0b1111. The instruction transfers the *FPSCR*.{N, Z, C, V} condition flags to the *APSR*.{N, Z, C, V} condition flags.

<spec_reg> Is the source Advanced SIMD and floating-point System register, encoded in “reg”:

reg	<spec_reg>
0000	FPSID
0001	FPSCR
001x	UNPREDICTABLE
0100	UNPREDICTABLE
0101	MVFR2
0110	MVFR1
0111	MVFR0
1000	FPEXC
1001	UNPREDICTABLE
101x	UNPREDICTABLE
11xx	UNPREDICTABLE

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if reg == '0001' then                 // FPSCR
        CheckVFPEnabled(TRUE);
        if t == 15 then
            PSTATE.<N,Z,C,V> = FPSR.<N,Z,C,V>;
        else
            R[t] = FPSCR;
    elsif PSTATE.EL == EL0 then
        UNDEFINED;                        // Non-FPSCR registers accessible only at PL1 or above
    else
        CheckVFPEnabled(FALSE);           // Non-FPSCR registers are not affected by FPEXC.EN
        AArch32.CheckAdvSIMDOrFPRegisterTraps(reg);
        case reg of
            when '0000'  R[t] = FPSID;
            when '0101'  R[t] = MVFR2;
            when '0110'  R[t] = MVFR1;
            when '0111'  R[t] = MVFR0;
            when '1000'  R[t] = FPEXC;
            otherwise    Unreachable();   // Dealt with above or in encoding-specific pseudocode
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VMSR

Move general-purpose register to SIMD&FP Special register moves the value of a general-purpose register to a floating-point System register.

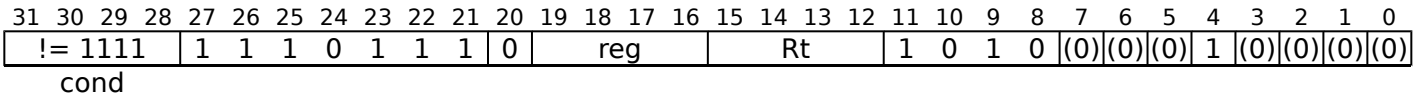
Depending on settings in the *CPACR*, *NSACR*, *HCPTR*, and *FPEXC* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

When these settings permit the execution of Advanced SIMD and floating-point instructions:

- If the specified floating-point System register is *FPSID* or *FPEXC*, the instruction is UNDEFINED if executed in User mode.
- If the specified floating-point System register is the *FPSID* and the instruction is executed in a mode other than User mode, the instruction is ignored.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



A1

```
VMSR{<c>}{<q>} <spec_reg>, <Rt>
```

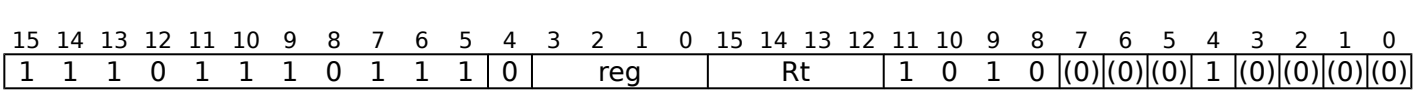
```
t = UInt(Rt);
if reg != '000x' && reg != '1000' then
    Constraint c = ConstrainUnpredictable(Unpredictable_VMSR);
    assert c IN {Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNDEF
            UNDEFINED;
        when Constraint_NOP
            EndOfInstruction();
if t == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

CONSTRAINED UNPREDICTABLE behavior

If `reg != '000x' && reg != '1000'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction transfers the value in the general-purpose register to one of the allocated registers accessible using VMSR at the same Exception level.

T1



T1

```
VMSR{<c>}{<q>} <spec_reg>, <Rt>
```

```
t = UInt(Rt);
if reg != '000x' && reg != '1000' then
    Constraint c = ConstrainUnpredictable(Unpredictable_VMSR);
    assert c IN {Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNDEF
            UNDEFINED;
        when Constraint_NOP
            EndOfInstruction();
if t == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

CONSTRAINED UNPREDICTABLE behavior

If `reg != '000x' && reg != '1000'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction transfers the value in the general-purpose register to one of the allocated registers accessible using VMSR at the same Exception level.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<spec_reg> Is the destination Advanced SIMD and floating-point System register, encoded in “reg”:

reg	<spec_reg>
0000	FPSID
0001	FPSCR
001x	UNPREDICTABLE
01xx	UNPREDICTABLE
1000	FPEXC
1001	UNPREDICTABLE
101x	UNPREDICTABLE
11xx	UNPREDICTABLE

<Rt> Is the general-purpose source register, encoded in the "Rt" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if reg == '0001' then                 // FPSCR
        CheckVFPEnabled(TRUE);
        FPSCR = R[t];
    elsif PSTATE.EL == EL0 then
        UNDEFINED;                        // Non-FPSCR registers accessible only at PL1 or above
    else
        CheckVFPEnabled(FALSE);           // Non-FPSCR registers are not affected by FPEXC.EN
        case reg of
            when '0000'                   // VMSR access to FPSID is ignored
            when '1000'  FPEXC = R[t];
            otherwise    Unreachable();   // Dealt with above or in encoding-specific pseudocode
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

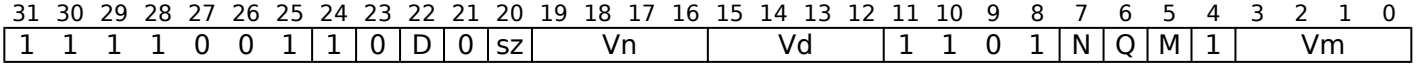
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VMUL (floating-point)

Vector Multiply multiplies corresponding elements in two vectors, and places the results in the destination vector. Depending on settings in the *CPACR*, *NSACR*, *HCPTR*, and *FPEXC* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2) .

A1



64-bit SIMD vector (Q == 0)

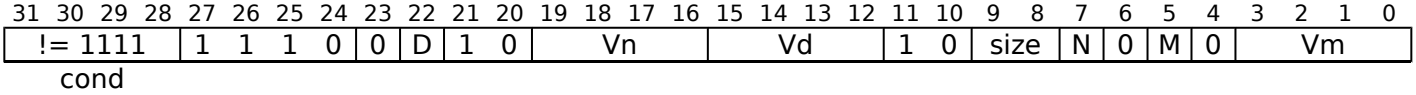
```
VMUL{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

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



Half-precision scalar (size == 01)
(FEAT_FP16)

```
VMUL{<c>}{<q>}.F16 {<Sd>,} <Sn>, <Sm>
```

Single-precision scalar (size == 10)

```
VMUL{<c>}{<q>}.F32 {<Sd>,} <Sn>, <Sm>
```

Double-precision scalar (size == 11)

```
VMUL{<c>}{<q>}.F64 {<Dd>,} <Dn>, <Dm>
```

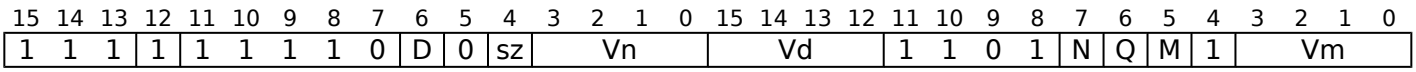
```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
advsimd = FALSE;

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)

```
VMUL{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VMUL{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

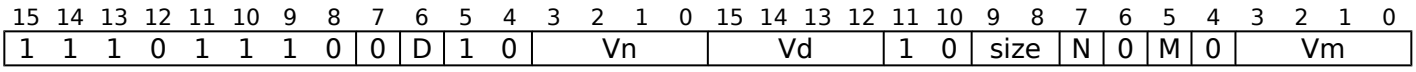
```
if sz == '1' && InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
advsimd = TRUE;
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd);  n = 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



Half-precision scalar (size == 01)
(FEAT_FP16)

```
VMUL{<c>}{<q>}.F16 {<Sd>,} <Sn>, <Sm>
```

Single-precision scalar (size == 10)

```
VMUL{<c>}{<q>}.F32 {<Sd>,} <Sn>, <Sm>
```

Double-precision scalar (size == 11)

```
VMUL{<c>}{<q>}.F64 {<Dd>,} <Dn>, <Dm>
```

```
if size == '01' && InITBlock()  then UNPREDICTABLE;
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
advsimd = FALSE;

case size of
    when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.
For encoding A2, T1 and T2: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the vectors, encoded in “sz”:

sz	<dt>
0	F32
1	F16

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

<Sn> Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.

<Sm> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
    if advsimd then  // Advanced SIMD instruction
        for r = 0 to regs-1
            for e = 0 to elements-1
                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_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

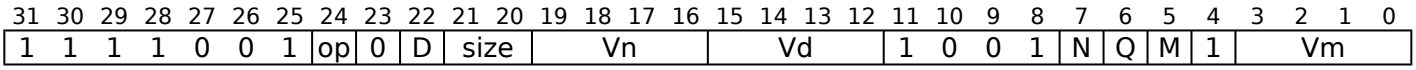
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VMUL (integer and polynomial)

Vector Multiply multiplies corresponding elements in two vectors.
For information about multiplying polynomials, see *Polynomial arithmetic over {0, 1}*.
Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



64-bit SIMD vector (Q == 0)

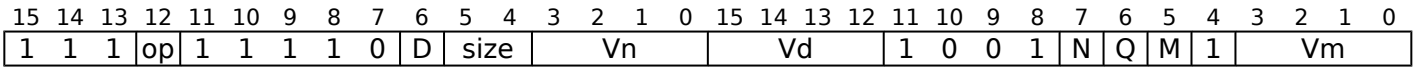
```
VMUL{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VMUL{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

```
if size == '11' || (op == '1' && size != '00') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = FALSE;  // "Don't care" value: TRUE produces same functionality
polynomial = (op == '1');  long_destination = FALSE;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1



64-bit SIMD vector (Q == 0)

```
VMUL{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VMUL{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

```
if size == '11' || (op == '1' && size != '00') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = FALSE;  // "Don't care" value: TRUE produces same functionality
polynomial = (op == '1');  long_destination = FALSE;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>	For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional. For encoding T1: see *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<dt>	Is the data type for the elements of the operands, encoded in “op:size”:

op	size	<dt>
0	00	I8
0	01	I16
0	10	I32
1	00	P8

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[Din[n+r],e,esize];  op1val = Int(op1, unsigned);
            op2 = Elem[Din[m+r],e,esize];  op2val = Int(op2, unsigned);
            if polynomial then
                product = PolynomialMult(op1,op2);
            else
                product = (op1val*op2val)<2*esize-1:0>;
            if long_destination then
                Elem[Q[d>>1],e,2*esize] = product;
            else
                Elem[D[d+r],e,esize] = product<esize-1:0>;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

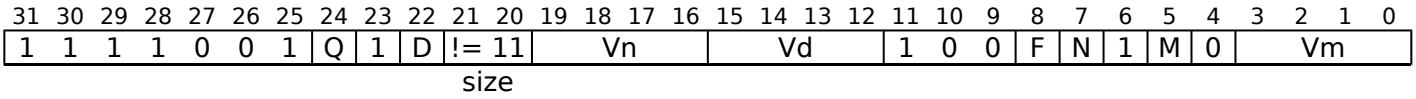
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VMUL (by scalar)

Vector Multiply multiplies each element in a vector by a scalar, and places the results in a second vector.
For more information about scalars see *Advanced SIMD scalars*.
Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



64-bit SIMD vector (Q == 0)

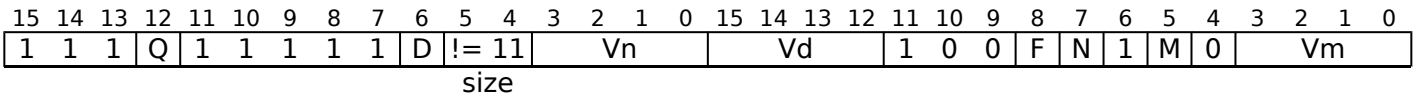
```
VMUL{<c>}{<q>}.<dt> {<Dd>,} <Dn>, <Dm>[<index>]
```

128-bit SIMD vector (Q == 1)

```
VMUL{<c>}{<q>}.<dt> {<Qd>,} <Qn>, <Dm>[<index>]
```

```
if size == '11' then SEE "Related encodings";
if size == '00' || (F == '1' && size == '01' && !HaveFP16Ext()) then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
unsigned = FALSE;  // "Don't care" value: TRUE produces same functionality
floating_point = (F == '1');  long_destination = FALSE;
d = UInt(D:Vd);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```

T1



64-bit SIMD vector (Q == 0)

```
VMUL{<c>}{<q>}.<dt> {<Dd>,} <Dn>, <Dm>[<index>]
```

128-bit SIMD vector (Q == 1)

```
VMUL{<c>}{<q>}.<dt> {<Qd>,} <Qn>, <Dm>[<index>]
```

```
if size == '11' then SEE "Related encodings";
if F == '1' && size == '01' && InITBlock() then UNPREDICTABLE;
if size == '00' || (F == '1' && size == '01' && !HaveFP16Ext()) then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
unsigned = FALSE;  // "Don't care" value: TRUE produces same functionality
floating_point = (F == '1');  long_destination = FALSE;
d = UInt(D:Vd);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```

CONSTRAINED UNPREDICTABLE behavior

If `F == '1' && size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Related encodings: See *Advanced SIMD data-processing* for the T32 instruction set, or *Advanced SIMD data-processing* for the A32 instruction set.

Assembler Symbols

<c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<dt> Is the data type for the scalar and the elements of the operand vector, encoded in “F:size”:

F	size	<dt>
0	01	I16
0	10	I32
1	01	F16
1	10	F32

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register. When <dt> is I16 or F16, this is encoded in the "Vm<2:0>" field. Otherwise it is encoded in the "Vm" field.

<index> Is the element index. When <dt> is I16 or F16, this is in the range 0 to 3 and is encoded in the "M:Vm<3>" field. Otherwise it is in the range 0 to 1 and is encoded in the "M" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    op2 = Elem[Din[m],index,esize];  op2val = Int(op2, unsigned);
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[Din[n+r],e,esize];  op1val = Int(op1, unsigned);
            if floating_point then
                Elem[D[d+r],e,esize] = FPMul(op1, op2, StandardFPSCRValue());
            else
                if long_destination then
                    Elem[Q[d>>1],e,2*esize] = (op1val*op2val)<2*esize-1:0>;
                else
                    Elem[D[d+r],e,esize] = (op1val*op2val)<esize-1:0>;
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VMULL (integer and polynomial)

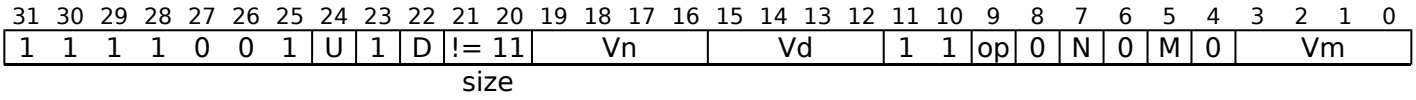
Vector Multiply Long multiplies corresponding elements in two vectors. The destination vector elements are twice as long as the elements that are multiplied.

For information about multiplying polynomials see *Polynomial arithmetic over {0, 1}*.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

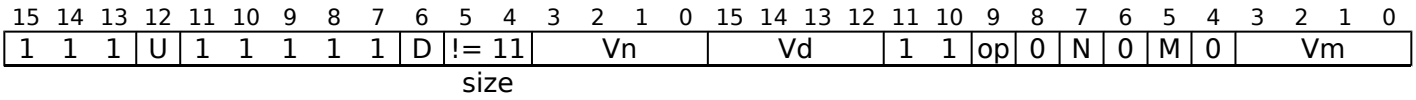


A1

```
VMULL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>
```

```
if size == '11' then SEE "Related encodings";
unsigned = (U == '1'); polynomial = (op == '1'); long_destination = TRUE;
esize = 8 << UInt(size); elements = 64 DIV esize;
if polynomial then
    if U == '1' || size == '01' then UNDEFINED;
    if size == '10' then    // .p64
        if !HaveBit128PMULLExt() then UNDEFINED;
        esize = 64; elements = 1;
if Vd<0> == '1' then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = 1;
```

T1



T1

```
VMULL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>
```

```
if size == '11' then SEE "Related encodings";
unsigned = (U == '1'); polynomial = (op == '1'); long_destination = TRUE;
esize = 8 << UInt(size); elements = 64 DIV esize;
if polynomial then
    if U == '1' || size == '01' then UNDEFINED;
    if size == '10' then    // .p64
        if InITBlock() then UNPREDICTABLE;
        if !HaveBit128PMULLExt() then UNDEFINED;
        esize = 64; elements = 1;
if Vd<0> == '1' then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = 1;
```

CONSTRAINED UNPREDICTABLE behavior

If `op == '1' && size == '10' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Related encodings: See *Advanced SIMD data-processing* for the T32 instruction set, or *Advanced SIMD data-processing* for the A32 instruction set.

Assembler Symbols

<c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.
For encoding T1: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the operands, encoded in “op:U:size”:

op	U	size	<dt>
0	0	00	S8
0	0	01	S16
0	0	10	S32
0	1	00	U8
0	1	01	U16
0	1	10	U32
1	0	00	P8
1	0	10	P64

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[Din[n+r],e,esize];  op1val = Int(op1, unsigned);
            op2 = Elem[Din[m+r],e,esize];  op2val = Int(op2, unsigned);
            if polynomial then
                product = PolynomialMult(op1,op2);
            else
                product = (op1val*op2val)<2*esize-1:0>;
            if long_destination then
                Elem[Q[d>>1],e,2*esize] = product;
            else
                Elem[D[d+r],e,esize] = product<esize-1:0>;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VMULL (by scalar)

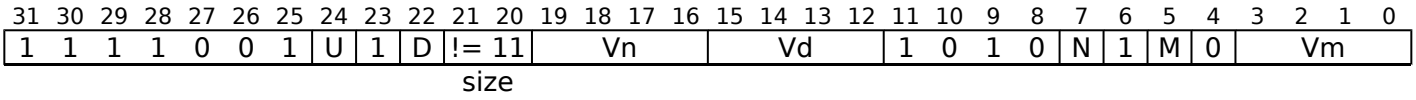
Vector Multiply Long multiplies each element in a vector by a scalar, and places the results in a second vector. The destination vector elements are twice as long as the elements that are multiplied.

For more information about scalars see *Advanced SIMD scalars*.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

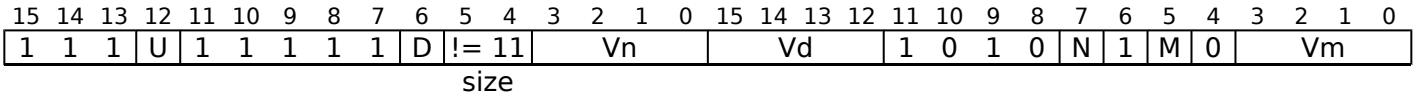


A1

```
VMULL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>[<index>]
```

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
unsigned = (U == '1');  long_destination = TRUE;  floating_point = FALSE;
d = UInt(D:Vd);  n = UInt(N:Vn);  regs = 1;
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```

T1



T1

```
VMULL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>[<index>]
```

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
unsigned = (U == '1');  long_destination = TRUE;  floating_point = FALSE;
d = UInt(D:Vd);  n = UInt(N:Vn);  regs = 1;
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```

Related encodings: See *Advanced SIMD data-processing* for the T32 instruction set, or *Advanced SIMD data-processing* for the A32 instruction set.

Assembler Symbols

<c>	For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional. For encoding T1: see *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<dt>	Is the data type for the scalar and the elements of the operand vector, encoded in “U:size”:

U	size	<dt>
0	01	S16
0	10	S32
1	01	U16
1	10	U32

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "Vm<2:0>" field when <dt> is S16 or U16, otherwise the "Vm" field.

<index> Is the element index in the range 0 to 3, encoded in the "M:Vm<3>" field when <dt> is S16 or U16, otherwise in range 0 to 1, encoded in the "M" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    op2 = Elem[Din[m],index,esize];  op2val = Int(op2, unsigned);
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[Din[n+r],e,esize];  op1val = Int(op1, unsigned);
            if floating_point then
                Elem[D[d+r],e,esize] = FPMul(op1, op2, StandardFPSCRValue());
            else
                if long_destination then
                    Elem[Q[d>>1],e,2*esize] = (op1val*op2val)<2*esize-1:0>;
                else
                    Elem[D[d+r],e,esize] = (op1val*op2val)<esize-1:0>;
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

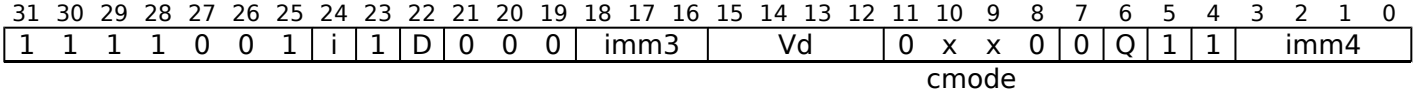
VMVN (immediate)

Vector Bitwise NOT (immediate) places the bitwise inverse of an immediate integer constant into every element of the destination register.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1 , A2 and A3) and T32 (T1 , T2 and T3) .

A1



64-bit SIMD vector (Q == 0)

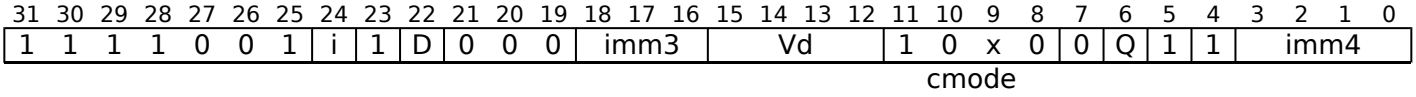
```
VMVN{<c>}{<q>}.I32 <Dd>, #<imm>
```

128-bit SIMD vector (Q == 1)

```
VMVN{<c>}{<q>}.I32 <Qd>, #<imm>
```

```
if (cmode<0> == '1' && cmode<3:2> != '11') || cmode<3:1> == '111' then SEE "Related encodings";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
imm64 = AdvSIMDExpandImm('1', cmode, i:imm3:imm4);
d = UInt(D:Vd);  regs = if Q == '0' then 1 else 2;
```

A2



64-bit SIMD vector (Q == 0)

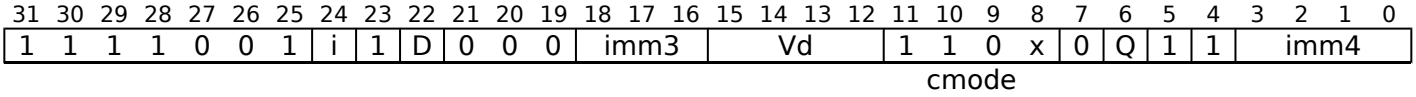
```
VMVN{<c>}{<q>}.I16 <Dd>, #<imm>
```

128-bit SIMD vector (Q == 1)

```
VMVN{<c>}{<q>}.I16 <Qd>, #<imm>
```

```
if (cmode<0> == '1' && cmode<3:2> != '11') || cmode<3:1> == '111' then SEE "Related encodings";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
imm64 = AdvSIMDExpandImm('1', cmode, i:imm3:imm4);
d = UInt(D:Vd);  regs = if Q == '0' then 1 else 2;
```

A3



64-bit SIMD vector (Q == 0)

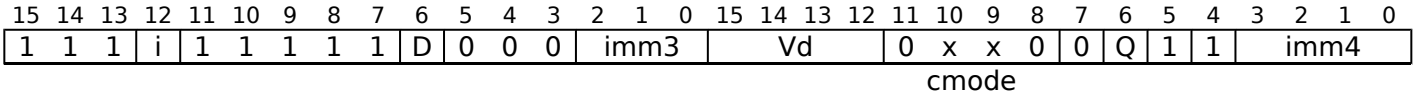
```
VMVN{<c>}{<q>}.I32 <Dd>, #<imm>
```

128-bit SIMD vector (Q == 1)

```
VMVN{<c>}{<q>}.I32 <Qd>, #<imm>
```

```
if (cmode<0> == '1' && cmode<3:2> != '11') || cmode<3:1> == '111' then SEE "Related encodings";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
imm64 = AdvSIMDExpandImm('1', cmode, i:imm3:imm4);
d = UInt(D:Vd);  regs = if Q == '0' then 1 else 2;
```

T1



64-bit SIMD vector (Q == 0)

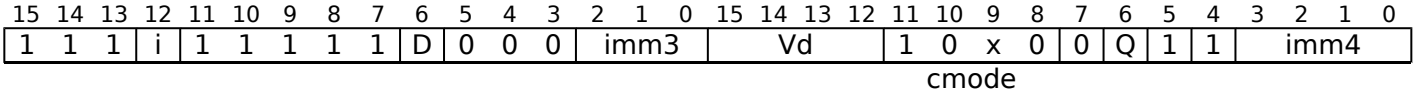
```
VMVN{<c>}{<q>}.I32 <Dd>, #<imm>
```

128-bit SIMD vector (Q == 1)

```
VMVN{<c>}{<q>}.I32 <Qd>, #<imm>
```

```
if (cmode<0> == '1' && cmode<3:2> != '11') || cmode<3:1> == '111' then SEE "Related encodings";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
imm64 = AdvSIMDExpandImm('1', cmode, i:imm3:imm4);
d = UInt(D:Vd);  regs = if Q == '0' then 1 else 2;
```

T2



64-bit SIMD vector (Q == 0)

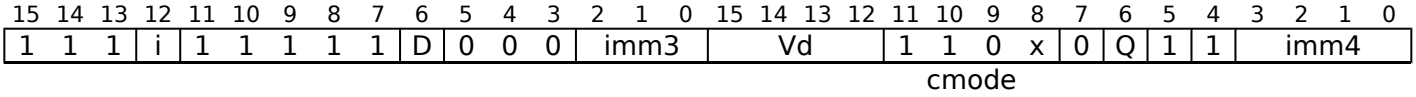
```
VMVN{<c>}{<q>}.I16 <Dd>, #<imm>
```

128-bit SIMD vector (Q == 1)

```
VMVN{<c>}{<q>}.I16 <Qd>, #<imm>
```

```
if (cmode<0> == '1' && cmode<3:2> != '11') || cmode<3:1> == '111' then SEE "Related encodings";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
imm64 = AdvSIMDExpandImm('1', cmode, i:imm3:imm4);
d = UInt(D:Vd);  regs = if Q == '0' then 1 else 2;
```

T3



64-bit SIMD vector (Q == 0)

```
VMVN{<c>}{<q>}.I32 <Dd>, #<imm>
```

128-bit SIMD vector (Q == 1)

```
VMVN{<c>}{<q>}.I32 <Qd>, #<imm>
```

```
if (cmode<0> == '1' && cmode<3:2> != '11') || cmode<3:1> == '111' then SEE "Related encodings";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
imm64 = AdvSIMDExpandImm('1', cmode, i:imm3:imm4);
d = UInt(D:Vd);  regs = if Q == '0' then 1 else 2;
```

Related encodings: See *Advanced SIMD one register and modified immediate* for the T32 instruction set, or *Advanced SIMD one register and modified immediate* for the A32 instruction set.

Assembler Symbols

<c>	For encoding A1, A2 and A3: see *Standard assembler syntax fields*. This encoding must be unconditional. For encoding T1, T2 and T3: see *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<imm>	Is a constant of the specified type that is replicated to fill the destination register. For details of the range of constants available and the encoding of <imm>, see *Modified immediate constants in T32 and A32 Advanced SIMD instructions*.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        D[d+r] = NOT(imm64);
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

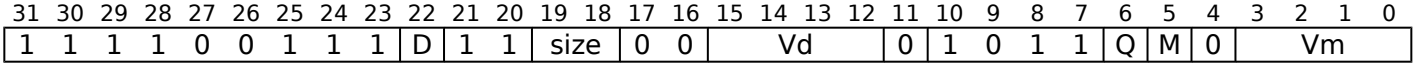
VMVN (register)

Vector Bitwise NOT (register) takes a value from a register, inverts the value of each bit, and places the result in the destination register. The registers can be either doubleword or quadword.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



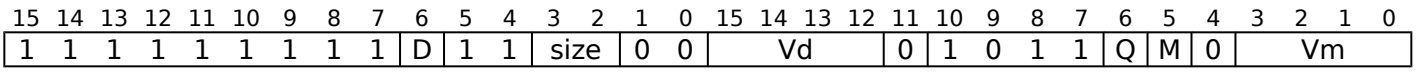
64-bit SIMD vector (Q == 0)

```
VMVN{<c>}{<q>}{.<dt>} <Dd>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VMVN{<c>}{<q>}{.<dt>} <Qd>, <Qm>

if size != '00' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1



64-bit SIMD vector (Q == 0)

```
VMVN{<c>}{<q>}{.<dt>} <Dd>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VMVN{<c>}{<q>}{.<dt>} <Qd>, <Qm>

if size != '00' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>	For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional. For encoding T1: see *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<dt>	An optional data type. It is ignored by assemblers, and does not affect the encoding.
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qm>	Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dm>	Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        D[d+r] = NOT(D[m+r]);
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

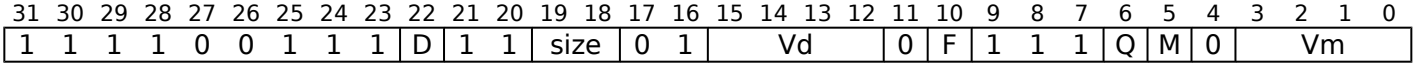
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VNEG

Vector Negate negates each element in a vector, and places the results in a second vector. The floating-point version only inverts the sign bit.
Depending on settings in the *CPACR*, *NSACR*, *HCPTR*, and *FPEXC* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2) .

A1



64-bit SIMD vector (Q == 0)

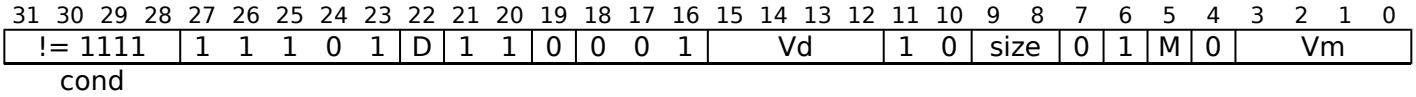
```
VNEG{<c>}{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VNEG{<c>}{<q>}.<dt> <Qd>, <Qm>
```

```
if size == '11' then UNDEFINED;
if F == '1' && ((size == '01' && !HaveFP16Ext()) || size == '00') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
advsimd = TRUE;  floating_point = (F == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

A2



Half-precision scalar (size == 01)
(FEAT_FP16)

```
VNEG{<c>}{<q>}.F16 <Sd>, <Sm>
```

Single-precision scalar (size == 10)

```
VNEG{<c>}{<q>}.F32 <Sd>, <Sm>
```

Double-precision scalar (size == 11)

```
VNEG{<c>}{<q>}.F64 <Dd>, <Dm>
```

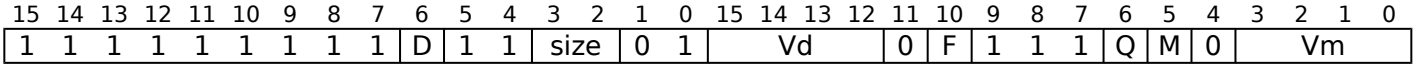
```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
advsimd = FALSE;
case size of
    when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
    when '10' esize = 32; 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



64-bit SIMD vector (Q == 0)

```
VNEG{<c>}{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VNEG{<c>}{<q>}.<dt> <Qd>, <Qm>
```

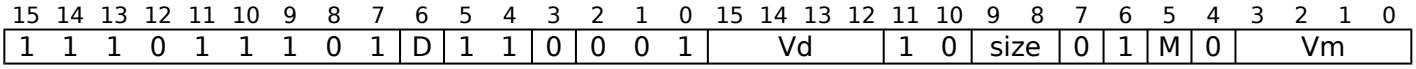
```
if size == '11' then UNDEFINED;
if F == '1' && ((size == '01' && !HaveFP16Ext()) || size == '00') then UNDEFINED;
if F == '1' && size == '01' && InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
advsimd = TRUE;  floating_point = (F == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `F == '1' && size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T2



Half-precision scalar (size == 01)
(FEAT_FP16)

```
VNEG{<c>}{<q>}.F16 <Sd>, <Sm>
```

Single-precision scalar (size == 10)

```
VNEG{<c>}{<q>}.F32 <Sd>, <Sm>
```

Double-precision scalar (size == 11)

```
VNEG{<c>}{<q>}.F64 <Dd>, <Dm>
```

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock()  then UNPREDICTABLE;
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
advsimd = FALSE;
case size of
    when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.

For encoding A2, T1 and T2: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the vectors, encoded in “F:size”:

F	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	01	F16
1	10	F32

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

<Sm> Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
    if advsimd then  // Advanced SIMD instruction
        for r = 0 to regs-1
            for e = 0 to elements-1
                if floating_point then
                    Elem[D[d+r],e,esize] = FPNeg(Elem[D[m+r],e,esize]);
                else
                    result = -SInt(Elem[D[m+r],e,esize]);
                    Elem[D[d+r],e,esize] = result<esize-1:0>;
    else             // VFP instruction
        case esize of
            when 16 S[d] = Zeros(16) : FPNeg(S[m]<15:0>);
            when 32 S[d] = FPNeg(S[m]);
            when 64 D[d] = FPNeg(D[m]);
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check and is operating only on integer vector elements, then the following apply:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VNMLA

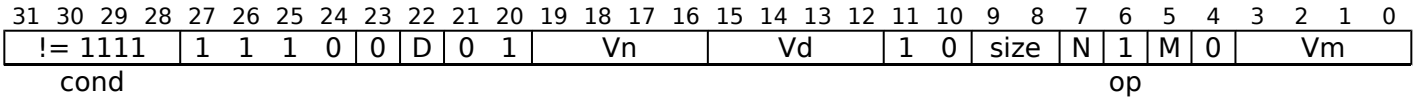
Vector Negate Multiply Accumulate multiplies together two floating-point register values, adds the negation of the floating-point value in the destination register to the negation of the product, and writes the result back to the destination register.

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



Half-precision scalar (size == 01)

(FEAT_FP16)

```
VNMLA{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>
```

Single-precision scalar (size == 10)

```
VNMLA{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>
```

Double-precision scalar (size == 11)

```
VNMLA{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>
```

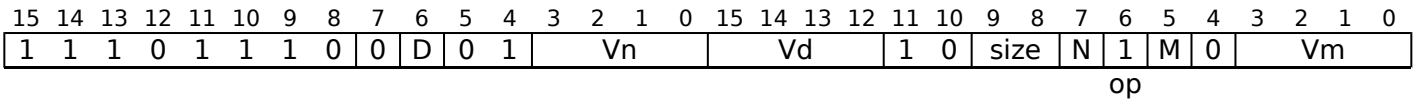
```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
vtype = if op == '1' then VFPNegMul_VNMLA else VFPNegMul_VNMLS;
case size of
    when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1



Half-precision scalar (size == 01)
(FEAT_FP16)

```
VNMLA{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>
```

Single-precision scalar (size == 10)

```
VNMLA{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>
```

Double-precision scalar (size == 11)

```
VNMLA{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>
```

```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock()  then UNPREDICTABLE;
vtype = if op == '1' then VFPNegMul_VNMLA else VFPNegMul_VNMLS;
case size of
    when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
<Sn>	Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.
<Sm>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    case esize of
        when 16
            product16 = FPMul(S[n]<15:0>, S[m]<15:0>, FPSCR[]);
            case vtype of
                when VFPNegMul_VNMLA  S[d] = Zeros(16) : FPAdd(FPNeg(S[d]<15:0>), FPNeg(product16), FPSCR
                when VFPNegMul_VNMLS  S[d] = Zeros(16) : FPAdd(FPNeg(S[d]<15:0>), product16, FPSCR[]);
                when VFPNegMul_VNMUL  S[d] = Zeros(16) : FPNeg(product16);
        when 32
            product32 = FPMul(S[n], S[m], FPSCR[]);
            case vtype of
                when VFPNegMul_VNMLA  S[d] = FPAdd(FPNeg(S[d]), FPNeg(product32), FPSCR[]);
                when VFPNegMul_VNMLS  S[d] = FPAdd(FPNeg(S[d]), product32, FPSCR[]);
                when VFPNegMul_VNMUL  S[d] = FPNeg(product32);
        when 64
            product64 = FPMul(D[n], D[m], FPSCR[]);
            case vtype of
                when VFPNegMul_VNMLA  D[d] = FPAdd(FPNeg(D[d]), FPNeg(product64), FPSCR[]);
                when VFPNegMul_VNMLS  D[d] = FPAdd(FPNeg(D[d]), product64, FPSCR[]);
                when VFPNegMul_VNMUL  D[d] = FPNeg(product64);
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

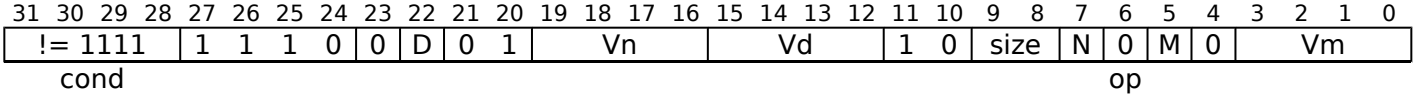
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VNMLS

Vector Negate Multiply Subtract multiplies together two floating-point register values, adds the negation of the floating-point value in the destination register to the product, and writes the result back to the destination register. Depending on settings in the *CPACR*, *NSACR*, *HCPTR*, and *FPEXC* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



Half-precision scalar (size == 01)
(FEAT_FP16)

```
VNMLS{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>
```

Single-precision scalar (size == 10)

```
VNMLS{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>
```

Double-precision scalar (size == 11)

```
VNMLS{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>
```

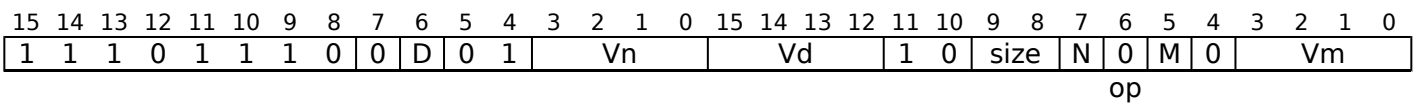
```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
vtype = if op == '1' then VFPNegMul_VNMLA else VFPNegMul_VNMLS;
case size of
    when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1



Half-precision scalar (size == 01)
(FEAT_FP16)

```
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

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    case esize of
        when 16
            product16 = FPMul(S[n]<15:0>, S[m]<15:0>, FPSCR[]);
            case vtype of
                when VFPNegMul_VNMLA  S[d] = Zeros(16) : FPAdd(FPNeg(S[d]<15:0>), FPNeg(product16), FPSCR
                when VFPNegMul_VNMLS  S[d] = Zeros(16) : FPAdd(FPNeg(S[d]<15:0>), product16, FPSCR[]);
                when VFPNegMul_VNMUL  S[d] = Zeros(16) : FPNeg(product16);
        when 32
            product32 = FPMul(S[n], S[m], FPSCR[]);
            case vtype of
                when VFPNegMul_VNMLA  S[d] = FPAdd(FPNeg(S[d]), FPNeg(product32), FPSCR[]);
                when VFPNegMul_VNMLS  S[d] = FPAdd(FPNeg(S[d]), product32, FPSCR[]);
                when VFPNegMul_VNMUL  S[d] = FPNeg(product32);
        when 64
            product64 = FPMul(D[n], D[m], FPSCR[]);
            case vtype of
                when VFPNegMul_VNMLA  D[d] = FPAdd(FPNeg(D[d]), FPNeg(product64), FPSCR[]);
                when VFPNegMul_VNMLS  D[d] = FPAdd(FPNeg(D[d]), product64, FPSCR[]);
                when VFPNegMul_VNMUL  D[d] = FPNeg(product64);
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

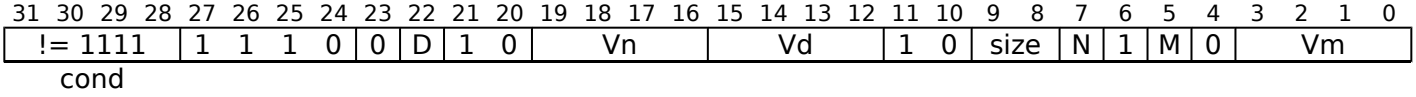
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VNMUL

Vector Negate Multiply multiplies together two floating-point register values, and writes the negation of the result to the destination register.
Depending on settings in the *CPACR*, *NSACR*, *HCPTR*, and *FPEXC* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



Half-precision scalar (size == 01)
(FEAT_FP16)

```
VNMUL{<c>}{<q>}.F16 {<Sd>,} <Sn>, <Sm>
```

Single-precision scalar (size == 10)

```
VNMUL{<c>}{<q>}.F32 {<Sd>,} <Sn>, <Sm>
```

Double-precision scalar (size == 11)

```
VNMUL{<c>}{<q>}.F64 {<Dd>,} <Dn>, <Dm>
```

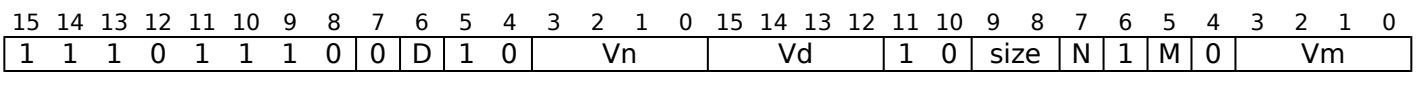
```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '01' && !HaveFP16Ext() then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
vtype = VFPNegMul_VNMUL;
case size of
    when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1



Half-precision scalar (size == 01)
(FEAT_FP16)

```
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

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    case esize of
        when 16
            product16 = FPMul(S[n]<15:0>, S[m]<15:0>, FPSCR[]);
            case vtype of
                when VFPNegMul_VNMLA  S[d] = Zeros(16) : FPAdd(FPNeg(S[d]<15:0>), FPNeg(product16), FPSCR
                when VFPNegMul_VNMLS  S[d] = Zeros(16) : FPAdd(FPNeg(S[d]<15:0>), product16, FPSCR[]);
                when VFPNegMul_VNMUL  S[d] = Zeros(16) : FPNeg(product16);
        when 32
            product32 = FPMul(S[n], S[m], FPSCR[]);
            case vtype of
                when VFPNegMul_VNMLA  S[d] = FPAdd(FPNeg(S[d]), FPNeg(product32), FPSCR[]);
                when VFPNegMul_VNMLS  S[d] = FPAdd(FPNeg(S[d]), product32, FPSCR[]);
                when VFPNegMul_VNMUL  S[d] = FPNeg(product32);
        when 64
            product64 = FPMul(D[n], D[m], FPSCR[]);
            case vtype of
                when VFPNegMul_VNMLA  D[d] = FPAdd(FPNeg(D[d]), FPNeg(product64), FPSCR[]);
                when VFPNegMul_VNMLS  D[d] = FPAdd(FPNeg(D[d]), product64, FPSCR[]);
                when VFPNegMul_VNMUL  D[d] = FPNeg(product64);
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

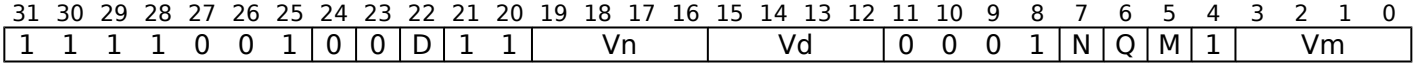
VORN (register)

Vector bitwise OR NOT (register) performs a bitwise OR NOT operation between two registers, and places the result in the destination register. The operand and result registers can be quadword or doubleword. They must all be the same size.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



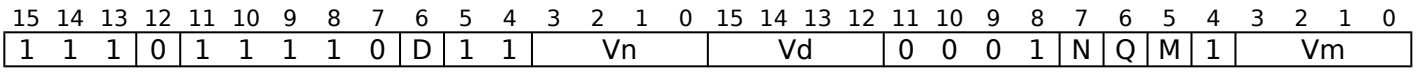
64-bit SIMD vector (Q == 0)

```
VORN{<c>}{<q>}{.<dt>} {<Dd>,} <Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VORN{<c>}{<q>}{.<dt>} {<Qd>,} <Qn>, <Qm>

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1



64-bit SIMD vector (Q == 0)

```
VORN{<c>}{<q>}{.<dt>} {<Dd>,} <Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VORN{<c>}{<q>}{.<dt>} {<Qd>,} <Qn>, <Qm>

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>	For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional. For encoding T1: see *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<dt>	An optional data type. It is ignored by assemblers, and does not affect the encoding.
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qn>	Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
<Qm>	Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        D[d+r] = D[n+r] OR NOT(D[m+r]);
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VORN (immediate)

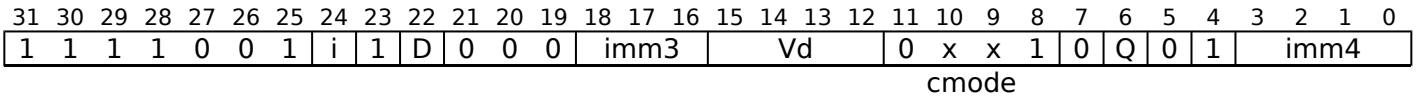
Vector Bitwise OR NOT (immediate) performs a bitwise OR between a register value and the complement of an immediate value, and returns the result into the destination vector

This is a pseudo-instruction of VORR (immediate). This means:

- The encodings in this description are named to match the encodings of VORR (immediate).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of VORR (immediate) gives the operational pseudocode for this instruction.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2) .

A1



64-bit SIMD vector (Q == 0)

```
VORN{<c>}{<q>}.I16 {<Dd>,} <Dd>, #<imm>
```

is equivalent to

```
VORR{<c>}{<q>}.I16 <Dd>, #~<imm>
```

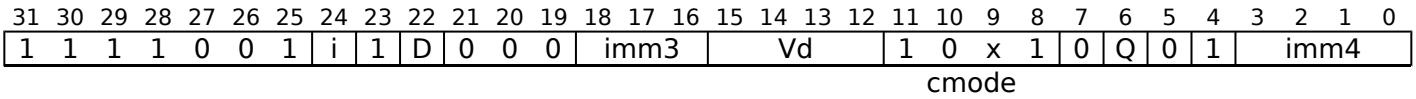
128-bit SIMD vector (Q == 1)

```
VORN{<c>}{<q>}.I16 {<Qd>,} <Qd>, #<imm>
```

is equivalent to

```
VORR{<c>}{<q>}.I16 <Qd>, #~<imm>
```

A2



64-bit SIMD vector (Q == 0)

```
VORN{<c>}{<q>}.I32 {<Dd>,} <Dd>, #<imm>
```

is equivalent to

```
VORR{<c>}{<q>}.I32 <Dd>, #~<imm>
```

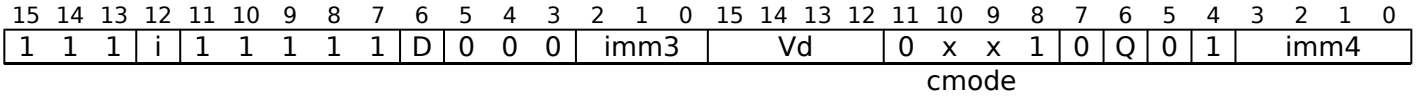
128-bit SIMD vector (Q == 1)

```
VORN{<c>}{<q>}.I32 {<Qd>,} <Qd>, #<imm>
```

is equivalent to

```
VORR{<c>}{<q>}.I32 <Qd>, #~<imm>
```

T1



64-bit SIMD vector (Q == 0)

```
VORN{<c>}{<q>}.I16 {<Dd>,} <Dd>, #<imm>
```

is equivalent to

```
VORR{<c>}{<q>}.I16 <Dd>, #~<imm>
```

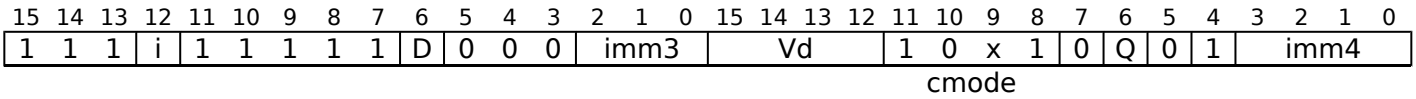
128-bit SIMD vector (Q == 1)

```
VORN{<c>}{<q>}.I16 {<Qd>,} <Qd>, #<imm>
```

is equivalent to

```
VORR{<c>}{<q>}.I16 <Qd>, #~<imm>
```

T2



64-bit SIMD vector (Q == 0)

```
VORN{<c>}{<q>}.I32 {<Dd>,} <Dd>, #<imm>
```

is equivalent to

```
VORR{<c>}{<q>}.I32 <Dd>, #~<imm>
```

128-bit SIMD vector (Q == 1)

```
VORN{<c>}{<q>}.I32 {<Qd>,} <Qd>, #<imm>
```

is equivalent to

```
VORR{<c>}{<q>}.I32 <Qd>, #~<imm>
```

Assembler Symbols

<c>	For encoding A1 and A2: see *Standard assembler syntax fields*. This encoding must be unconditional. For encoding T1 and T2: see *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<imm>	Is a constant of the specified type that is replicated to fill the destination register. For details of the range of constants available and the encoding of <imm>, see *Modified immediate constants in T32 and A32 Advanced SIMD instructions*.

Operation

The description of VORR (immediate) gives the operational pseudocode for this instruction.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

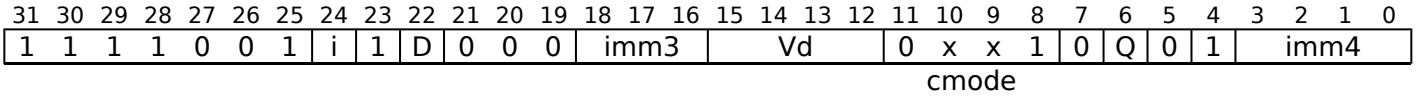
VORR (immediate)

Vector Bitwise OR (immediate) performs a bitwise OR between a register value and an immediate value, and returns the result into the destination vector.
Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

This instruction is used by the pseudo-instruction VORN (immediate).

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2) .

A1



64-bit SIMD vector (Q == 0)

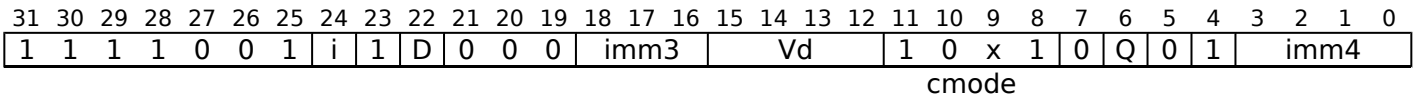
```
VORR{<c>}{<q>}.I32 {<Dd>,} <Dd>, #<imm>
```

128-bit SIMD vector (Q == 1)

```
VORR{<c>}{<q>}.I32 {<Qd>,} <Qd>, #<imm>
```

```
if cmode<0> == '0' || cmode<3:2> == '11' then SEE "VMOV (immediate)";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
imm64 = AdvSIMDExpandImm('0', cmode, i:imm3:imm4);
d = UInt(D:Vd);  regs = if Q == '0' then 1 else 2;
```

A2



64-bit SIMD vector (Q == 0)

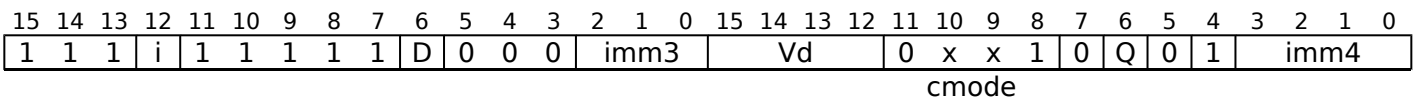
```
VORR{<c>}{<q>}.I16 {<Dd>,} <Dd>, #<imm>
```

128-bit SIMD vector (Q == 1)

```
VORR{<c>}{<q>}.I16 {<Qd>,} <Qd>, #<imm>
```

```
if cmode<0> == '0' || cmode<3:2> == '11' then SEE "VMOV (immediate)";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
imm64 = AdvSIMDExpandImm('0', cmode, i:imm3:imm4);
d = UInt(D:Vd);  regs = if Q == '0' then 1 else 2;
```

T1



64-bit SIMD vector (Q == 0)

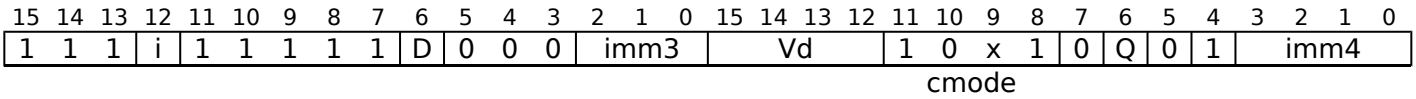
```
VORR{<c>}{<q>}.I32 {<Dd>,} <Dd>, #<imm>
```

128-bit SIMD vector (Q == 1)

```
VORR{<c>}{<q>}.I32 {<Qd>,} <Qd>, #<imm>
```

```
if cmode<0> == '0' || cmode<3:2> == '11' then SEE "VMOV (immediate)";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
imm64 = AdvSIMDExpandImm('0', cmode, i:imm3:imm4);
d = UInt(D:Vd);  regs = if Q == '0' then 1 else 2;
```

T2



64-bit SIMD vector (Q == 0)

```
VORR{<c>}{<q>}.I16 {<Dd>,} <Dd>, #<imm>
```

128-bit SIMD vector (Q == 1)

```
VORR{<c>}{<q>}.I16 {<Qd>,} <Qd>, #<imm>
```

```
if cmode<0> == '0' || cmode<3:2> == '11' then SEE "VMOV (immediate)";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
imm64 = AdvSIMDExpandImm('0', cmode, i:imm3:imm4);
d = UInt(D:Vd);  regs = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>	For encoding A1 and A2: see *Standard assembler syntax fields*. This encoding must be unconditional. For encoding T1 and T2: see *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<imm>	Is a constant of the specified type that is replicated to fill the destination register. For details of the range of constants available and the encoding of <imm>, see *Modified immediate constants in T32 and A32 Advanced SIMD instructions*.

The I8, I64, and F32 data types are permitted as pseudo-instructions, if the immediate can be represented by this instruction, and are encoded using a permitted encoding of the I16 or I32 data type.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        D[d+r] = D[d+r] OR imm64;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.

- The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VORR (register)

Vector bitwise OR (register) performs a bitwise OR operation between two registers, and places the result in the destination register. The operand and result registers can be quadword or doubleword. They must all be the same size.

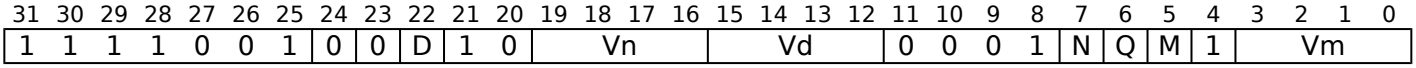
Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

This instruction is used by the alias VMOV (register, SIMD).

This instruction is used by the pseudo-instructions VRSHR (zero), and VSHR (zero).

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



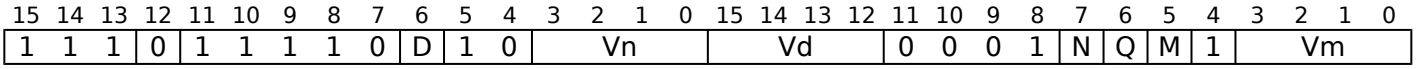
64-bit SIMD vector (Q == 0)

```
VORR{<c>}{<q>}{.<dt>} {<Dd>,} <Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VORR{<c>}{<q>}{.<dt>} {<Qd>,} <Qn>, <Qm>

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1



64-bit SIMD vector (Q == 0)

```
VORR{<c>}{<q>}{.<dt>} {<Dd>,} <Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VORR{<c>}{<q>}{.<dt>} {<Qd>,} <Qn>, <Qm>

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>	For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional. For encoding T1: see *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<dt>	An optional data type. It is ignored by assemblers, and does not affect the encoding.
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qn>	Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
<Qm>	Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Alias Conditions

Alias	Is preferred when
VMOV (register, SIMD)	`N:Vn == M:Vm`
VRSHR (zero)	`Never`
VSHR (zero)	`Never`

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        D[d+r] = D[n+r] OR D[m+r];
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

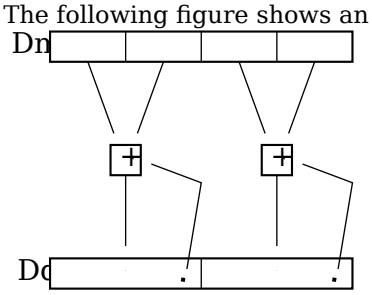
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VPADAL

Vector Pairwise Add and Accumulate Long adds adjacent pairs of elements of a vector, and accumulates the results into the elements of the destination vector.

The vectors can be doubleword or quadword. The operand elements can be 8-bit, 16-bit, or 32-bit integers. The result elements are twice the length of the operand elements.

The following figure shows an example of the operation of VPADAL doubleword operation for data type S16.



Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size		0	0	Vd				0	1	1	0	op	Q	M	0	Vm			

64-bit SIMD vector (Q == 0)

```
VPADAL{<c>}{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VPADAL{<c>}{<q>}.<dt> <Qd>, <Qm>
```

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = (op == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size		0	0	Vd				0	1	1	0	op	Q	M	0	Vm			

64-bit SIMD vector (Q == 0)

```
VPADAL{<c>}{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VPADAL{<c>}{<q>}.<dt> <Qd>, <Qm>
```

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = (op == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the vectors, encoded in “op:size”:

op	size	<dt>
0	00	S8
0	01	S16
0	10	S32
0	11	RESERVED
1	00	U8
1	01	U16
1	10	U32
1	11	RESERVED

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    h = elements DIV 2;

    for r = 0 to regs-1
        for e = 0 to h-1
            op1 = Elem[D[m+r],2*e,esize];  op2 = Elem[D[m+r],2*e+1,esize];
            result = Int(op1, unsigned) + Int(op2, unsigned);
            Elem[D[d+r],e,2*esize] = Elem[D[d+r],e,2*esize] + result;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VPADD (floating-point)

Vector Pairwise Add (floating-point) adds adjacent pairs of elements of two vectors, and places the results in the destination vector.

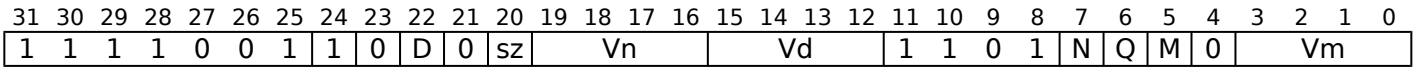
The operands and result are doubleword vectors.

The operand and result elements are floating-point numbers.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

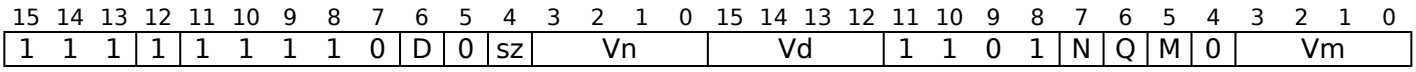


A1

```
VPADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

```
if Q == '1' then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

T1



T1

```
VPADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

```
if Q == '1' then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `sz == '1' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>	For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional. For encoding T1: see *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the vectors, encoded in “sz”:

sz	<dt>
0	F32
1	F16

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    bits(64) dest;
    h = elements DIV 2;

    for e = 0 to h-1
        Elem[dest,e,esize]   = FPAdd(Elem[D[n],2*e,esize], Elem[D[n],2*e+1,esize], StandardFPSCRValue());
        Elem[dest,e+h,esize] = FPAdd(Elem[D[m],2*e,esize], Elem[D[m],2*e+1,esize], StandardFPSCRValue());

    D[d] = dest;
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

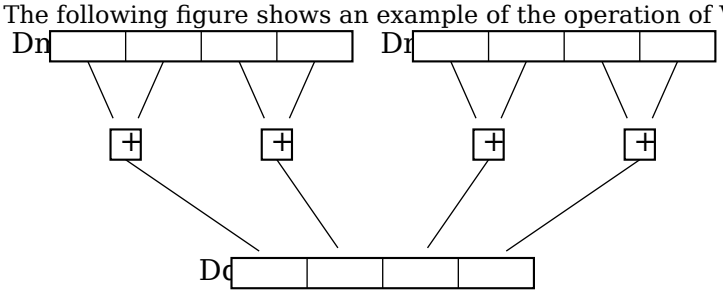
VPADD (integer)

Vector Pairwise Add (integer) adds adjacent pairs of elements of two vectors, and places the results in the destination vector.

The operands and result are doubleword vectors.

The operand and result elements must all be the same type, and can be 8-bit, 16-bit, or 32-bit integers. There is no distinction between signed and unsigned integers.

The following figure shows an example of the operation of VPADD doubleword operation for data type I16.



Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	D	size		Vn				Vd				1	0	1	1	N	Q	M	1	Vm			

A1

```
VPADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

if size == '11' || Q == '1' then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	1	0	D	size		Vn				Vd				1	0	1	1	N	Q	M	1	Vm			

T1

```
VPADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

if size == '11' || Q == '1' then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

Assembler Symbols

<c>	For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional. For encoding T1: see *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<dt>	Is the data type for the elements of the vectors, encoded in “size”:

size	<dt>
00	I8
01	I16
10	I32

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    bits(64) dest;
    h = elements DIV 2;

    for e = 0 to h-1
        Elem[dest,e,esize]   = Elem[D[n],2*e,esize] + Elem[D[n],2*e+1,esize];
        Elem[dest,e+h,esize] = Elem[D[m],2*e,esize] + Elem[D[m],2*e+1,esize];

    D[d] = dest;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

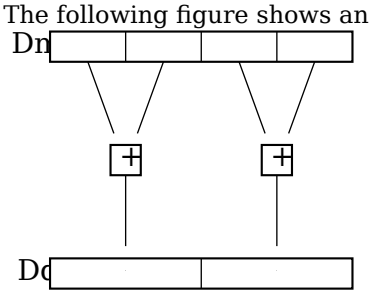
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VPADDL

Vector Pairwise Add Long adds adjacent pairs of elements of two vectors, and places the results in the destination vector.
The vectors can be doubleword or quadword. The operand elements can be 8-bit, 16-bit, or 32-bit integers. The result elements are twice the length of the operand elements.
The following figure shows an example of the operation of VPADDL doubleword operation for data type S16.



Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size		0	0	Vd				0	0	1	0	op	Q	M	0	Vm			

64-bit SIMD vector (Q == 0)

```
VPADDL{<c>}{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VPADDL{<c>}{<q>}.<dt> <Qd>, <Qm>
```

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = (op == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size		0	0	Vd				0	0	1	0	op	Q	M	0	Vm			

64-bit SIMD vector (Q == 0)

```
VPADDL{<c>}{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VPADDL{<c>}{<q>}.<dt> <Qd>, <Qm>
```

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = (op == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the vectors, encoded in “op:size”:

op	size	<dt>
0	00	S8
0	01	S16
0	10	S32
0	11	RESERVED
1	00	U8
1	01	U16
1	10	U32
1	11	RESERVED

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    h = elements DIV 2;

    for r = 0 to regs-1
        for e = 0 to h-1
            op1 = Elem[D[m+r],2*e,esize];  op2 = Elem[D[m+r],2*e+1,esize];
            result = Int(op1, unsigned) + Int(op2, unsigned);
            Elem[D[d+r],e,2*esize] = result<2*esize-1:0>;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

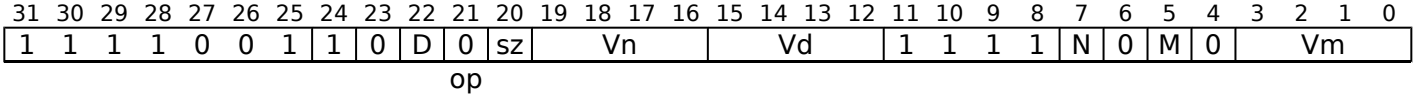
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VPMAX (floating-point)

Vector Pairwise Maximum compares adjacent pairs of elements in two doubleword vectors, and copies the larger of each pair into the corresponding element in the destination doubleword vector.
Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

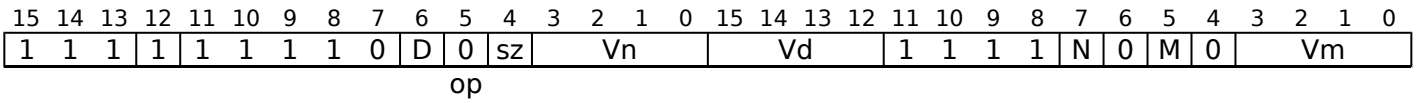


A1

```
VPMAX{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

```
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
maximum = (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);
```

T1



T1

```
VPMAX{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

```
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
maximum = (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);
```

CONSTRAINED UNPREDICTABLE behavior

If `sz == '1' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>	For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional. For encoding T1: see *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the vectors, encoded in “sz”:

sz	<dt>
0	F32
1	F16

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    bits(64) dest;
    h = elements DIV 2;

    for e = 0 to h-1
        op1 = Elem[D[n],2*e,esize];  op2 = Elem[D[n],2*e+1,esize];
        Elem[dest,e,esize] = if maximum then FPMax(op1,op2,StandardFPSCRValue()) else FPMin(op1,op2,Stand
        op1 = Elem[D[m],2*e,esize];  op2 = Elem[D[m],2*e+1,esize];
        Elem[dest,e+h,esize] = if maximum then FPMax(op1,op2,StandardFPSCRValue()) else FPMin(op1,op2,Sta

    D[d] = dest;
```

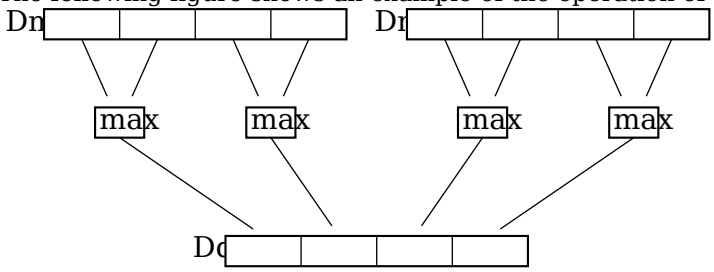
Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VPMAX (integer)

Vector Pairwise Maximum compares adjacent pairs of elements in two doubleword vectors, and copies the larger of each pair into the corresponding element in the destination doubleword vector.

The following figure shows an example of the operation of VPMAX doubleword operation for data type S16 or U16.



Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	0	D	size		Vn				Vd				1	0	1	0	N	0	M	0	Vm			
																											op				

A1

```
VPMAX{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

```
if size == '11' then UNDEFINED;
maximum = (op == '0');  unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	0	D	size		Vn				Vd				1	0	1	0	N	0	M	0	Vm			
																											op				

T1

```
VPMAX{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

```
if size == '11' then UNDEFINED;
maximum = (op == '0');  unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

Assembler Symbols

<c>	For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional. For encoding T1: see *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<dt>	Is the data type for the elements of the operands, encoded in “U:size”:

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	00	U8
1	01	U16
1	10	U32

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    bits(64) dest;
    h = elements DIV 2;

    for e = 0 to h-1
        op1 = Int(Elem[D[n],2*e,esize], unsigned);
        op2 = Int(Elem[D[n],2*e+1,esize], unsigned);
        result = if maximum then Max(op1,op2) else Min(op1,op2);
        Elem[dest,e,esize] = result<esize-1:0>;
        op1 = Int(Elem[D[m],2*e,esize], unsigned);
        op2 = Int(Elem[D[m],2*e+1,esize], unsigned);
        result = if maximum then Max(op1,op2) else Min(op1,op2);
        Elem[dest,e+h,esize] = result<esize-1:0>;

    D[d] = dest;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

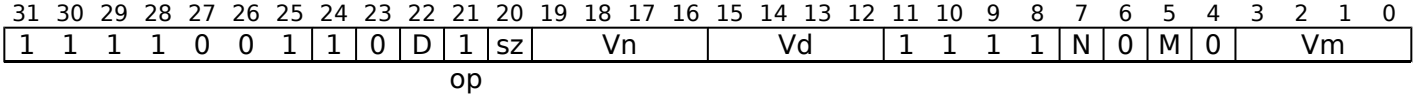
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VPMIN (floating-point)

Vector Pairwise Minimum compares adjacent pairs of elements in two doubleword vectors, and copies the smaller of each pair into the corresponding element in the destination doubleword vector.
Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

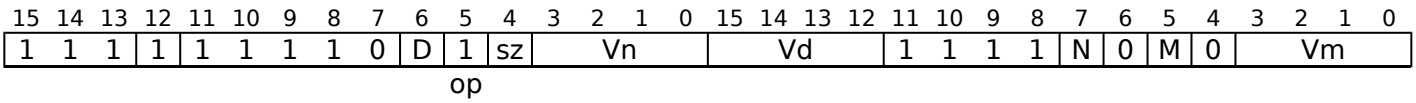


A1

```
VPMIN{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

```
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
maximum = (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);
```

T1



T1

```
VPMIN{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

```
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
maximum = (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);
```

CONSTRAINED UNPREDICTABLE behavior

If `sz == '1' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>	For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional. For encoding T1: see *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the vectors, encoded in “sz”:

sz	<dt>
0	F32
1	F16

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    bits(64) dest;
    h = elements DIV 2;

    for e = 0 to h-1
        op1 = Elem[D[n],2*e,esize];  op2 = Elem[D[n],2*e+1,esize];
        Elem[dest,e,esize] = if maximum then FPMax(op1,op2,StandardFPSCRValue()) else FPMin(op1,op2,Stand
        op1 = Elem[D[m],2*e,esize];  op2 = Elem[D[m],2*e+1,esize];
        Elem[dest,e+h,esize] = if maximum then FPMax(op1,op2,StandardFPSCRValue()) else FPMin(op1,op2,Sta

    D[d] = dest;
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

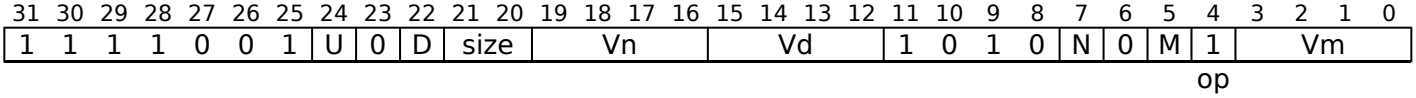
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VPMIN (integer)

Vector Pairwise Minimum compares adjacent pairs of elements in two doubleword vectors, and copies the smaller of each pair into the corresponding element in the destination doubleword vector.
Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

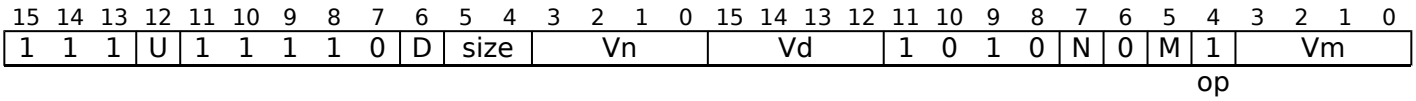


A1

```
VPMIN{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

```
if size == '11' then UNDEFINED;
maximum = (op == '0');  unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

T1



T1

```
VPMIN{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

```
if size == '11' then UNDEFINED;
maximum = (op == '0');  unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

Assembler Symbols

<c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.
For encoding T1: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the operands, encoded in “U:size”:

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	00	U8
1	01	U16
1	10	U32

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    bits(64) dest;
    h = elements DIV 2;

    for e = 0 to h-1
        op1 = Int(Elem[D[n],2*e,esize], unsigned);
        op2 = Int(Elem[D[n],2*e+1,esize], unsigned);
        result = if maximum then Max(op1,op2) else Min(op1,op2);
        Elem[dest,e,esize] = result<esize-1:0>;
        op1 = Int(Elem[D[m],2*e,esize], unsigned);
        op2 = Int(Elem[D[m],2*e+1,esize], unsigned);
        result = if maximum then Max(op1,op2) else Min(op1,op2);
        Elem[dest,e+h,esize] = result<esize-1:0>;

    D[d] = dest;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VPOP

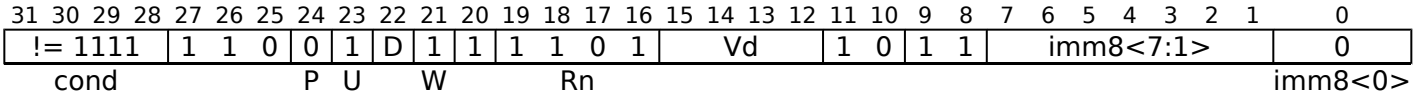
Pop SIMD&FP registers from Stack loads multiple consecutive Advanced SIMD and floating-point register file registers from the stack

This is an alias of VLDM, VLDMDB, VLDMIA. This means:

- The encodings in this description are named to match the encodings of VLDM, VLDMDB, VLDMIA.
- The description of VLDM, VLDMDB, VLDMIA gives the operational pseudocode for this instruction.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2) .

A1



Increment After

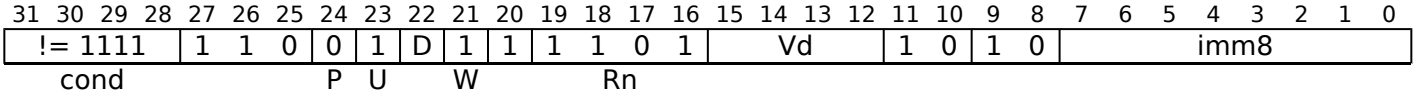
```
VPOP{<c>}{<q>}{.<size>} <dreglist>
```

is equivalent to

```
VLDM{<c>}{<q>}{.<size>} SP!, <dreglist>
```

and is always the preferred disassembly.

A2



Increment After

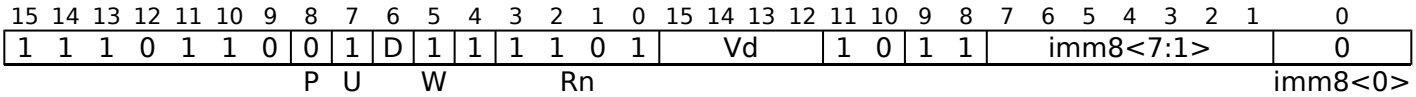
```
VPOP{<c>}{<q>}{.<size>} <sreglist>
```

is equivalent to

```
VLDM{<c>}{<q>}{.<size>} SP!, <sreglist>
```

and is always the preferred disassembly.

T1



Increment After

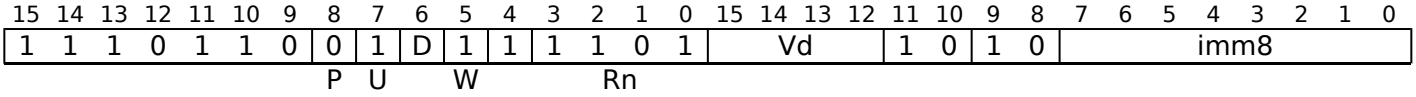
```
VPOP{<c>}{<q>}{.<size>} <dreglist>
```

is equivalent to

```
VLDM{<c>}{<q>}{.<size>} SP!, <dreglist>
```

and is always the preferred disassembly.

T2



Increment After

```
VPOP{<c>}{<q>}{.<size>} <sreglist>
```

is equivalent to

```
VLDM{<c>}{<q>}{.<size>} SP!, <sreglist>
```

and is always the preferred disassembly.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<size>	An optional data size specifier. If present, it must be equal to the size in bits, 32 or 64, of the registers being transferred.
<sreglist>	Is the list of consecutively numbered 32-bit SIMD&FP registers to be transferred. The first register in the list is encoded in "Vd:D", and "imm8" is set to the number of registers in the list. The list must contain at least one register.
<dreglist>	Is the list of consecutively numbered 64-bit SIMD&FP registers to be transferred. The first register in the list is encoded in "D:Vd", and "imm8" is set to twice the number of registers in the list. The list must contain at least one register, and must not contain more than 16 registers.

Operation

The description of VLDM, VLDMDB, VLDMIA gives the operational pseudocode for this instruction.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VPUSH

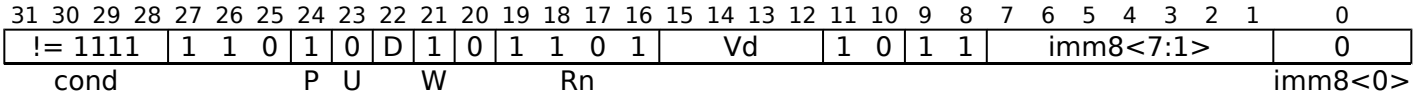
Push SIMD&FP registers to Stack stores multiple consecutive registers from the Advanced SIMD and floating-point register file to the stack

This is an alias of VSTM, VSTMDB, VSTMIA. This means:

- The encodings in this description are named to match the encodings of VSTM, VSTMDB, VSTMIA.
- The description of VSTM, VSTMDB, VSTMIA gives the operational pseudocode for this instruction.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2) .

A1



Decrement Before

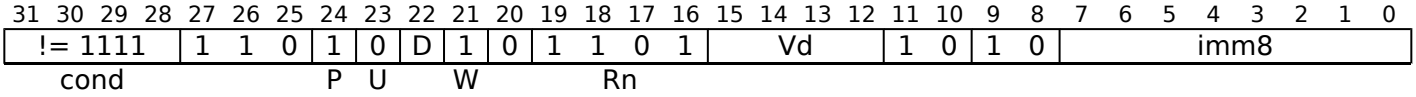
```
VPUSH{<c>}{<q>}{.<size>} <dreglist>
```

is equivalent to

```
VSTMDB{<c>}{<q>}{.<size>} SP!, <dreglist>
```

and is always the preferred disassembly.

A2



Decrement Before

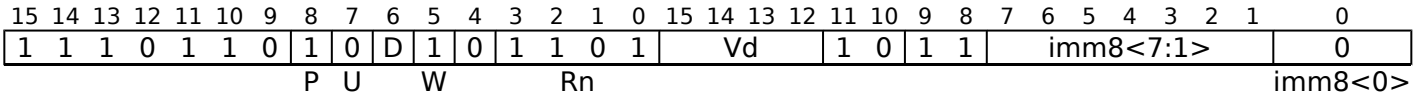
```
VPUSH{<c>}{<q>}{.<size>} <sreglist>
```

is equivalent to

```
VSTMDB{<c>}{<q>}{.<size>} SP!, <sreglist>
```

and is always the preferred disassembly.

T1



Decrement Before

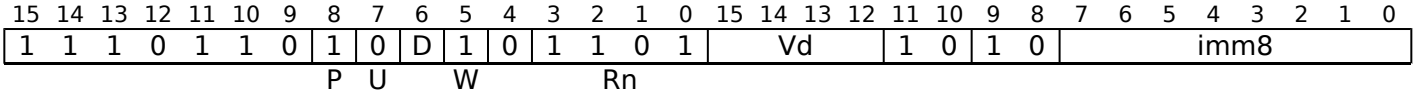
```
VPUSH{<c>}{<q>}{.<size>} <dreglist>
```

is equivalent to

```
VSTMDB{<c>}{<q>}{.<size>} SP!, <dreglist>
```

and is always the preferred disassembly.

T2



Decrement Before

```
VPUSH{<c>}{<q>}{.<size>} <sreglist>
```

is equivalent to

```
VSTMDB{<c>}{<q>}{.<size>} SP!, <sreglist>
```

and is always the preferred disassembly.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<size>	An optional data size specifier. If present, it must be equal to the size in bits, 32 or 64, of the registers being transferred.
<sreglist>	Is the list of consecutively numbered 32-bit SIMD&FP registers to be transferred. The first register in the list is encoded in "Vd:D", and "imm8" is set to the number of registers in the list. The list must contain at least one register.
<dreglist>	Is the list of consecutively numbered 64-bit SIMD&FP registers to be transferred. The first register in the list is encoded in "D:Vd", and "imm8" is set to twice the number of registers in the list. The list must contain at least one register, and must not contain more than 16 registers.

Operation

The description of VSTM, VSTMDB, VSTMIA gives the operational pseudocode for this instruction.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VQABS

Vector Saturating Absolute takes the absolute value of each element in a vector, and places the results in the destination vector.
If any of the results overflow, they are saturated. The cumulative saturation bit, *FPSCR*.QC, is set if saturation occurs. For details see *Pseudocode details of saturation*.
Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size		0	0	Vd				0	1	1	1	0	Q	M	0	Vm			

64-bit SIMD vector (Q == 0)

```
VQABS{<c>}{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VQABS{<c>}{<q>}.<dt> <Qd>, <Qm>
```

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size		0	0	Vd				0	1	1	1	0	Q	M	0	Vm			

64-bit SIMD vector (Q == 0)

```
VQABS{<c>}{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VQABS{<c>}{<q>}.<dt> <Qd>, <Qm>
```

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.
For encoding T1: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the vectors, encoded in “size”:

size	<dt>
00	S8
01	S16
10	S32
11	RESERVED

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            result = Abs(SInt(Elem[D[m+r],e,esize]));
            (Elem[D[d+r],e,esize], sat) = SignedSatQ(result, esize);
            if sat then FPSCR.QC = '1';
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

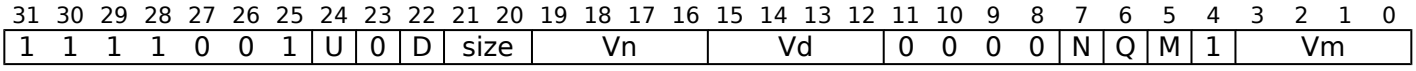
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VQADD

Vector Saturating Add adds the values of corresponding elements of two vectors, and places the results in the destination vector.
If any of the results overflow, they are saturated. The cumulative saturation bit, *FPSCR*.QC, is set if saturation occurs. For details see *Pseudocode details of saturation*.
Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



64-bit SIMD vector (Q == 0)

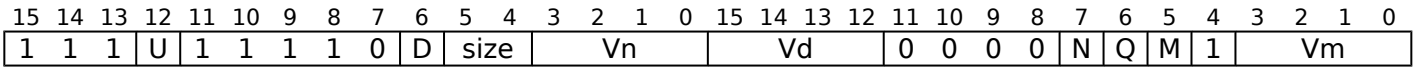
```
VQADD{<c>}{<q>}.<dt> {<Dd>,} <Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VQADD{<c>}{<q>}.<dt> {<Qd>,} <Qn>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1



64-bit SIMD vector (Q == 0)

```
VQADD{<c>}{<q>}.<dt> {<Dd>,} <Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VQADD{<c>}{<q>}.<dt> {<Qd>,} <Qn>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>	For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional. For encoding T1: see *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<dt>	Is the data type for the elements of the vectors, encoded in “U:size”:

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
0	11	S64
1	00	U8
1	01	U16
1	10	U32
1	11	U64

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            sum = Int(Elem[D[n+r],e,esize], unsigned) + Int(Elem[D[m+r],e,esize], unsigned);
            (Elem[D[d+r],e,esize], sat) = SatQ(sum, esize, unsigned);
            if sat then FPSCR.QC = '1';
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VQDMLAL

Vector Saturating Doubling Multiply Accumulate Long multiplies corresponding elements in two doubleword vectors, doubles the products, and accumulates the results into the elements of a quadword vector.

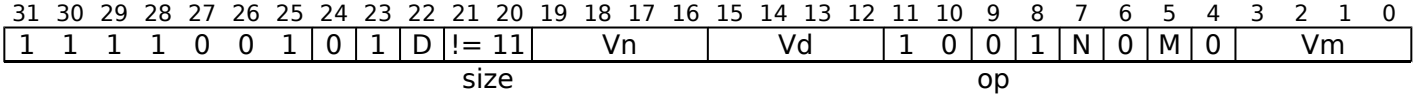
The second operand can be a scalar instead of a vector. For more information about scalars see *Advanced SIMD scalars*.

If any of the results overflow, they are saturated. The cumulative saturation bit, *FPSCR*.QC, is set if saturation occurs. For details see *Pseudocode details of saturation*.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to 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

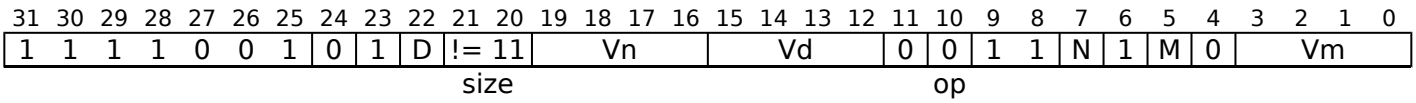


A1

```
VQDMLAL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>
```

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
add = (op == '0');
scalar_form = FALSE;  d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
esize = 8 << UInt(size);  elements = 64 DIV esize;
```

A2

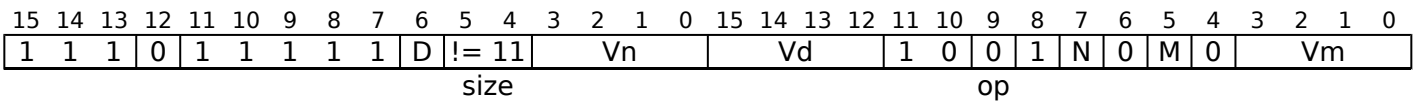


A2

```
VQDMLAL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>[<index>]
```

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
add = (op == '0');
scalar_form = TRUE;  d = UInt(D:Vd);  n = UInt(N:Vn);
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```

T1

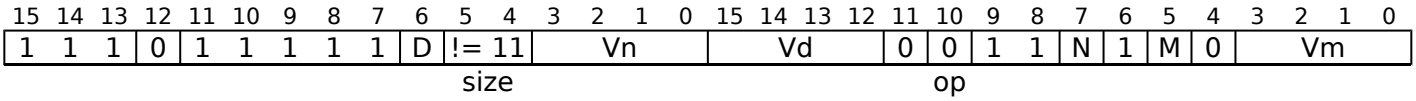


T1

```
VQDMLAL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>
```

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
add = (op == '0');
scalar_form = FALSE;  d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
esize = 8 << UInt(size);  elements = 64 DIV esize;
```

T2



T2

```
VQDMLAL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>[<index>]
```

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
add = (op == '0');
scalar_form = TRUE;  d = UInt(D:Vd);  n = UInt(N:Vn);
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```

Related encodings: See *Advanced SIMD data-processing* for the T32 instruction set, or *Advanced SIMD data-processing* for the A32 instruction set.

Assembler Symbols

<c> For encoding A1 and A2: see *Standard assembler syntax fields*. This encoding must be unconditional.

For encoding T1 and T2: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the operands, encoded in “size”:

size	<dt>
01	S16
10	S32

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> For encoding A1 and T1: is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

For encoding A2 and T2: is the 64-bit name of the second SIMD&FP source register, encoded in the "Vm<2:0>" field when <dt> is S16, otherwise the "Vm" field.

<index> Is the element index in the range 0 to 3, encoded in the "M:Vm<3>" field when <dt> is S16, otherwise in range 0 to 1, encoded in the "M" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    if scalar_form then op2 = SInt(Elem[Din[m],index,esize]);
    for e = 0 to elements-1
        if !scalar_form then op2 = SInt(Elem[Din[m],e,esize]);
        op1 = SInt(Elem[Din[n],e,esize]);
        // The following only saturates if both op1 and op2 equal -(2^(esize-1))
        (product, sat1) = SignedSatQ(2*op1*op2, 2*esize);
        if add then
            result = SInt(Elem[Qin[d>>1],e,2*esize]) + SInt(product);
        else
            result = SInt(Elem[Qin[d>>1],e,2*esize]) - SInt(product);
        (Elem[Q[d>>1],e,2*esize], sat2) = SignedSatQ(result, 2*esize);
        if sat1 || sat2 then FPSCR.QC = '1';
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VQDMLSL

Vector Saturating Doubling Multiply Subtract Long multiplies corresponding elements in two doubleword vectors, subtracts double the products from corresponding elements of a quadword vector, and places the results in the same quadword vector.

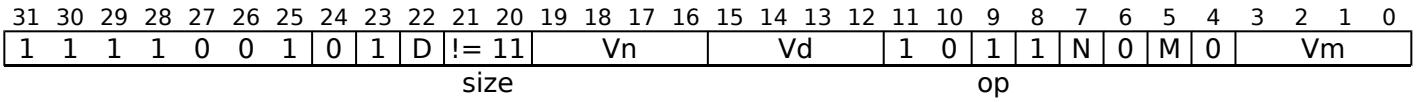
The second operand can be a scalar instead of a vector. For more information about scalars see *Advanced SIMD scalars*.

If any of the results overflow, they are saturated. The cumulative saturation bit, *FPSCR*.QC, is set if saturation occurs. For details see *Pseudocode details of saturation*.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2) .

A1

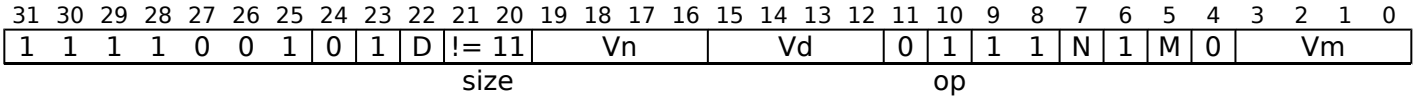


A1

```
VQDMLSL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>
```

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
add = (op == '0');
scalar_form = FALSE;  d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
esize = 8 << UInt(size);  elements = 64 DIV esize;
```

A2

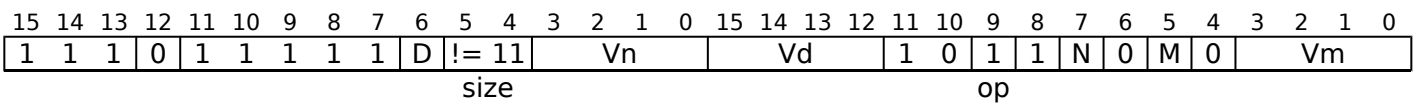


A2

```
VQDMLSL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>[<index>]
```

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
add = (op == '0');
scalar_form = TRUE;  d = UInt(D:Vd);  n = UInt(N:Vn);
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```

T1

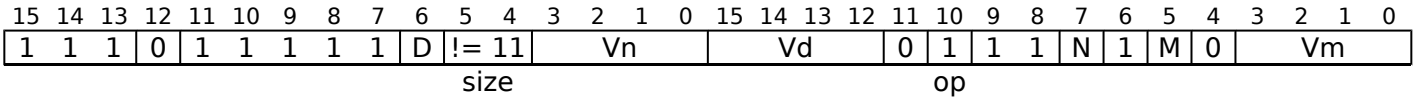


T1

```
VQDMLSL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>
```

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
add = (op == '0');
scalar_form = FALSE;  d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
esize = 8 << UInt(size);  elements = 64 DIV esize;
```

T2



T2

```
VQDMLSL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>[<index>]
```

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
add = (op == '0');
scalar_form = TRUE;  d = UInt(D:Vd);  n = UInt(N:Vn);
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```

Related encodings: See *Advanced SIMD data-processing* for the T32 instruction set, or *Advanced SIMD data-processing* for the A32 instruction set.

Assembler Symbols

<c> For encoding A1 and A2: see *Standard assembler syntax fields*. This encoding must be unconditional.

For encoding T1 and T2: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the operands, encoded in “size”:

size	<dt>
01	S16
10	S32

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> For encoding A1 and T1: is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

For encoding A2 and T2: is the 64-bit name of the second SIMD&FP source register, encoded in the "Vm<2:0>" field when <dt> is S16, otherwise the "Vm" field.

<index> Is the element index in the range 0 to 3, encoded in the "M:Vm<3>" field when <dt> is S16, otherwise in range 0 to 1, encoded in the "M" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    if scalar_form then op2 = SInt(Elem[Din[m],index,esize]);
    for e = 0 to elements-1
        if !scalar_form then op2 = SInt(Elem[Din[m],e,esize]);
        op1 = SInt(Elem[Din[n],e,esize]);
        // The following only saturates if both op1 and op2 equal -(2^(esize-1))
        (product, sat1) = SignedSatQ(2*op1*op2, 2*esize);
        if add then
            result = SInt(Elem[Qin[d>>1],e,2*esize]) + SInt(product);
        else
            result = SInt(Elem[Qin[d>>1],e,2*esize]) - SInt(product);
        (Elem[Q[d>>1],e,2*esize], sat2) = SignedSatQ(result, 2*esize);
        if sat1 || sat2 then FPSCR.QC = '1';
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VQDMULH

Vector Saturating Doubling Multiply Returning High Half multiplies corresponding elements in two vectors, doubles the results, and places the most significant half of the final results in the destination vector. The results are truncated, for rounded results see *VQRDMULH*.

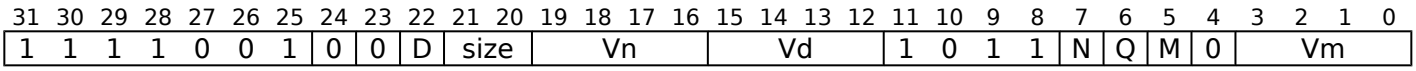
The second operand can be a scalar instead of a vector. For more information about scalars see *Advanced SIMD scalars*.

If any of the results overflow, they are saturated. The cumulative saturation bit, *FPSCR*.QC, is set if saturation occurs. For details see *Pseudocode details of saturation*.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2) .

A1



64-bit SIMD vector (Q == 0)

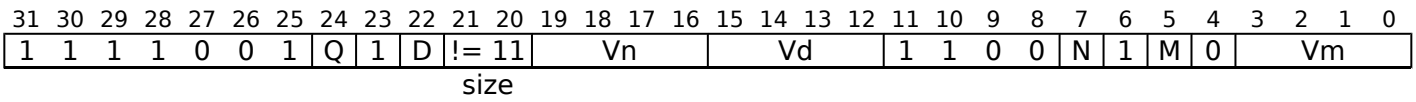
```
VQDMULH{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VQDMULH{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '00' || size == '11' then UNDEFINED;
scalar_form = FALSE;  esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

A2



64-bit SIMD vector (Q == 0)

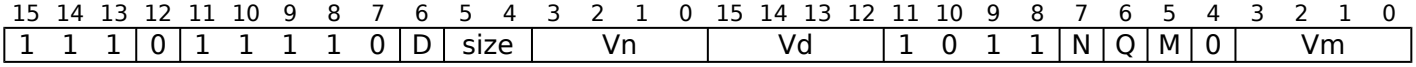
```
VQDMULH{<c>}{<q>}.<dt> {<Dd>,} <Dn>, <Dm[x]>
```

128-bit SIMD vector (Q == 1)

```
VQDMULH{<c>}{<q>}.<dt> {<Qd>,} <Qn>, <Dm[x]>
```

```
if size == '11' then SEE "Related encodings";
if size == '00' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
scalar_form = TRUE;  d = UInt(D:Vd);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```

T1



64-bit SIMD vector (Q == 0)

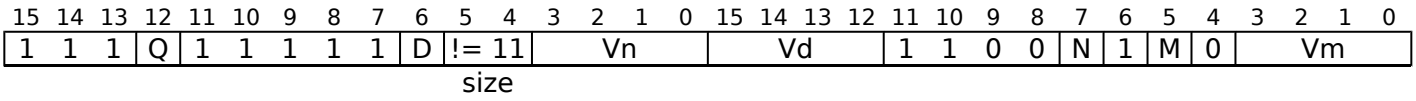
```
VQDMULH{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VQDMULH{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '00' || size == '11' then UNDEFINED;
scalar_form = FALSE;  esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T2



64-bit SIMD vector (Q == 0)

```
VQDMULH{<c>}{<q>}.<dt> {<Dd>,} <Dn>, <Dm[x]>
```

128-bit SIMD vector (Q == 1)

```
VQDMULH{<c>}{<q>}.<dt> {<Qd>,} <Qn>, <Dm[x]>
```

```
if size == '11' then SEE "Related encodings";
if size == '00' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
scalar_form = TRUE;  d = UInt(D:Vd);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```

Related encodings: See *Advanced SIMD data-processing* for the T32 instruction set, or *Advanced SIMD data-processing* for the A32 instruction set.

Assembler Symbols

<c> For encoding A1 and A2: see *Standard assembler syntax fields*. This encoding must be unconditional.

For encoding T1 and T2: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the operands, encoded in “size”:

size	<dt>
01	S16
10	S32

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm[x]> Is the 64-bit name of the second SIMD&FP source register holding the scalar. If <dt> is S16, Dm is restricted to D0-D7. Dm is encoded in "Vm<2:0>", and x is encoded in "M:Vm<3>". If <dt> is S32, Dm is restricted to D0-D15. Dm is encoded in "Vm", and x is encoded in "M".

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    if scalar_form then op2 = SInt(Elem[D[m],index,esize]);
    for r = 0 to regs-1
        for e = 0 to elements-1
            if !scalar_form then op2 = SInt(Elem[D[m+r],e,esize]);
            op1 = SInt(Elem[D[n+r],e,esize]);
            // The following only saturates if both op1 and op2 equal -(2^(esize-1))
            (result, sat) = SignedSatQ((2*op1*op2) >> esize, esize);
            Elem[D[d+r],e,esize] = result;
            if sat then FPSCR.QC = '1';
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VQDMULL

Vector Saturating Doubling Multiply Long multiplies corresponding elements in two doubleword vectors, doubles the products, and places the results in a quadword vector.

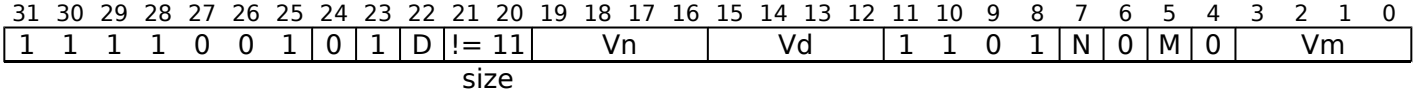
The second operand can be a scalar instead of a vector. For more information about scalars see *Advanced SIMD scalars*.

If any of the results overflow, they are saturated. The cumulative saturation bit, *FPSCR*.QC, is set if saturation occurs. For details see *Pseudocode details of saturation*.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2) .

A1

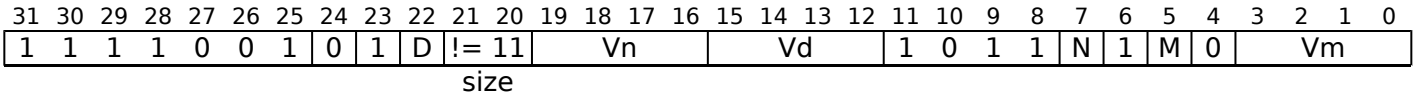


A1

```
VQDMULL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>
```

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
scalar_form = FALSE;  d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
esize = 8 << UInt(size);  elements = 64 DIV esize;
```

A2

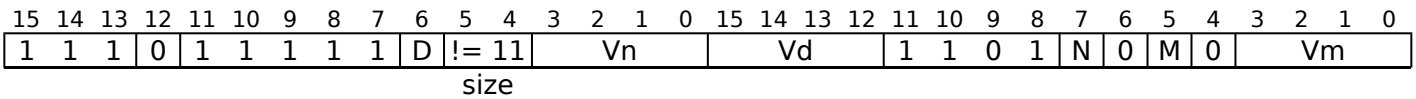


A2

```
VQDMULL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm[x]>
```

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
scalar_form = TRUE;  d = UInt(D:Vd);  n = UInt(N:Vn);
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```

T1

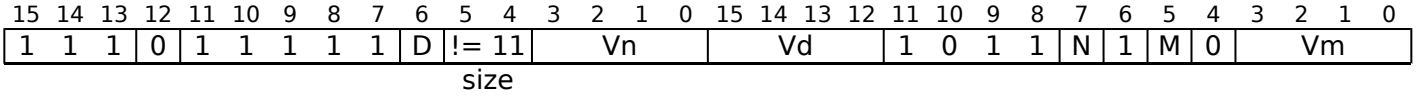


T1

```
VQDMULL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>
```

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
scalar_form = FALSE;  d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
esize = 8 << UInt(size);  elements = 64 DIV esize;
```

T2



T2

```
VQDMULL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm[x]>
```

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
scalar_form = TRUE;  d = UInt(D:Vd);  n = UInt(N:Vn);
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```

Related encodings: See *Advanced SIMD data-processing* for the T32 instruction set, or *Advanced SIMD data-processing* for the A32 instruction set.

Assembler Symbols

<c> For encoding A1 and A2: see *Standard assembler syntax fields*. This encoding must be unconditional.

For encoding T1 and T2: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the operands, encoded in “size”:

size	<dt>
01	S16
10	S32

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm[x]> Is the 64-bit name of the second SIMD&FP source register holding the scalar. If <dt> is S16, Dm is restricted to D0-D7. Dm is encoded in "Vm<2:0>", and x is encoded in "M:Vm<3>". If <dt> is S32, Dm is restricted to D0-D15. Dm is encoded in "Vm", and x is encoded in "M".

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    if scalar_form then op2 = SInt(Elem[Din[m],index,esize]);
    for e = 0 to elements-1
        if !scalar_form then op2 = SInt(Elem[Din[m],e,esize]);
        op1 = SInt(Elem[Din[n],e,esize]);
        // The following only saturates if both op1 and op2 equal -(2^(esize-1))
        (product, sat) = SignedSatQ(2*op1*op2, 2*esize);
        Elem[Q[d>>1],e,2*esize] = product;
        if sat then FPSCR.QC = '1';
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VQMOVN, VQMOVUN

Vector Saturating Move and Narrow copies each element of the operand vector to the corresponding element of the destination vector.

The operand is a quadword vector. The elements can be any one of:

- 16-bit, 32-bit, or 64-bit signed integers.
- 16-bit, 32-bit, or 64-bit unsigned integers.

The result is a doubleword vector. The elements are half the length of the operand vector elements. If the operand is unsigned, the results are unsigned. If the operand is signed, the results can be signed or unsigned.

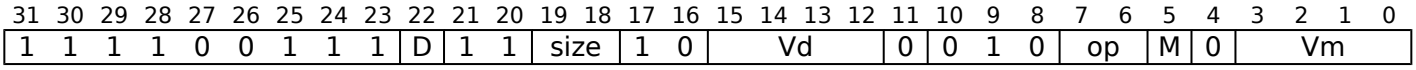
If any of the results overflow, they are saturated. The cumulative saturation bit, *FPSCR*.QC, is set if saturation occurs. For details see *Pseudocode details of saturation*.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

This instruction is used by the pseudo-instructions VQRSHRN (zero), VQRSHRUN (zero), VQSHRN (zero), and VQSHRUN (zero).

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



Signed result (op == 1x)

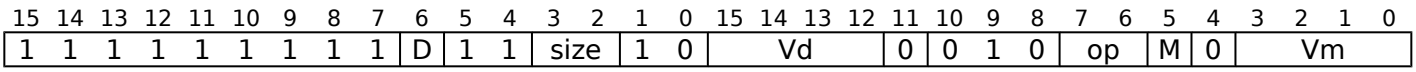
```
VQMOVN{<c>}{<q>}.<dt> <Dd>, <Qm>
```

Unsigned result (op == 01)

```
VQMOVUN{<c>}{<q>}.<dt> <Dd>, <Qm>
```

```
if op == '00' then SEE "VMOVN";
if size == '11' || Vm<0> == '1' then UNDEFINED;
src_unsigned = (op == '11');  dest_unsigned = (op<0> == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);
```

T1



Signed result (op == 1x)

```
VQMOVN{<c>}{<q>}.<dt> <Dd>, <Qm>
```

Unsigned result (op == 01)

```
VQMOVUN{<c>}{<q>}.<dt> <Dd>, <Qm>
```

```
if op == '00' then SEE "VMOVN";
if size == '11' || Vm<0> == '1' then UNDEFINED;
src_unsigned = (op == '11');  dest_unsigned = (op<0> == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);
```

Assembler Symbols

<c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<dt> For the signed result variant: is the data type for the elements of the operand, encoded in “op<0>:size”:

op<0>	size	<dt>
0	00	S16
0	01	S32
0	10	S64
0	11	RESERVED
1	00	U16
1	01	U32
1	10	U64
1	11	RESERVED

For the unsigned result variant: is the data type for the elements of the operand, encoded in “size”:

size	<dt>
00	S16
01	S32
10	S64
11	RESERVED

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for e = 0 to elements-1
        operand = Int(Elem[Qin[m>>1],e,2*esize], src_unsigned);
        (Elem[D[d],e,esize], sat) = SatQ(operand, esize, dest_unsigned);
        if sat then FPSCR.QC = '1';
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

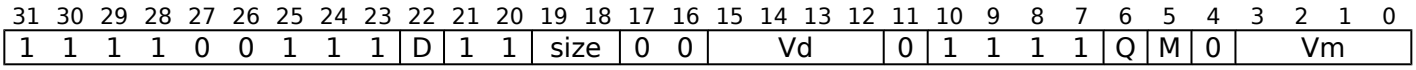
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VQNEG

Vector Saturating Negate negates each element in a vector, and places the results in the destination vector.
If any of the results overflow, they are saturated. The cumulative saturation bit, *FPSCR*.QC, is set if saturation occurs. For details see *Pseudocode details of saturation*.
Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



64-bit SIMD vector (Q == 0)

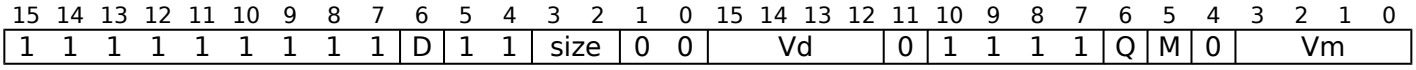
```
VQNEG{<c>}{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VQNEG{<c>}{<q>}.<dt> <Qd>, <Qm>
```

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1



64-bit SIMD vector (Q == 0)

```
VQNEG{<c>}{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VQNEG{<c>}{<q>}.<dt> <Qd>, <Qm>
```

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>	For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional. For encoding T1: see *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<dt>	Is the data type for the elements of the vectors, encoded in “size”:

size	<dt>
00	S8
01	S16
10	S32
11	RESERVED

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            result = -SInt(Elem[D[m+r],e,esize]);
            (Elem[D[d+r],e,esize], sat) = SignedSatQ(result, esize);
            if sat then FPSCR.QC = '1';
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VQRDMLAH

Vector Saturating Rounding Doubling Multiply Accumulate Returning High Half. This instruction multiplies the vector elements of the first source SIMD&FP register with either the corresponding vector elements of the second source SIMD&FP register or the value of a vector element of the second source SIMD&FP register, without saturating the multiply results, doubles the results, and accumulates the most significant half of the final results with the vector elements of the destination SIMD&FP register. The results are rounded.

If any of the results overflow, they are saturated. The cumulative saturation bit, *FPSCR*.QC, is set if saturation occurs. For details see *Pseudocode details of saturation*.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2) .

A1

(FEAT_RDM)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	0	D	size		Vn				Vd				1	0	1	1	N	Q	M	1	Vm			

64-bit SIMD vector (Q == 0)

```
VQRDMLAH{<q>}.<dt> <Dd>, <Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VQRDMLAH{<q>}.<dt> <Qd>, <Qn>, <Qm>
```

```
if !HaveQRDMLAHExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '00' || size == '11' then UNDEFINED;
add = TRUE;  scalar_form = FALSE;  esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

A2

(FEAT_RDM)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	Q	1	D	!= 11		Vn				Vd				1	1	1	0	N	1	M	0	Vm			
										size																					

64-bit SIMD vector (Q == 0)

```
VQRDMLAH{<q>}.<dt> <Dd>, <Dn>, <Dm[x]>
```

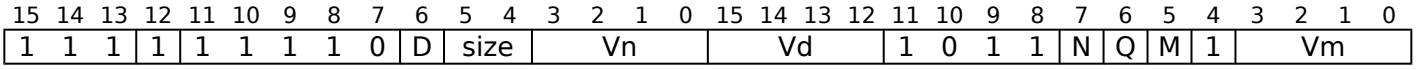
128-bit SIMD vector (Q == 1)

```
VQRDMLAH{<q>}.<dt> <Qd>, <Qn>, <Dm[x]>
```

```
if !HaveQRDMLAHExt() then UNDEFINED;
if size == '11' then SEE "Related encodings";
if size == '00' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
add = TRUE;  scalar_form = TRUE;  d = UInt(D:Vd);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```

T1

(FEAT_RDM)



64-bit SIMD vector (Q == 0)

```
VQRDMLAH{<q>}.<dt> <Dd>, <Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VQRDMLAH{<q>}.<dt> <Qd>, <Qn>, <Qm>
```

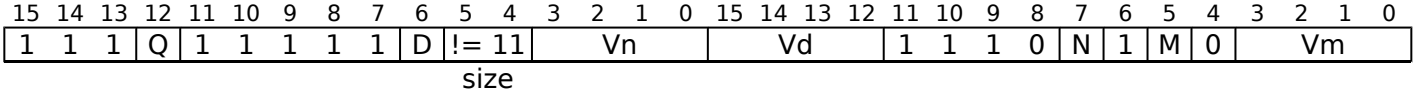
```
if !HaveQRDMLAHExt() then UNDEFINED;
if InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '00' || size == '11' then UNDEFINED;
add = TRUE;  scalar_form = FALSE;  esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T2
(FEAT_RDM)



64-bit SIMD vector (Q == 0)

```
VQRDMLAH{<q>}.<dt> <Dd>, <Dn>, <Dm[x]>
```

128-bit SIMD vector (Q == 1)

```
VQRDMLAH{<q>}.<dt> <Qd>, <Qn>, <Dm[x]>
```

```
if !HaveQRDMLAHExt() then UNDEFINED;
if InITBlock() then UNPREDICTABLE;
if size == '11' then SEE "Related encodings";
if size == '00' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
add = TRUE;  scalar_form = TRUE;  d = UInt(D:Vd);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Related encodings: See *Advanced SIMD data-processing* for the T32 instruction set, or *Advanced SIMD data-processing* for the A32 instruction set.

Assembler Symbols

<q> See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the operands, encoded in “size”:

size	<dt>
01	S16
10	S32

<Qd> Is the 128-bit name of the SIMD&FP register holding the accumulate vector, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP register holding the accumulate vector, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm[x]> Is the 64-bit name of the second SIMD&FP source register holding the scalar. If <dt> is S16, Dm is restricted to D0-D7. Dm is encoded in "Vm<2:0>", and x is encoded in "M:Vm<3>". If <dt> is S32, Dm is restricted to D0-D15. Dm is encoded in "Vm", and x is encoded in "M".

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
EncodingSpecificOperations();  CheckAdvSIMDEnabled();
round_const = 1 << (esize-1);
if scalar_form then op2 = SInt(Elem[D[m],index,esize]);
for r = 0 to regs-1
    for e = 0 to elements-1
        op1 = SInt(Elem[D[n+r],e,esize]);
        op3 = SInt(Elem[D[d+r],e,esize]) << esize;
        if !scalar_form then op2 = SInt(Elem[D[m+r],e,esize]);
        (result, sat) = SignedSatQ((op3 + 2*(op1*op2) + round_const) >> esize, esize);
        Elem[D[d+r],e,esize] = result;
        if sat then FPSCR.QC = '1';
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VQRDMLSH

Vector Saturating Rounding Doubling Multiply Subtract Returning High Half. This instruction multiplies the vector elements of the first source SIMD&FP register with either the corresponding vector elements of the second source SIMD&FP register or the value of a vector element of the second source SIMD&FP register, without saturating the multiply results, doubles the results, and subtracts the most significant half of the final results from the vector elements of the destination SIMD&FP register. The results are rounded.

If any of the results overflow, they are saturated. The cumulative saturation bit, *FPSCR*.QC, is set if saturation occurs. For details see *Pseudocode details of saturation*.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2) .

A1

(FEAT_RDM)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	0	D	size		Vn				Vd				1	1	0	0	N	Q	M	1	Vm			

64-bit SIMD vector (Q == 0)

```
VQRDMLSH{<q>}.<dt> <Dd>, <Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VQRDMLSH{<q>}.<dt> <Qd>, <Qn>, <Qm>
```

```
if !HaveQRDMLAHExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '00' || size == '11' then UNDEFINED;
add = FALSE;  scalar_form = FALSE;  esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

A2

(FEAT_RDM)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	Q	1	D	!= 11		Vn				Vd				1	1	1	1	N	1	M	0	Vm			
										size																					

64-bit SIMD vector (Q == 0)

```
VQRDMLSH{<q>}.<dt> <Dd>, <Dn>, <Dm[x]>
```

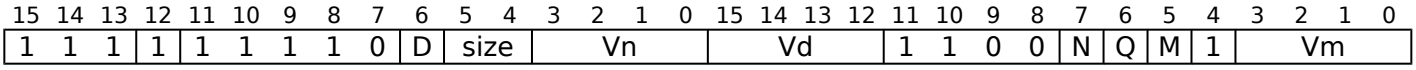
128-bit SIMD vector (Q == 1)

```
VQRDMLSH{<q>}.<dt> <Qd>, <Qn>, <Dm[x]>
```

```
if !HaveQRDMLAHExt() then UNDEFINED;
if size == '11' then SEE "Related encodings";
if size == '00' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
add = FALSE;  scalar_form = TRUE;  d = UInt(D:Vd);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```

T1

(FEAT_RDM)



64-bit SIMD vector (Q == 0)

```
VQRDMLSH{<q>}.<dt> <Dd>, <Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VQRDMLSH{<q>}.<dt> <Qd>, <Qn>, <Qm>
```

```
if !HaveQRDMLAHExt() then UNDEFINED;
if InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '00' || size == '11' then UNDEFINED;
add = FALSE;  scalar_form = FALSE;  esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

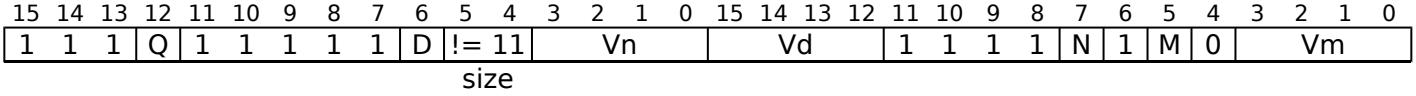
CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T2

(FEAT_RDM)



64-bit SIMD vector (Q == 0)

```
VQRDMLSH{<q>}.<dt> <Dd>, <Dn>, <Dm[x]>
```

128-bit SIMD vector (Q == 1)

```
VQRDMLSH{<q>}.<dt> <Qd>, <Qn>, <Dm[x]>
```

```
if !HaveQRDMLAHExt() then UNDEFINED;
if InITBlock() then UNPREDICTABLE;
if size == '11' then SEE "Related encodings";
if size == '00' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
add = FALSE;  scalar_form = TRUE;  d = UInt(D:Vd);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Related encodings: See *Advanced SIMD data-processing* for the T32 instruction set, or *Advanced SIMD data-processing* for the A32 instruction set.

Assembler Symbols

<q> See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the operands, encoded in “size”:

size	<dt>
01	S16
10	S32

<Qd> Is the 128-bit name of the SIMD&FP register holding the accumulate vector, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP register holding the accumulate vector, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm[x]> Is the 64-bit name of the second SIMD&FP source register holding the scalar. If <dt> is S16, Dm is restricted to D0-D7. Dm is encoded in "Vm<2:0>", and x is encoded in "M:Vm<3>". If <dt> is S32, Dm is restricted to D0-D15. Dm is encoded in "Vm", and x is encoded in "M".

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
EncodingSpecificOperations();  CheckAdvSIMDEnabled();
round_const = 1 << (esize-1);
if scalar_form then op2 = SInt(Elem[D[m],index,esize]);
for r = 0 to regs-1
    for e = 0 to elements-1
        op1 = SInt(Elem[D[n+r],e,esize]);
        op3 = SInt(Elem[D[d+r],e,esize]) << esize;
        if !scalar_form then op2 = SInt(Elem[D[m+r],e,esize]);
        (result, sat) = SignedSatQ((op3 - 2*(op1*op2) + round_const) >> esize, esize);
        Elem[D[d+r],e,esize] = result;
        if sat then FPSCR.QC = '1';
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VQRDMULH

Vector Saturating Rounding Doubling Multiply Returning High Half multiplies corresponding elements in two vectors, doubles the results, and places the most significant half of the final results in the destination vector. The results are rounded. For truncated results see *VQDMULH*.

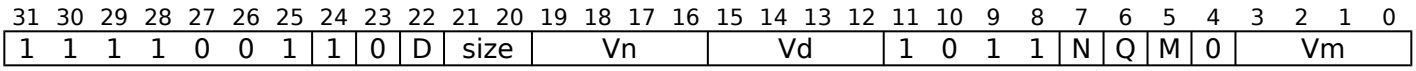
The second operand can be a scalar instead of a vector. For more information about scalars see *Advanced SIMD scalars*.

If any of the results overflow, they are saturated. The cumulative saturation bit, *FPSCR*.QC, is set if saturation occurs. For details see *Pseudocode details of saturation*.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2) .

A1



64-bit SIMD vector (Q == 0)

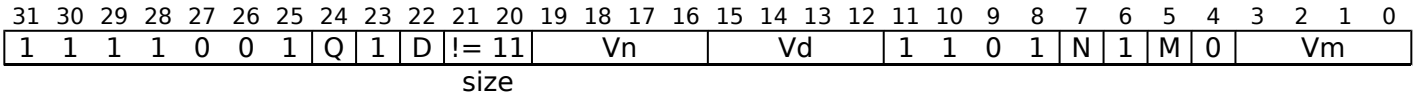
```
VQRDMULH{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VQRDMULH{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '00' || size == '11' then UNDEFINED;
scalar_form = FALSE;  esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

A2



64-bit SIMD vector (Q == 0)

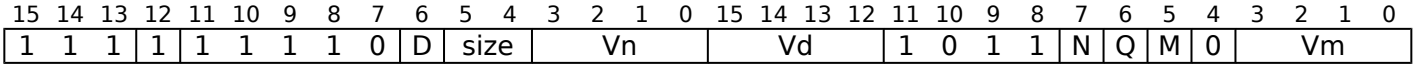
```
VQRDMULH{<c>}{<q>}.<dt> {<Dd>,} <Dn>, <Dm[x]>
```

128-bit SIMD vector (Q == 1)

```
VQRDMULH{<c>}{<q>}.<dt> {<Qd>,} <Qn>, <Dm[x]>
```

```
if size == '11' then SEE "Related encodings";
if size == '00' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
scalar_form = TRUE;  d = UInt(D:Vd);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```

T1



64-bit SIMD vector (Q == 0)

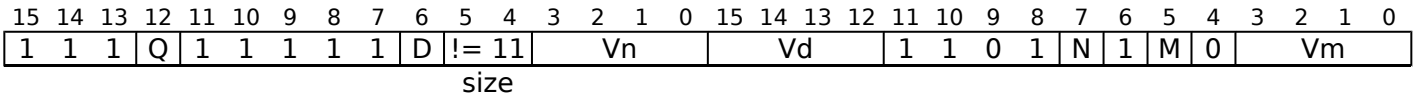
```
VQRDMULH{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VQRDMULH{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '00' || size == '11' then UNDEFINED;
scalar_form = FALSE;  esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T2



64-bit SIMD vector (Q == 0)

```
VQRDMULH{<c>}{<q>}.<dt> {<Dd>,} <Dn>, <Dm[x]>
```

128-bit SIMD vector (Q == 1)

```
VQRDMULH{<c>}{<q>}.<dt> {<Qd>,} <Qn>, <Dm[x]>
```

```
if size == '11' then SEE "Related encodings";
if size == '00' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
scalar_form = TRUE;  d = UInt(D:Vd);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;
if size == '01' then esize = 16;  elements = 4;  m = UInt(Vm<2:0>);  index = UInt(M:Vm<3>);
if size == '10' then esize = 32;  elements = 2;  m = UInt(Vm);  index = UInt(M);
```

Related encodings: See *Advanced SIMD data-processing* for the T32 instruction set, or *Advanced SIMD data-processing* for the A32 instruction set.

Assembler Symbols

<c> For encoding A1 and A2: see *Standard assembler syntax fields*. This encoding must be unconditional.

For encoding T1 and T2: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the operands, encoded in “size”:

size	<dt>
01	S16
10	S32

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm[x]> Is the 64-bit name of the second SIMD&FP source register holding the scalar. If <dt> is S16, Dm is restricted to D0-D7. Dm is encoded in "Vm<2:0>", and x is encoded in "M:Vm<3>". If <dt> is S32, Dm is restricted to D0-D15. Dm is encoded in "Vm", and x is encoded in "M".

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    round_const = 1 << (esize-1);
    if scalar_form then op2 = SInt(Elem[D[m],index,esize]);
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = SInt(Elem[D[n+r],e,esize]);
            if !scalar_form then op2 = SInt(Elem[D[m+r],e,esize]);
            (result, sat) = SignedSatQ((2*op1*op2 + round_const) >> esize, esize);
            Elem[D[d+r],e,esize] = result;
            if sat then FPSCR.QC = '1';
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VQRSHL

Vector Saturating Rounding Shift Left takes each element in a vector, shifts them by a value from the least significant byte of the corresponding element of a second vector, and places the results in the destination vector. If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift.

For truncated results see *VQSHL (register)*.

The first operand and result elements are the same data type, and can be any one of:

- 8-bit, 16-bit, 32-bit, or 64-bit signed integers.
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

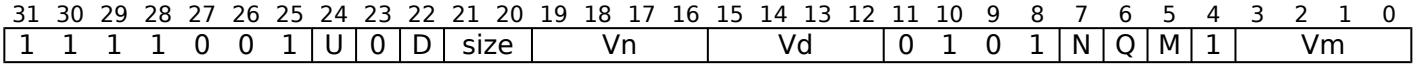
The second operand is a signed integer of the same size.

If any of the results overflow, they are saturated. The cumulative saturation bit, *FPSCR*.QC, is set if saturation occurs. For details see *Pseudocode details of saturation*.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



64-bit SIMD vector (Q == 0)

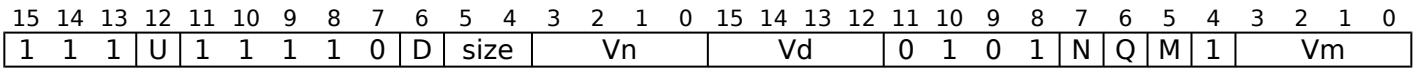
```
VQRSHL{<c>}{<q>}.<dt> {<Dd>,} <Dm>, <Dn>
```

128-bit SIMD vector (Q == 1)

```
VQRSHL{<c>}{<q>}.<dt> {<Qd>,} <Qm>, <Qn>
```

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1' || Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;
```

T1



64-bit SIMD vector (Q == 0)

```
VQRSHL{<c>}{<q>}.<dt> {<Dd>,} <Dm>, <Dn>
```

128-bit SIMD vector (Q == 1)

```
VQRSHL{<c>}{<q>}.<dt> {<Qd>,} <Qm>, <Qn>
```

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1' || Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>	For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional. For encoding T1: see *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the vectors, encoded in “U:size”:

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
0	11	S64
1	00	U8
1	01	U16
1	10	U32
1	11	U64

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            shift = SInt(Elem[D[n+r],e,esize]<7:0>);
            round_const = 1 << (-1-shift); // 0 for left shift, 2^(n-1) for right shift
            operand = Int(Elem[D[m+r],e,esize], unsigned);
            (result, sat) = SatQ((operand + round_const) << shift, esize, unsigned);
            Elem[D[d+r],e,esize] = result;
            if sat then FPSCR.QC = '1';
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VQRSHRN, VQRSHRUN

Vector Saturating Rounding Shift Right, Narrow takes each element in a quadword vector of integers, right shifts them by an immediate value, and places the rounded results in a doubleword vector.

For truncated results, see *VQSHRN and VQSHRUN*.

The operand elements must all be the same size, and can be any one of:

- 16-bit, 32-bit, or 64-bit signed integers.
- 16-bit, 32-bit, or 64-bit unsigned integers.

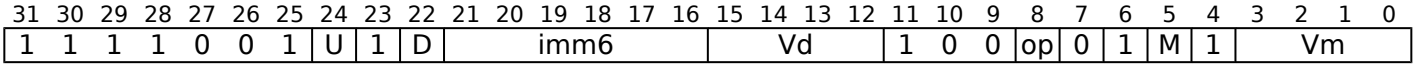
The result elements are half the width of the operand elements. If the operand elements are signed, the results can be either signed or unsigned. If the operand elements are unsigned, the result elements must also be unsigned.

If any of the results overflow, they are saturated. The cumulative saturation bit, *FPSCR*.QC, is set if saturation occurs. For details see *Pseudocode details of saturation*.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



Signed result (!(imm6 == 000xxx) && op == 1)

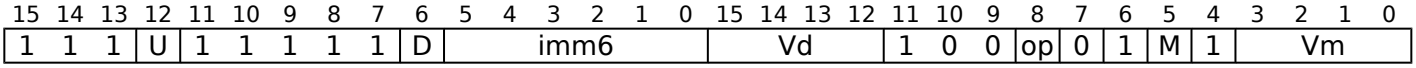
```
VQRSHRN{<c>}{<q>}.<type><size> <Dd>, <Qm>, #<imm>
```

Unsigned result (U == 1 && !(imm6 == 000xxx) && op == 0)

```
VQRSHRUN{<c>}{<q>}.<type><size> <Dd>, <Qm>, #<imm>
```

```
if imm6 == '000xxx' then SEE "Related encodings";
if U == '0' && op == '0' then SEE "VRSHRN";
if Vm<0> == '1' then UNDEFINED;
case imm6 of
    when '001xxx'  esize = 8;  elements = 8;  shift_amount = 16 - UInt(imm6);
    when '01xxxx'  esize = 16;  elements = 4;  shift_amount = 32 - UInt(imm6);
    when '1xxxxx'  esize = 32;  elements = 2;  shift_amount = 64 - UInt(imm6);
src_unsigned = (U == '1' && op == '1');  dest_unsigned = (U == '1');
d = UInt(D:Vd);  m = UInt(M:Vm);
```

T1



Signed result (!(imm6 == 000xxx) && op == 1)

```
VQRSHRN{<c>}{<q>}.<type><size> <Dd>, <Qm>, #<imm>
```

Unsigned result (U == 1 && !(imm6 == 000xxx) && op == 0)

```
VQRSHRUN{<c>}{<q>}.<type><size> <Dd>, <Qm>, #<imm>
```

```
if imm6 == '000xxx' then SEE "Related encodings";
if U == '0' && op == '0' then SEE "VRSHRN";
if Vm<0> == '1' then UNDEFINED;
case imm6 of
    when '001xxx'  esize = 8;  elements = 8;  shift_amount = 16 - UInt(imm6);
    when '01xxxx'  esize = 16;  elements = 4;  shift_amount = 32 - UInt(imm6);
    when '1xxxxx'  esize = 32;  elements = 2;  shift_amount = 64 - UInt(imm6);
src_unsigned = (U == '1' && op == '1');  dest_unsigned = (U == '1');
d = UInt(D:Vd);  m = UInt(M:Vm);
```

Related encodings: See *Advanced SIMD one register and modified immediate* for the T32 instruction set, or *Advanced SIMD one register and modified immediate* for the A32 instruction set.

Assembler Symbols

<c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<type> For the signed result variant: is the data type for the elements of the vectors, encoded in “U”:

U	<type>
0	S
1	U

For the unsigned result variant: is the data type for the elements of the vectors, encoded in “U”:

U	<type>
1	S

<size> Is the data size for the elements of the vectors, encoded in “imm6<5:3>”:

imm6<5:3>	<size>
001	16
01x	32
1xx	64

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<imm> Is an immediate value, in the range 1 to <size>/2, encoded in the "imm6" field as <size>/2 - <imm>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    round_const = 1 << (shift_amount - 1);
    for e = 0 to elements-1
        operand = Int(Elem[Qin[m>>1],e,2*esize], src_unsigned);
        (result, sat) = SatQ((operand + round_const) >> shift_amount, esize, dest_unsigned);
        Elem[D[d],e,esize] = result;
        if sat then FPSCR.QC = '1';
```

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VQRSHRN (zero)

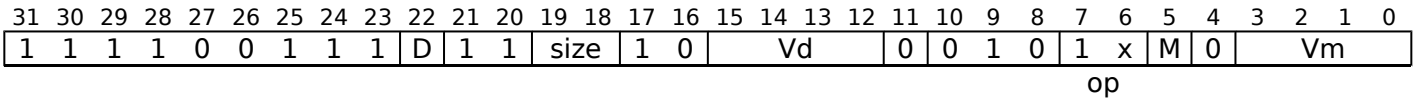
Vector Saturating Rounding Shift Right, Narrow takes each element in a quadword vector of integers, right shifts them by an immediate value, and places the signed rounded results in a doubleword vector

This is a pseudo-instruction of VQMOVN, VQMOVUN. This means:

- The encodings in this description are named to match the encodings of VQMOVN, VQMOVUN.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of VQMOVN, VQMOVUN gives the operational pseudocode for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



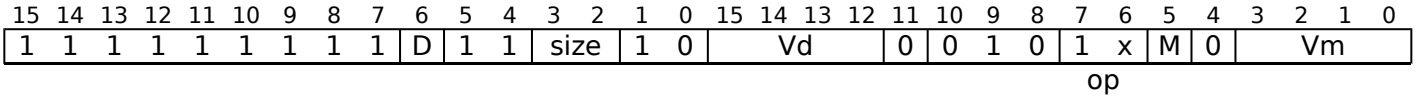
Signed result

```
VQRSHRN{<c>}{<q>}.<dt> <Dd>, <Qm>, #0
```

is equivalent to

```
VQMOVN{<c>}{<q>}.<dt> <Dd>, <Qm>
```

T1



Signed result

```
VQRSHRN{<c>}{<q>}.<dt> <Dd>, <Qm>, #0
```

is equivalent to

```
VQMOVN{<c>}{<q>}.<dt> <Dd>, <Qm>
```

Assembler Symbols

<c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.
For encoding T1: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the operand, encoded in “op<0>:size”:

op<0>	size	<dt>
0	00	S16
0	01	S32
0	10	S64
0	11	RESERVED
1	00	U16
1	01	U32
1	10	U64
1	11	RESERVED

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

The description of VQMOVN, VQMOVUN gives the operational pseudocode for this instruction.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VQRSHRUN (zero)

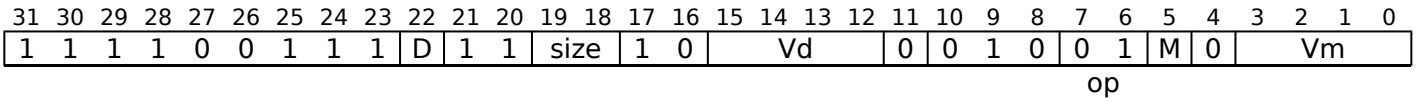
Vector Saturating Rounding Shift Right, Narrow takes each element in a quadword vector of integers, right shifts them by an immediate value, and places the unsigned rounded results in a doubleword vector

This is a pseudo-instruction of VQMOVN, VQMOVUN. This means:

- The encodings in this description are named to match the encodings of VQMOVN, VQMOVUN.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of VQMOVN, VQMOVUN gives the operational pseudocode for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



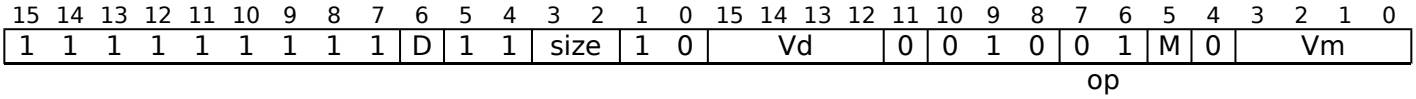
Unsigned result

```
VQRSHRUN{<c>}{<q>}.<dt> <Dd>, <Qm>, #0
```

is equivalent to

```
VQMOVUN{<c>}{<q>}.<dt> <Dd>, <Qm>
```

T1



Unsigned result

```
VQRSHRUN{<c>}{<q>}.<dt> <Dd>, <Qm>, #0
```

is equivalent to

```
VQMOVUN{<c>}{<q>}.<dt> <Dd>, <Qm>
```

Assembler Symbols

<c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.
For encoding T1: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the operand, encoded in “size”:

size	<dt>
00	S16
01	S32
10	S64
11	RESERVED

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

The description of VQMOVN, VQMOVUN gives the operational pseudocode for this instruction.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VQSHL, VQSHLU (immediate)

Vector Saturating Shift Left (immediate) takes each element in a vector of integers, left shifts them by an immediate value, and places the results in a second vector.

The operand elements must all be the same size, and can be any one of:

- 8-bit, 16-bit, 32-bit, or 64-bit signed integers.
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

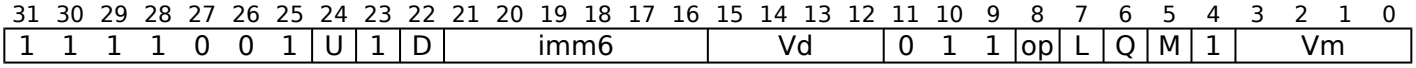
The result elements are the same size as the operand elements. If the operand elements are signed, the results can be either signed or unsigned. If the operand elements are unsigned, the result elements must also be unsigned.

If any of the results overflow, they are saturated. The cumulative saturation bit, *FPSCR*.QC, is set if saturation occurs. For details see *Pseudocode details of saturation*.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



VQSHL,double,signed-result (!(imm6 == 000xxx && L == 0) && op == 1 && Q == 0)

```
VQSHL{<c>}{<q>}.<type><size> {<Dd>,} <Dm>, #<imm>
```

VQSHL,quad,signed-result (!(imm6 == 000xxx && L == 0) && op == 1 && Q == 1)

```
VQSHL{<c>}{<q>}.<type><size> {<Qd>,} <Qm>, #<imm>
```

VQSHLU,double,unsigned-result (U == 1 && !(imm6 == 000xxx && L == 0) && op == 0 && Q == 0)

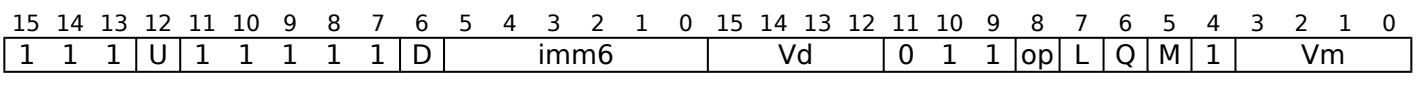
```
VQSHLU{<c>}{<q>}.<type><size> {<Dd>,} <Dm>, #<imm>
```

VQSHLU,quad,unsigned-result (U == 1 && !(imm6 == 000xxx && L == 0) && op == 0 && Q == 1)

```
VQSHLU{<c>}{<q>}.<type><size> {<Qd>,} <Qm>, #<imm>
```

```
if (L:imm6) == '0000xxx' then SEE "Related encodings";
if U == '0' && op == '0' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
case L:imm6 of
    when '0001xxx'  esize = 8;  elements = 8;  shift_amount = UInt(imm6) - 8;
    when '001xxxx'  esize = 16;  elements = 4;  shift_amount = UInt(imm6) - 16;
    when '01xxxxx'  esize = 32;  elements = 2;  shift_amount = UInt(imm6) - 32;
    when '1xxxxxx'  esize = 64;  elements = 1;  shift_amount = UInt(imm6);
src_unsigned = (U == '1' && op == '1');  dest_unsigned = (U == '1');
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1



VQSHL,double,signed-result (!(imm6 == 000xxx && L == 0) && op == 1 && Q == 0)

```
VQSHL{<c>}{<q>}.<type><size> {<Dd>,} <Dm>, #<imm>
```

VQSHL,quad,signed-result (!(imm6 == 000xxx && L == 0) && op == 1 && Q == 1)

```
VQSHL{<c>}{<q>}.<type><size> {<Qd>,} <Qm>, #<imm>
```

VQSHLU,double,unsigned-result (U == 1 && !(imm6 == 000xxx && L == 0) && op == 0 && Q == 0)

```
VQSHLU{<c>}{<q>}.<type><size> {<Dd>,} <Dm>, #<imm>
```

VQSHLU,quad,unsigned-result (U == 1 && !(imm6 == 000xxx && L == 0) && op == 0 && Q == 1)

```
VQSHLU{<c>}{<q>}.<type><size> {<Qd>,} <Qm>, #<imm>
```

```
if (L:imm6) == '0000xxx' then SEE "Related encodings";
if U == '0' && op == '0' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
case L:imm6 of
    when '0001xxx'  esize = 8;  elements = 8;  shift_amount = UInt(imm6) - 8;
    when '001xxxx'  esize = 16;  elements = 4;  shift_amount = UInt(imm6) - 16;
    when '01xxxxx'  esize = 32;  elements = 2;  shift_amount = UInt(imm6) - 32;
    when '1xxxxxx'  esize = 64;  elements = 1;  shift_amount = UInt(imm6);
src_unsigned = (U == '1' && op == '1');  dest_unsigned = (U == '1');
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Related encodings: See *Advanced SIMD one register and modified immediate* for the T32 instruction set, or *Advanced SIMD one register and modified immediate* for the A32 instruction set.

Assembler Symbols

<c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.
For encoding T1: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<type> Is the data type for the elements of the vectors, encoded in “U”:

U	<type>
0	S
1	U

<size> Is the data size for the elements of the vectors, encoded in “L:imm6<5:3>”:

L	imm6<5:3>	<size>
0	001	8
0	01x	16
0	1xx	32
1	xxx	64

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

<imm> Is an immediate value, in the range 0 to <size>-1, encoded in the "imm6" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            operand = Int(Elem[D[m+r],e,esize], src_unsigned);
            (result, sat) = SatQ(operand << shift_amount, esize, dest_unsigned);
            Elem[D[d+r],e,esize] = result;
            if sat then FPSCR.QC = '1';
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VQSHL (register)

Vector Saturating Shift Left (register) takes each element in a vector, shifts them by a value from the least significant byte of the corresponding element of a second vector, and places the results in the destination vector. If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift.

The results are truncated. For rounded results, see *VQRSHL*.

The first operand and result elements are the same data type, and can be any one of:

- 8-bit, 16-bit, 32-bit, or 64-bit signed integers.
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

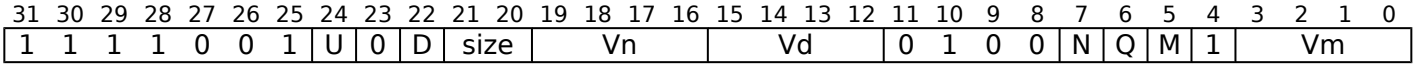
The second operand is a signed integer of the same size.

If any of the results overflow, they are saturated. The cumulative saturation bit, *FPSCR*.QC, is set if saturation occurs. For details see *Pseudocode details of saturation*.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



64-bit SIMD vector (Q == 0)

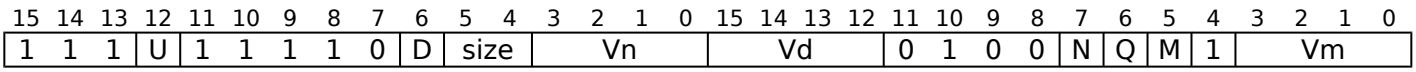
```
VQSHL{<c>}{<q>}.<dt> {<Dd>,} <Dm>, <Dn>
```

128-bit SIMD vector (Q == 1)

```
VQSHL{<c>}{<q>}.<dt> {<Qd>,} <Qm>, <Qn>
```

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1' || Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;
```

T1



64-bit SIMD vector (Q == 0)

```
VQSHL{<c>}{<q>}.<dt> {<Dd>,} <Dm>, <Dn>
```

128-bit SIMD vector (Q == 1)

```
VQSHL{<c>}{<q>}.<dt> {<Qd>,} <Qm>, <Qn>
```

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1' || Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>	For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional. For encoding T1: see *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the vectors, encoded in “U:size”:

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
0	11	S64
1	00	U8
1	01	U16
1	10	U32
1	11	U64

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            shift = SInt(Elem[D[n+r],e,esize]<7:0>);
            operand = Int(Elem[D[m+r],e,esize], unsigned);
            (result,sat) = SatQ(operand << shift, esize, unsigned);
            Elem[D[d+r],e,esize] = result;
            if sat then FPSCR.QC = '1';
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VQSHRN, VQSHRUN

Vector Saturating Shift Right, Narrow takes each element in a quadword vector of integers, right shifts them by an immediate value, and places the truncated results in a doubleword vector.

For rounded results, see *VQRSHRN and VQRSHRUN*.

The operand elements must all be the same size, and can be any one of:

- 16-bit, 32-bit, or 64-bit signed integers.
- 16-bit, 32-bit, or 64-bit unsigned integers.

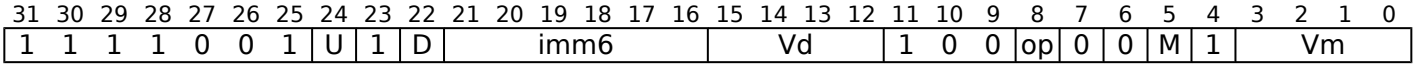
The result elements are half the width of the operand elements. If the operand elements are signed, the results can be either signed or unsigned. If the operand elements are unsigned, the result elements must also be unsigned.

If any of the results overflow, they are saturated. The cumulative saturation bit, *FPSCR*.QC, is set if saturation occurs. For details see *Pseudocode details of saturation*.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



Signed result (!(imm6 == 000xxx) && op == 1)

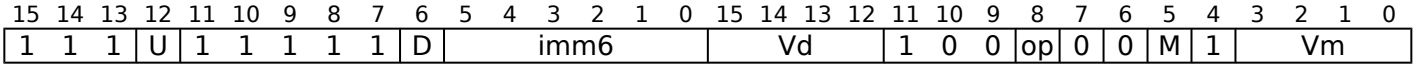
```
VQSHRN{<c>}{<q>}.<type><size> <Dd>, <Qm>, #<imm>
```

Unsigned result (U == 1 && !(imm6 == 000xxx) && op == 0)

```
VQSHRUN{<c>}{<q>}.<type><size> <Dd>, <Qm>, #<imm>
```

```
if imm6 == '000xxx' then SEE "Related encodings";
if U == '0' && op == '0' then SEE "VSHRN";
if Vm<0> == '1' then UNDEFINED;
case imm6 of
    when '001xxx'  esize = 8;  elements = 8;  shift_amount = 16 - UInt(imm6);
    when '01xxxx'  esize = 16;  elements = 4;  shift_amount = 32 - UInt(imm6);
    when '1xxxxx'  esize = 32;  elements = 2;  shift_amount = 64 - UInt(imm6);
src_unsigned = (U == '1' && op == '1');  dest_unsigned = (U == '1');
d = UInt(D:Vd);  m = UInt(M:Vm);
```

T1



Signed result (!(imm6 == 000xxx) && op == 1)

```
VQSHRN{<c>}{<q>}.<type><size> <Dd>, <Qm>, #<imm>
```

Unsigned result (U == 1 && !(imm6 == 000xxx) && op == 0)

```
VQSHRUN{<c>}{<q>}.<type><size> <Dd>, <Qm>, #<imm>
```

```
if imm6 == '000xxx' then SEE "Related encodings";
if U == '0' && op == '0' then SEE "VSHRN";
if Vm<0> == '1' then UNDEFINED;
case imm6 of
    when '001xxx'  esize = 8;  elements = 8;  shift_amount = 16 - UInt(imm6);
    when '01xxxx'  esize = 16;  elements = 4;  shift_amount = 32 - UInt(imm6);
    when '1xxxxx'  esize = 32;  elements = 2;  shift_amount = 64 - UInt(imm6);
src_unsigned = (U == '1' && op == '1');  dest_unsigned = (U == '1');
d = UInt(D:Vd);  m = UInt(M:Vm);
```

Related encodings: See *Advanced SIMD one register and modified immediate* for the T32 instruction set, or *Advanced SIMD one register and modified immediate* for the A32 instruction set.

Assembler Symbols

<c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<type> For the signed result variant: is the data type for the elements of the vectors, encoded in “U”:

U	<type>
0	S
1	U

For the unsigned result variant: is the data type for the elements of the vectors, encoded in “U”:

U	<type>
1	S

<size> Is the data size for the elements of the vectors, encoded in “imm6<5:3>”:

imm6<5:3>	<size>
001	16
01x	32
1xx	64

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<imm> Is an immediate value, in the range 1 to <size>/2, encoded in the "imm6" field as <size>/2 - <imm>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for e = 0 to elements-1
        operand = Int(Elem[Qin[m>>1],e,2*esize], src_unsigned);
        (result, sat) = SatQ(operand >> shift_amount, esize, dest_unsigned);
        Elem[D[d],e,esize] = result;
        if sat then FPSCR.QC = '1';
```


Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VQSHRN (zero)

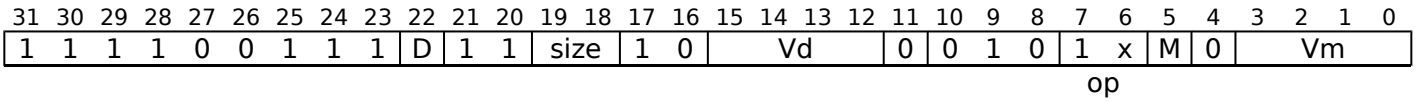
Vector Saturating Shift Right, Narrow takes each element in a quadword vector of integers, right shifts them by an immediate value, and places the signed truncated results in a doubleword vector

This is a pseudo-instruction of VQMOVN, VQMOVUN. This means:

- The encodings in this description are named to match the encodings of VQMOVN, VQMOVUN.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of VQMOVN, VQMOVUN gives the operational pseudocode for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



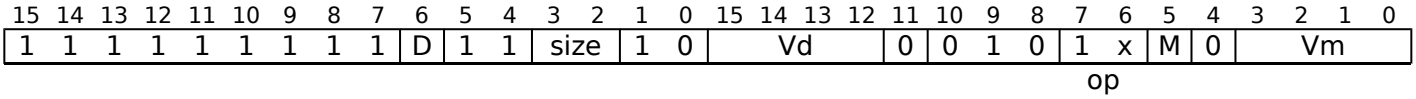
Signed result

```
VQSHRN{<c>}{<q>}.<dt> <Dd>, <Qm>, #0
```

is equivalent to

```
VQMOVN{<c>}{<q>}.<dt> <Dd>, <Qm>
```

T1



Signed result

```
VQSHRN{<c>}{<q>}.<dt> <Dd>, <Qm>, #0
```

is equivalent to

```
VQMOVN{<c>}{<q>}.<dt> <Dd>, <Qm>
```

Assembler Symbols

<c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.
For encoding T1: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the operand, encoded in “op<0>:size”:

op<0>	size	<dt>
0	00	S16
0	01	S32
0	10	S64
0	11	RESERVED
1	00	U16
1	01	U32
1	10	U64
1	11	RESERVED

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

The description of VQMOVN, VQMOVUN gives the operational pseudocode for this instruction.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VQSHRUN (zero)

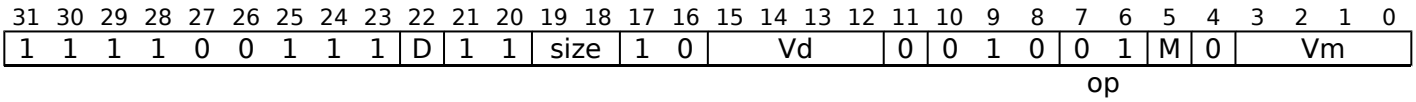
Vector Saturating Shift Right, Narrow takes each element in a quadword vector of integers, right shifts them by an immediate value, and places the unsigned truncated results in a doubleword vector

This is a pseudo-instruction of VQMOVN, VQMOVUN. This means:

- The encodings in this description are named to match the encodings of VQMOVN, VQMOVUN.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of VQMOVN, VQMOVUN gives the operational pseudocode for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



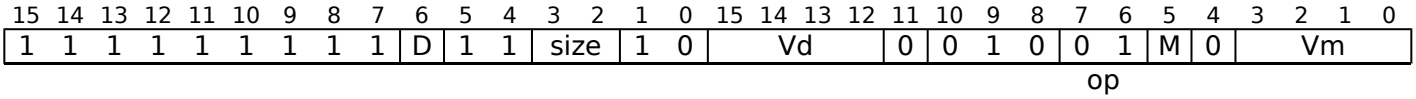
Unsigned result

```
VQSHRUN{<c>}{<q>}.<dt> <Dd>, <Qm>, #0
```

is equivalent to

```
VQMOVUN{<c>}{<q>}.<dt> <Dd>, <Qm>
```

T1



Unsigned result

```
VQSHRUN{<c>}{<q>}.<dt> <Dd>, <Qm>, #0
```

is equivalent to

```
VQMOVUN{<c>}{<q>}.<dt> <Dd>, <Qm>
```

Assembler Symbols

<c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.
For encoding T1: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the operand, encoded in “size”:

size	<dt>
00	S16
01	S32
10	S64
11	RESERVED

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

The description of VQMOVN, VQMOVUN gives the operational pseudocode for this instruction.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VQSUB

Vector Saturating Subtract subtracts the elements of the second operand vector from the corresponding elements of the first operand vector, and places the results in the destination vector. Signed and unsigned operations are distinct.

The operand and result elements must all be the same type, and can be any one of:

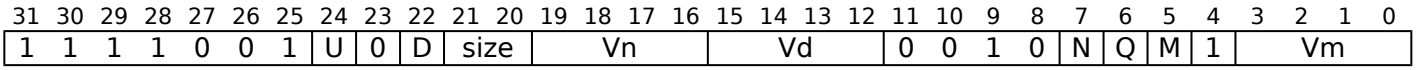
- 8-bit, 16-bit, 32-bit, or 64-bit signed integers.
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

If any of the results overflow, they are saturated. The cumulative saturation bit, *FPSCR*.QC, is set if saturation occurs. For details see *Pseudocode details of saturation*.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



64-bit SIMD vector (Q == 0)

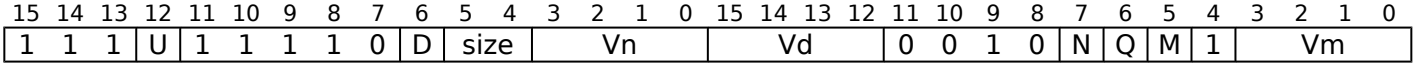
```
VQSUB{<c>}{<q>}.<dt> {<Dd>,} <Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VQSUB{<c>}{<q>}.<dt> {<Qd>,} <Qn>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1



64-bit SIMD vector (Q == 0)

```
VQSUB{<c>}{<q>}.<dt> {<Dd>,} <Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VQSUB{<c>}{<q>}.<dt> {<Qd>,} <Qn>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>	For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional. For encoding T1: see *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<dt>	Is the data type for the elements of the vectors, encoded in “U:size”:

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
0	11	S64
1	00	U8
1	01	U16
1	10	U32
1	11	U64

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            diff = Int(Elem[D[n+r],e,esize], unsigned) - Int(Elem[D[m+r],e,esize], unsigned);
            (Elem[D[d+r],e,esize], sat) = SatQ(diff, esize, unsigned);
            if sat then FPSCR.QC = '1';
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VRADDHN

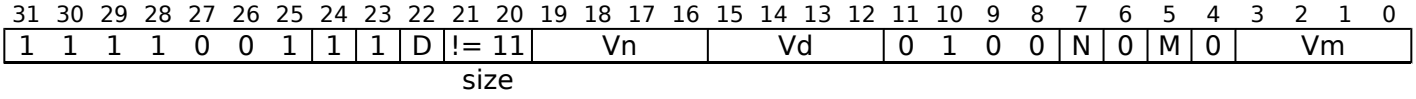
Vector Rounding Add and Narrow, returning High Half adds corresponding elements in two quadword vectors, and places the most significant half of each result in a doubleword vector. The results are rounded. For truncated results, see *VADDHN*.

The operand elements can be 16-bit, 32-bit, or 64-bit integers. There is no distinction between signed and unsigned integers.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

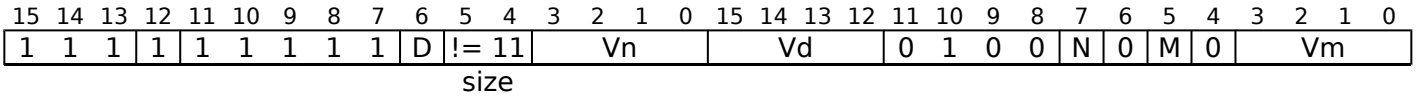


A1

```
VRADDHN{<c>}{<q>}.<dt> <Dd>, <Qn>, <Qm>
```

```
if size == '11' then SEE "Related encodings";
if Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

T1



T1

```
VRADDHN{<c>}{<q>}.<dt> <Dd>, <Qn>, <Qm>
```

```
if size == '11' then SEE "Related encodings";
if Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

Related encodings: See *Advanced SIMD data-processing* for the T32 instruction set, or *Advanced SIMD data-processing* for the A32 instruction set.

Assembler Symbols

<c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the operands, encoded in “size”:

size	<dt>
00	I16
01	I32
10	I64

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    round_const = 1 << (esize-1);
    for e = 0 to elements-1
        result = Elem[Qin[n>>1],e,2*esize] + Elem[Qin[m>>1],e,2*esize] + round_const;
        Elem[D[d],e,esize] = result<2*esize-1:esize>;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VRECPE

Vector Reciprocal Estimate finds an approximate reciprocal of each element in the operand vector, and places the results in the destination vector.

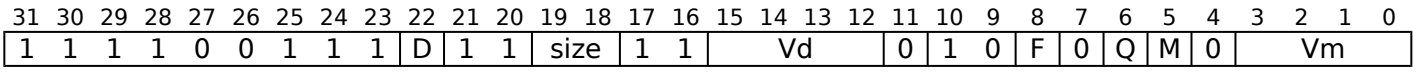
The operand and result elements are the same type, and can be floating-point numbers or unsigned integers.

For details of the operation performed by this instruction see *Floating-point reciprocal square root estimate and step*.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



64-bit SIMD vector (Q == 0)

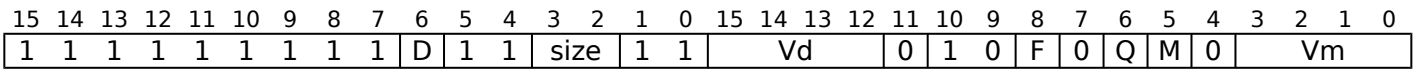
```
VRECPE{<c>}{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VRECPE{<c>}{<q>}.<dt> <Qd>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && (!HaveFP16Ext() || F == '0')) || size IN {'00', '11'} then UNDEFINED;
floating_point = (F == '1');
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



64-bit SIMD vector (Q == 0)

```
VRECPE{<c>}{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VRECPE{<c>}{<q>}.<dt> <Qd>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && (!HaveFP16Ext() || F == '0')) || size IN {'00', '11'} then UNDEFINED;
if size == '01' && InITBlock()  then UNPREDICTABLE;
floating_point = (F == '1');
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_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VRECPS

Vector Reciprocal Step multiplies the elements of one vector by the corresponding elements of another vector, subtracts each of the products from 2.0, and places the results into the elements of the destination vector.

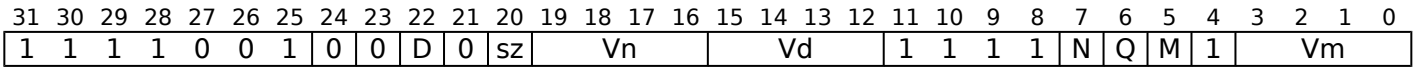
The operand and result elements are floating-point numbers.

For details of the operation performed by this instruction see *Floating-point reciprocal estimate and step*.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



64-bit SIMD vector (Q == 0)

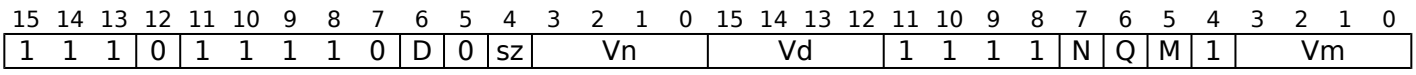
```
VRECPS{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VRECPS{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1



64-bit SIMD vector (Q == 0)

```
VRECPS{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VRECPS{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.
For encoding T1: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the vectors, encoded in “sz”:

sz	<dt>
0	F32
1	F16

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Newton-Raphson iteration
For details of the operation performed and how it can be used in a Newton-Raphson iteration to calculate the reciprocal of a number, see *Floating-point reciprocal estimate and step*.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            Elem[D[d+r],e,esize] = FPRecipStep(Elem[D[n+r],e,esize], Elem[D[m+r],e,esize]);
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

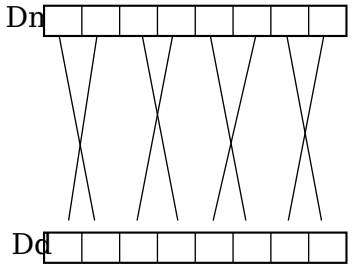
VREV16

Vector Reverse in halfwords reverses the order of 8-bit elements in each halfword of the vector, and places the result in the corresponding destination vector.

There is no distinction between data types, other than size.

The following figure shows an example of the operation of VREV16 doubleword operation.

VREV16.8, doubleword



Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size		0	0	Vd				0	0	0	1	0	Q	M	0	Vm			
																							op								

64-bit SIMD vector (Q == 0)

```
VREV16{<c>}{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VREV16{<c>}{<q>}.<dt> <Qd>, <Qm>
```

```
if UInt(op)+UInt(size) >= 3 then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;

esize = 8 << UInt(size);
integer container_size;
case op of
    when '10' container_size = 16;
    when '01' container_size = 32;
    when '00' container_size = 64;
integer containers = 64 DIV container_size;
integer elements_per_container = container_size DIV esize;

d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size		0	0	Vd				0	0	0	1	0	Q	M	0	Vm			
																							op								

64-bit SIMD vector (Q == 0)

```
VREV16{<c>}{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VREV16{<c>}{<q>}.<dt> <Qd>, <Qm>
```

```
if UInt(op)+UInt(size) >= 3 then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;

esize = 8 << UInt(size);
integer container_size;
case op of
    when '10' container_size = 16;
    when '01' container_size = 32;
    when '00' container_size = 64;
integer containers = 64 DIV container_size;
integer elements_per_container = container_size DIV esize;

d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.
For encoding T1: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the operand, encoded in “size”:

size	<dt>
00	8
01	RESERVED
1x	RESERVED

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();

    bits(64) result;
    integer element;
    integer rev_element;
    for r = 0 to regs-1
        element = 0;
        for c = 0 to containers-1
            rev_element = element + elements_per_container - 1;
            for e = 0 to elements_per_container-1
                Elem[result, rev_element, esize] = Elem[D[m+r], element, esize];
                element = element + 1;
                rev_element = rev_element - 1;
        D[d+r] = result;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:

    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

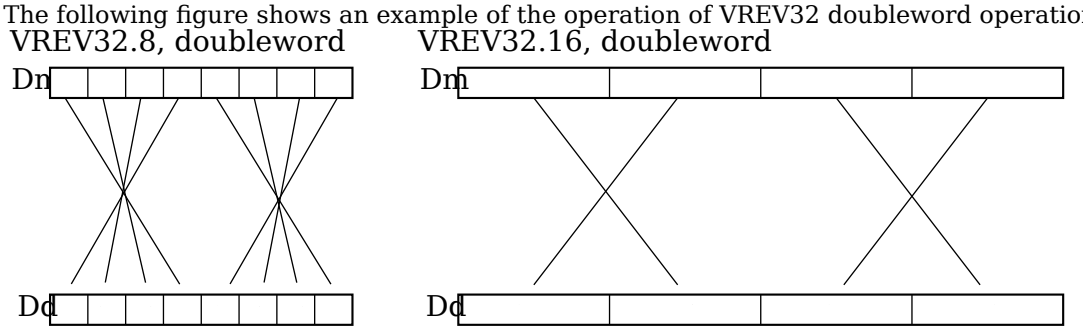
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VREV32

Vector Reverse in words reverses the order of 8-bit or 16-bit elements in each word of the vector, and places the result in the corresponding destination vector.

There is no distinction between data types, other than size.

The following figure shows an example of the operation of VREV32 doubleword operations.



Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size		0	0	Vd				0	0	0	0	1	Q	M	0	Vm			
																							op								

64-bit SIMD vector (Q == 0)

```
VREV32{<c>}{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VREV32{<c>}{<q>}.<dt> <Qd>, <Qm>
```

```
if UInt(op)+UInt(size) >= 3 then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;

esize = 8 << UInt(size);
integer container_size;
case op of
    when '10' container_size = 16;
    when '01' container_size = 32;
    when '00' container_size = 64;
integer containers = 64 DIV container_size;
integer elements_per_container = container_size DIV esize;

d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size		0	0	Vd				0	0	0	0	1	Q	M	0	Vm			
																							op								

64-bit SIMD vector (Q == 0)

```
VREV32{<c>}{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VREV32{<c>}{<q>}.<dt> <Qd>, <Qm>
```

```
if UInt(op)+UInt(size) >= 3 then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;

esize = 8 << UInt(size);
integer container_size;
case op of
    when '10' container_size = 16;
    when '01' container_size = 32;
    when '00' container_size = 64;
integer containers = 64 DIV container_size;
integer elements_per_container = container_size DIV esize;

d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.
For encoding T1: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the operand, encoded in “size”:

size	<dt>
00	8
01	16
1x	RESERVED

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();

    bits(64) result;
    integer element;
    integer rev_element;
    for r = 0 to regs-1
        element = 0;
        for c = 0 to containers-1
            rev_element = element + elements_per_container - 1;
            for e = 0 to elements_per_container-1
                Elem[result, rev_element, esize] = Elem[D[m+r], element, esize];
                element = element + 1;
                rev_element = rev_element - 1;
        D[d+r] = result;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:

    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

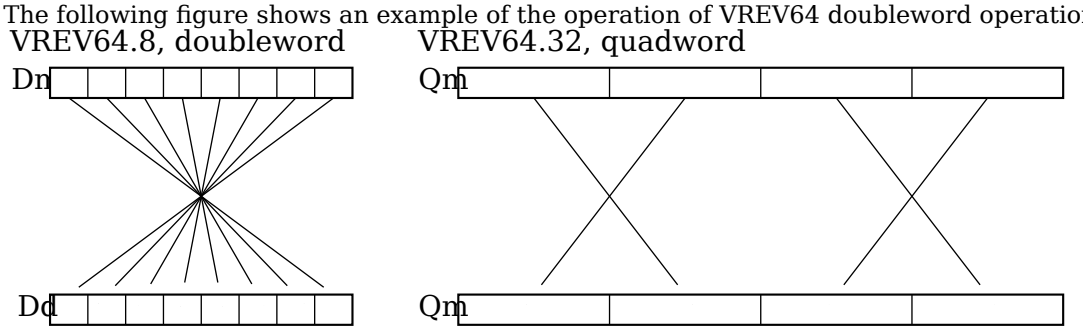
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VREV64

Vector Reverse in doublewords reverses the order of 8-bit, 16-bit, or 32-bit elements in each doubleword of the vector, and places the result in the corresponding destination vector.

There is no distinction between data types, other than size.

The following figure shows an example of the operation of VREV64 doubleword operations.



Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size		0	0	Vd				0	0	0	0	0	Q	M	0	Vm			
																							op								

64-bit SIMD vector (Q == 0)

```
VREV64{<c>}{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VREV64{<c>}{<q>}.<dt> <Qd>, <Qm>
```

```
if UInt(op)+UInt(size) >= 3 then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;

esize = 8 << UInt(size);
integer container_size;
case op of
    when '10' container_size = 16;
    when '01' container_size = 32;
    when '00' container_size = 64;
integer containers = 64 DIV container_size;
integer elements_per_container = container_size DIV esize;

d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size		0	0	Vd				0	0	0	0	0	Q	M	0	Vm			
																							op								

64-bit SIMD vector (Q == 0)

```
VREV64{<c>}{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VREV64{<c>}{<q>}.<dt> <Qd>, <Qm>
```

```
if UInt(op)+UInt(size) >= 3 then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;

esize = 8 << UInt(size);
integer container_size;
case op of
    when '10' container_size = 16;
    when '01' container_size = 32;
    when '00' container_size = 64;
integer containers = 64 DIV container_size;
integer elements_per_container = container_size DIV esize;

d = UInt(D:Vd); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.
For encoding T1: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the operand, encoded in “size”:

size	<dt>
00	8
01	16
10	32
11	RESERVED

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();

    bits(64) result;
    integer element;
    integer rev_element;
    for r = 0 to regs-1
        element = 0;
        for c = 0 to containers-1
            rev_element = element + elements_per_container - 1;
            for e = 0 to elements_per_container-1
                Elem[result, rev_element, esize] = Elem[D[m+r], element, esize];
                element = element + 1;
                rev_element = rev_element - 1;
        D[d+r] = result;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VRHADD

Vector Rounding Halving Add adds corresponding elements in two vectors of integers, shifts each result right one bit, and places the final results in the destination vector.

The operand and result elements are all the same type, and can be any one of:

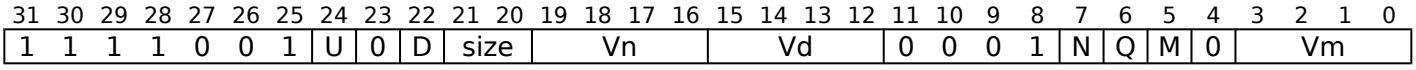
- 8-bit, 16-bit, or 32-bit signed integers.
- 8-bit, 16-bit, or 32-bit unsigned integers.

The results of the halving operations are rounded. For truncated results, see *VHADD*.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



64-bit SIMD vector (Q == 0)

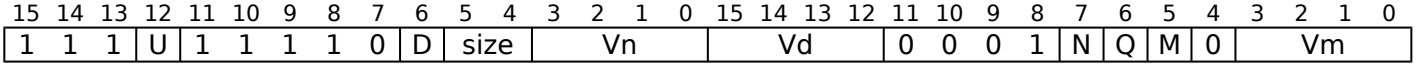
```
VRHADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VRHADD{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1



64-bit SIMD vector (Q == 0)

```
VRHADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VRHADD{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>	For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional. For encoding T1: see *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the operands, encoded in “U:size”:

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	00	U8
1	01	U16
1	10	U32

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Int(Elem[D[n+r],e,esize], unsigned);
            op2 = Int(Elem[D[m+r],e,esize], unsigned);
            result = op1 + op2 + 1;
            Elem[D[d+r],e,esize] = result<esize:1>;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

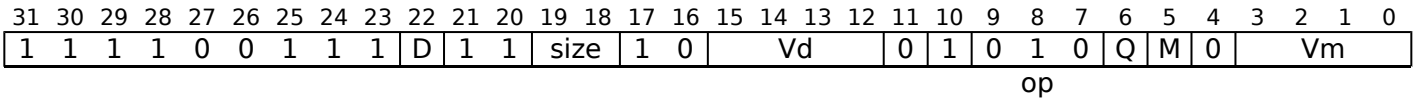
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VRINTA (Advanced SIMD)

Vector Round floating-point to integer towards Nearest with Ties to Away rounds a vector of floating-point values to integral floating-point values of the same size using the Round to Nearest with Ties to Away rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



64-bit SIMD vector (Q == 0)

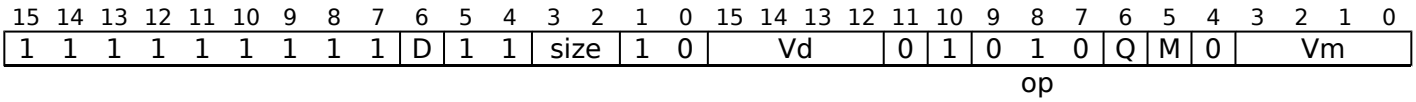
```
VRINTA{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VRINTA{<q>}.<dt> <Qd>, <Qm>
```

```
if op<2> != op<0> then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
// Rounding encoded differently from other VCVT and VRINT instructions
rounding = FPDecodeRM(op<2>:NOT(op<1>));  exact = FALSE;
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



64-bit SIMD vector (Q == 0)

```
VRINTA{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VRINTA{<q>}.<dt> <Qd>, <Qm>
```

```
if op<2> != op<0> then SEE "Related encodings";
if InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
// Rounding encoded differently from other VCVT and VRINT instructions
rounding = FPDecodeRM(op<2>:NOT(op<1>));  exact = FALSE;
case size of
    when '01' esize = 16; elements = 4;
    when '10' esize = 32; elements = 2;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Related encodings: See *Advanced SIMD two registers misc* for the T32 instruction set, or *Advanced SIMD two registers misc* for the A32 instruction set.

Assembler Symbols

<q> See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the vectors, encoded in “size”:

size	<dt>
01	F16
10	F32

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
EncodingSpecificOperations(); CheckAdvSIMDEnabled();
for r = 0 to regs-1
    for e = 0 to elements-1
        op1 = Elem[D[m+r],e,esize];
        result = FPRoundInt(op1, StandardFPSCRValue(), rounding, exact);
        Elem[D[d+r],e,esize] = result;
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

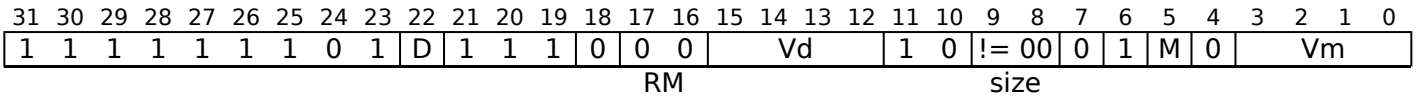
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VRINTA (floating-point)

Round floating-point to integer to Nearest with Ties to Away rounds a floating-point value to an integral floating-point value of the same size using the Round to Nearest with Ties to Away rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



Half-precision scalar (size == 01)
(FEAT_FP16)

```
VRINTA{<q>}.F16 <Sd>, <Sm>
```

Single-precision scalar (size == 10)

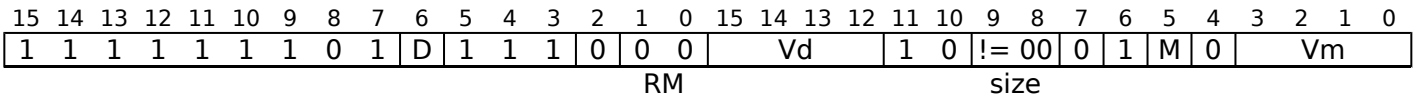
```
VRINTA{<q>}.F32 <Sd>, <Sm>
```

Double-precision scalar (size == 11)

```
VRINTA{<q>}.F64 <Dd>, <Dm>
```

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
rounding = FPDecodeRM(RM);  exact = FALSE;
case size of
    when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);
```

T1



Half-precision scalar (size == 01)
(FEAT_FP16)

```
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(); CheckVFPEnabled(TRUE);
case esize of
    when 16
        S[d] = Zeros(16) : FPRoundInt(S[m]<15:0>, FPSCR[], rounding, exact);
    when 32
        S[d] = FPRoundInt(S[m], FPSCR[], rounding, exact);
    when 64
        D[d] = FPRoundInt(D[m], FPSCR[], rounding, exact);
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

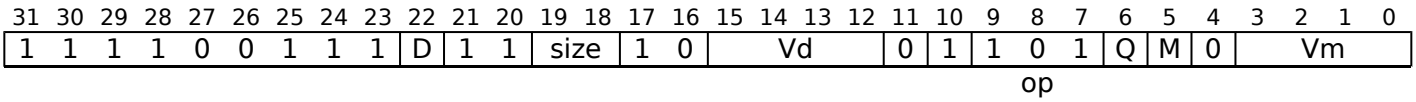
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VRINTM (Advanced SIMD)

Vector Round floating-point to integer towards -Infinity rounds a vector of floating-point values to integral floating-point values of the same size, using the Round towards -Infinity rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



64-bit SIMD vector (Q == 0)

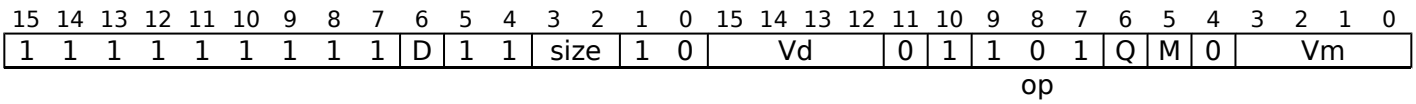
```
VRINTM{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VRINTM{<q>}.<dt> <Qd>, <Qm>
```

```
if op<2> != op<0> then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
// Rounding encoded differently from other VCVT and VRINT instructions
rounding = FPDecodeRM(op<2>:NOT(op<1>));  exact = FALSE;
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



64-bit SIMD vector (Q == 0)

```
VRINTM{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VRINTM{<q>}.<dt> <Qd>, <Qm>
```

```
if op<2> != op<0> then SEE "Related encodings";
if InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
// Rounding encoded differently from other VCVT and VRINT instructions
rounding = FPDecodeRM(op<2>:NOT(op<1>));  exact = FALSE;
case size of
    when '01' esize = 16; elements = 4;
    when '10' esize = 32; elements = 2;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Related encodings: See *Advanced SIMD two registers misc* for the T32 instruction set, or *Advanced SIMD two registers misc* for the A32 instruction set.

Assembler Symbols

<q> See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the vectors, encoded in “size”:

size	<dt>
01	F16
10	F32

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
EncodingSpecificOperations(); CheckAdvSIMDEnabled();
for r = 0 to regs-1
    for e = 0 to elements-1
        op1 = Elem[D[m+r],e,esize];
        result = FPRoundInt(op1, StandardFPSCRValue(), rounding, exact);
        Elem[D[d+r],e,esize] = result;
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

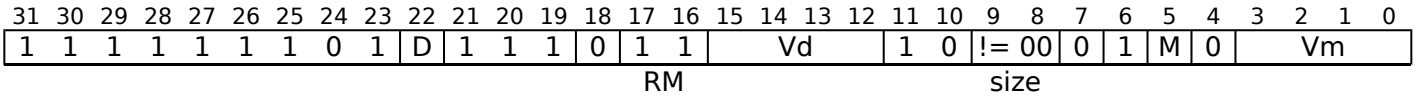
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VRINTM (floating-point)

Round floating-point to integer towards -Infinity rounds a floating-point value to an integral floating-point value of the same size using the Round towards -Infinity rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



Half-precision scalar (size == 01)
(FEAT_FP16)

```
VRINTM{<q>}.F16 <Sd>, <Sm>
```

Single-precision scalar (size == 10)

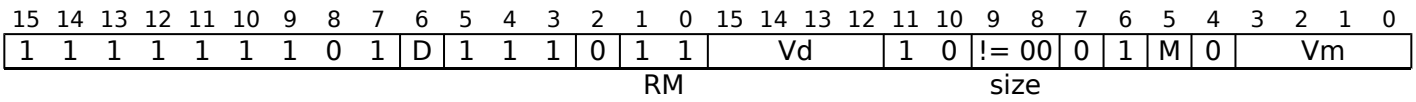
```
VRINTM{<q>}.F32 <Sd>, <Sm>
```

Double-precision scalar (size == 11)

```
VRINTM{<q>}.F64 <Dd>, <Dm>
```

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
rounding = FPDecodeRM(RM);  exact = FALSE;
case size of
    when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);
```

T1



Half-precision scalar (size == 01)
(FEAT_FP16)

```
VRINTM{<q>}.F16 <Sd>, <Sm>
```

Single-precision scalar (size == 10)

```
VRINTM{<q>}.F32 <Sd>, <Sm>
```

Double-precision scalar (size == 11)

```
VRINTM{<q>}.F64 <Dd>, <Dm>
```

```
if InITBlock() then UNPREDICTABLE;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
rounding = FPDecodeRM(RM);  exact = FALSE;
case size of
    when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<q>	See *Standard assembler syntax fields*.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
<Sm>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dm>	Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
case esize of
    when 16
        S[d] = Zeros(16) : FPRoundInt(S[m]<15:0>, FPSCR[], rounding, exact);
    when 32
        S[d] = FPRoundInt(S[m], FPSCR[], rounding, exact);
    when 64
        D[d] = FPRoundInt(D[m], FPSCR[], rounding, exact);
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

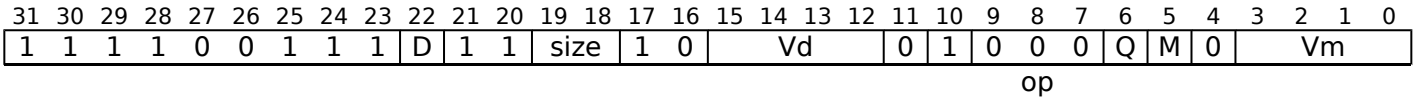
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VRINTN (Advanced SIMD)

Vector Round floating-point to integer to Nearest rounds a vector of floating-point values to integral floating-point values of the same size using the Round to Nearest rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



64-bit SIMD vector (Q == 0)

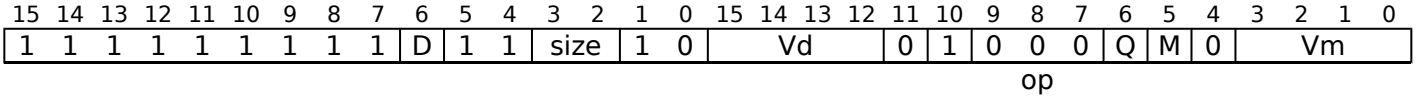
```
VRINTN{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VRINTN{<q>}.<dt> <Qd>, <Qm>
```

```
if op<2> != op<0> then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
// Rounding encoded differently from other VCVT and VRINT instructions
rounding = FPDecodeRM(op<2>:NOT(op<1>));  exact = FALSE;
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



64-bit SIMD vector (Q == 0)

```
VRINTN{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VRINTN{<q>}.<dt> <Qd>, <Qm>
```

```
if op<2> != op<0> then SEE "Related encodings";
if InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
// Rounding encoded differently from other VCVT and VRINT instructions
rounding = FPDecodeRM(op<2>:NOT(op<1>));  exact = FALSE;
case size of
    when '01' esize = 16; elements = 4;
    when '10' esize = 32; elements = 2;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Related encodings: See *Advanced SIMD two registers misc* for the T32 instruction set, or *Advanced SIMD two registers misc* for the A32 instruction set.

Assembler Symbols

<q> See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the vectors, encoded in “size”:

size	<dt>
01	F16
10	F32

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
EncodingSpecificOperations(); CheckAdvSIMDEnabled();
for r = 0 to regs-1
    for e = 0 to elements-1
        op1 = Elem[D[m+r],e,esize];
        result = FPRoundInt(op1, StandardFPSCRValue(), rounding, exact);
        Elem[D[d+r],e,esize] = result;
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

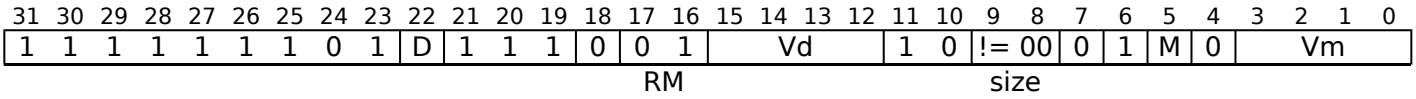
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VRINTN (floating-point)

Round floating-point to integer to Nearest rounds a floating-point value to an integral floating-point value of the same size using the Round to Nearest rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



Half-precision scalar (size == 01)
(FEAT_FP16)

```
VRINTN{<q>}.F16 <Sd>, <Sm>
```

Single-precision scalar (size == 10)

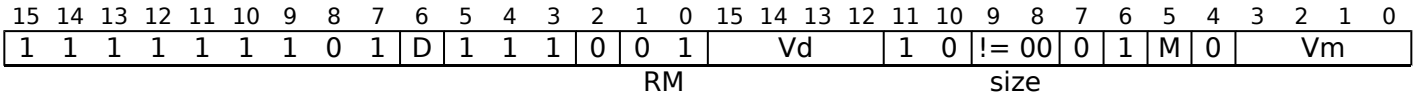
```
VRINTN{<q>}.F32 <Sd>, <Sm>
```

Double-precision scalar (size == 11)

```
VRINTN{<q>}.F64 <Dd>, <Dm>
```

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
rounding = FPDecodeRM(RM);  exact = FALSE;
case size of
    when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);
```

T1



Half-precision scalar (size == 01)
(FEAT_FP16)

```
VRINTN{<q>}.F16 <Sd>, <Sm>
```

Single-precision scalar (size == 10)

```
VRINTN{<q>}.F32 <Sd>, <Sm>
```

Double-precision scalar (size == 11)

```
VRINTN{<q>}.F64 <Dd>, <Dm>
```

```
if InITBlock() then UNPREDICTABLE;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
rounding = FPDecodeRM(RM);  exact = FALSE;
case size of
    when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<q>	See *Standard assembler syntax fields*.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
<Sm>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dm>	Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
case esize of
    when 16
        S[d] = Zeros(16) : FPRoundInt(S[m]<15:0>, FPSCR[], rounding, exact);
    when 32
        S[d] = FPRoundInt(S[m], FPSCR[], rounding, exact);
    when 64
        D[d] = FPRoundInt(D[m], FPSCR[], rounding, exact);
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

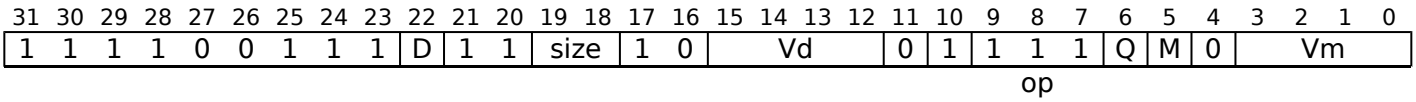
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VRINTP (Advanced SIMD)

Vector Round floating-point to integer towards +Infinity rounds a vector of floating-point values to integral floating-point values of the same size using the Round towards +Infinity rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



64-bit SIMD vector (Q == 0)

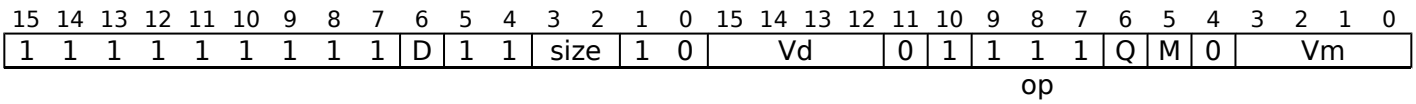
```
VRINTP{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VRINTP{<q>}.<dt> <Qd>, <Qm>
```

```
if op<2> != op<0> then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
// Rounding encoded differently from other VCVT and VRINT instructions
rounding = FPDecodeRM(op<2>:NOT(op<1>));  exact = FALSE;
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



64-bit SIMD vector (Q == 0)

```
VRINTP{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VRINTP{<q>}.<dt> <Qd>, <Qm>
```

```
if op<2> != op<0> then SEE "Related encodings";
if InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
// Rounding encoded differently from other VCVT and VRINT instructions
rounding = FPDecodeRM(op<2>:NOT(op<1>));  exact = FALSE;
case size of
    when '01' esize = 16; elements = 4;
    when '10' esize = 32; elements = 2;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Related encodings: See *Advanced SIMD two registers misc* for the T32 instruction set, or *Advanced SIMD two registers misc* for the A32 instruction set.

Assembler Symbols

<q> See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the vectors, encoded in “size”:

size	<dt>
01	F16
10	F32

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
EncodingSpecificOperations(); CheckAdvSIMDEnabled();
for r = 0 to regs-1
    for e = 0 to elements-1
        op1 = Elem[D[m+r],e,esize];
        result = FPRoundInt(op1, StandardFPSCRValue(), rounding, exact);
        Elem[D[d+r],e,esize] = result;
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

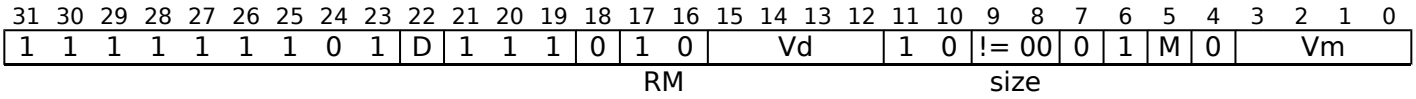
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VRINTP (floating-point)

Round floating-point to integer towards +Infinity rounds a floating-point value to an integral floating-point value of the same size using the Round towards +Infinity rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



Half-precision scalar (size == 01)
(FEAT_FP16)

```
VRINTP{<q>}.F16 <Sd>, <Sm>
```

Single-precision scalar (size == 10)

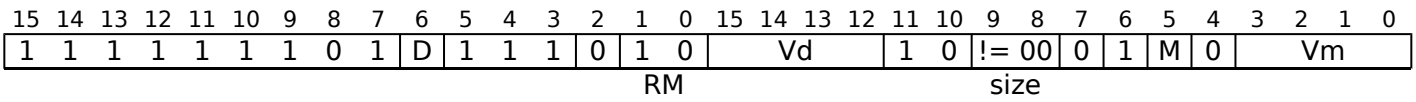
```
VRINTP{<q>}.F32 <Sd>, <Sm>
```

Double-precision scalar (size == 11)

```
VRINTP{<q>}.F64 <Dd>, <Dm>
```

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
rounding = FPDecodeRM(RM);  exact = FALSE;
case size of
    when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);
```

T1



Half-precision scalar (size == 01)
(FEAT_FP16)

```
VRINTP{<q>}.F16 <Sd>, <Sm>
```

Single-precision scalar (size == 10)

```
VRINTP{<q>}.F32 <Sd>, <Sm>
```

Double-precision scalar (size == 11)

```
VRINTP{<q>}.F64 <Dd>, <Dm>
```

```
if InITBlock() then UNPREDICTABLE;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
rounding = FPDecodeRM(RM);  exact = FALSE;
case size of
    when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<q>	See *Standard assembler syntax fields*.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
<Sm>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dm>	Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
case esize of
    when 16
        S[d] = Zeros(16) : FPRoundInt(S[m]<15:0>, FPSCR[], rounding, exact);
    when 32
        S[d] = FPRoundInt(S[m], FPSCR[], rounding, exact);
    when 64
        D[d] = FPRoundInt(D[m], FPSCR[], rounding, exact);
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

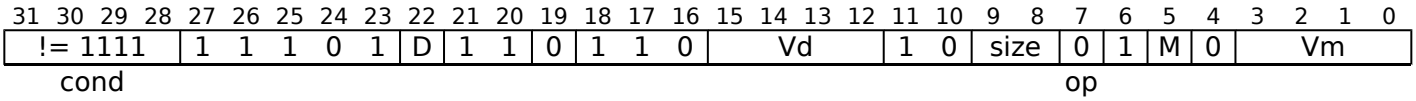
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VRINTR

Round floating-point to integer rounds a floating-point value to an integral floating-point value of the same size using the rounding mode specified in the FPSCR. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



Half-precision scalar (size == 01)
(FEAT_FP16)

```
VRINTR{<c>}{<q>}.F16 <Sd>, <Sm>
```

Single-precision scalar (size == 10)

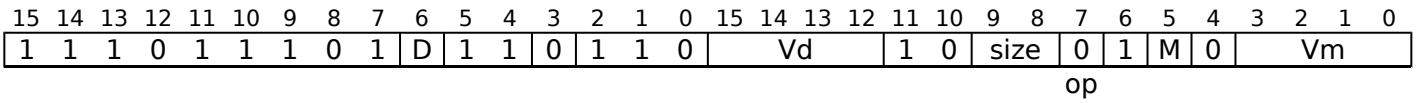
```
VRINTR{<c>}{<q>}.F32 <Sd>, <Sm>
```

Double-precision scalar (size == 11)

```
VRINTR{<c>}{<q>}.F64 <Dd>, <Dm>
```

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
rounding = if op == '1' then FPRounding_ZERO else FPRoundingMode(FPSCR[]);
exact = FALSE;
case size of
    when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);
```

T1



Half-precision scalar (size == 01)
(FEAT_FP16)

```
VRINTR{<c>}{<q>}.F16 <Sd>, <Sm>
```

Single-precision scalar (size == 10)

```
VRINTR{<c>}{<q>}.F32 <Sd>, <Sm>
```

Double-precision scalar (size == 11)

```
VRINTR{<c>}{<q>}.F64 <Dd>, <Dm>
```

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock()  then UNPREDICTABLE;
rounding = if op == '1' then FPRounding_ZERO else FPRoundingMode(FPSCR[]);
exact = FALSE;
case size of
    when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
<Sm>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dm>	Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
    case esize of
        when 16
            S[d] = Zeros(16) : FPRoundInt(S[m]<15:0>, FPSCR[], rounding, exact);
        when 32
            S[d] = FPRoundInt(S[m], FPSCR[], rounding, exact);
        when 64
            D[d] = FPRoundInt(D[m], FPSCR[], rounding, exact);
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

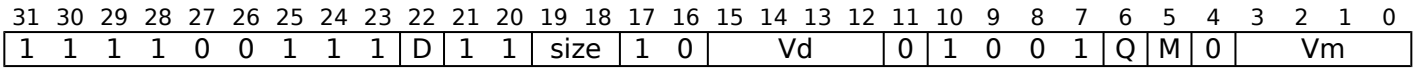
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VRINTX (Advanced SIMD)

Vector round floating-point to integer inexact rounds a vector of floating-point values to integral floating-point values of the same size, using the Round to Nearest rounding mode, and raises the Inexact exception when the result value is not numerically equal to the input value. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



64-bit SIMD vector (Q == 0)

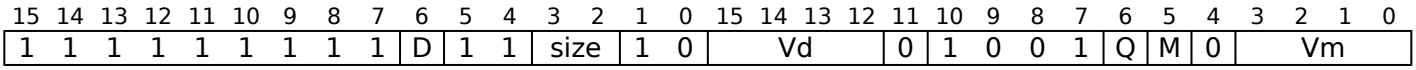
```
VRINTX{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VRINTX{<q>}.<dt> <Qd>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
rounding = FPRounding_TIEEVEN;  exact = TRUE;
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



64-bit SIMD vector (Q == 0)

```
VRINTX{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VRINTX{<q>}.<dt> <Qd>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
rounding = FPRounding_TIEEVEN;  exact = TRUE;
case size of
    when '01' esize = 16; elements = 4;
    when '10' esize = 32; elements = 2;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
if InITBlock() then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<q> See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the vectors, encoded in “size”:

size	<dt>
01	F16
10	F32

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
EncodingSpecificOperations(); CheckAdvSIMDEnabled();
for r = 0 to regs-1
    for e = 0 to elements-1
        op1 = Elem[D[m+r],e,esize];
        result = FPRoundInt(op1, StandardFPSCRValue(), rounding, exact);
        Elem[D[d+r],e,esize] = result;
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

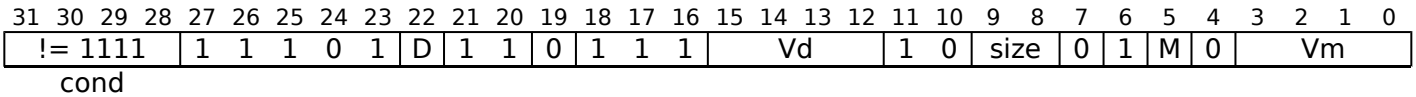
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VRINTX (floating-point)

Round floating-point to integer inexact rounds a floating-point value to an integral floating-point value of the same size, using the rounding mode specified in the FPSCR, and raises an Inexact exception when the result value is not numerically equal to the input value. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



Half-precision scalar (size == 01)
(FEAT_FP16)

```
VRINTX{<c>}{<q>}.F16 <Sd>, <Sm>
```

Single-precision scalar (size == 10)

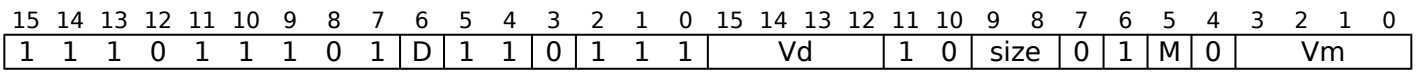
```
VRINTX{<c>}{<q>}.F32 <Sd>, <Sm>
```

Double-precision scalar (size == 11)

```
VRINTX{<c>}{<q>}.F64 <Dd>, <Dm>
```

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
exact = TRUE;
case size of
    when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);
```

T1



Half-precision scalar (size == 01)
(FEAT_FP16)

```
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 *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
<Sm>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dm>	Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
    rounding = FPRoundingMode(FPSCR[]);
    case esize of
        when 16
            S[d] = Zeros(16) : FPRoundInt(S[m]<15:0>, FPSCR[], rounding, exact);
        when 32
            S[d] = FPRoundInt(S[m], FPSCR[], rounding, exact);
        when 64
            D[d] = FPRoundInt(D[m], FPSCR[], rounding, exact);
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

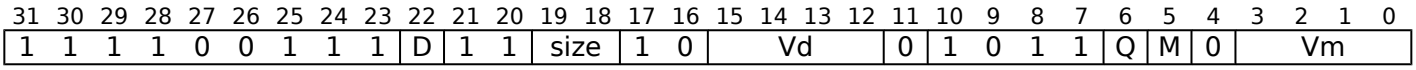
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VRINTZ (Advanced SIMD)

Vector round floating-point to integer towards Zero rounds a vector of floating-point values to integral floating-point values of the same size, using the Round towards Zero rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



64-bit SIMD vector (Q == 0)

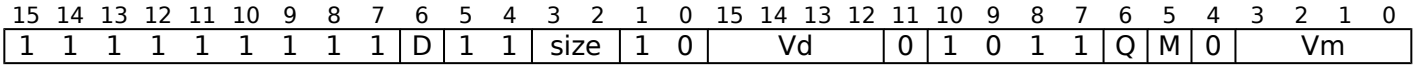
```
VRINTZ{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VRINTZ{<q>}.<dt> <Qd>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
rounding = FPRounding_ZERO;  exact = FALSE;
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



64-bit SIMD vector (Q == 0)

```
VRINTZ{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VRINTZ{<q>}.<dt> <Qd>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
rounding = FPRounding_ZERO;  exact = FALSE;
case size of
    when '01' esize = 16; elements = 4;
    when '10' esize = 32; elements = 2;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
if InITBlock() then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<q> See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the vectors, encoded in “size”:

size	<dt>
01	F16
10	F32

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
EncodingSpecificOperations(); CheckAdvSIMDEnabled();
for r = 0 to regs-1
    for e = 0 to elements-1
        op1 = Elem[D[m+r],e,esize];
        result = FPRoundInt(op1, StandardFPSCRValue(), rounding, exact);
        Elem[D[d+r],e,esize] = result;
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

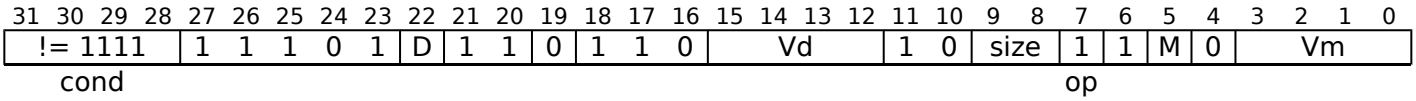
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VRINTZ (floating-point)

Round floating-point to integer towards Zero rounds a floating-point value to an integral floating-point value of the same size, using the Round towards Zero rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



Half-precision scalar (size == 01)
(FEAT_FP16)

```
VRINTZ{<c>}{<q>}.F16 <Sd>, <Sm>
```

Single-precision scalar (size == 10)

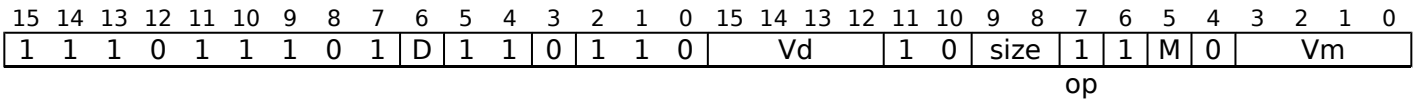
```
VRINTZ{<c>}{<q>}.F32 <Sd>, <Sm>
```

Double-precision scalar (size == 11)

```
VRINTZ{<c>}{<q>}.F64 <Dd>, <Dm>
```

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
rounding = if op == '1' then FPRounding_ZERO else FPRoundingMode(FPSCR[]);
exact = FALSE;
case size of
    when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);
```

T1



Half-precision scalar (size == 01)
(FEAT_FP16)

```
VRINTZ{<c>}{<q>}.F16 <Sd>, <Sm>
```

Single-precision scalar (size == 10)

```
VRINTZ{<c>}{<q>}.F32 <Sd>, <Sm>
```

Double-precision scalar (size == 11)

```
VRINTZ{<c>}{<q>}.F64 <Dd>, <Dm>
```

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock()  then UNPREDICTABLE;
rounding = if op == '1' then FPRounding_ZERO else FPRoundingMode(FPSCR[]);
exact = FALSE;
case size of
    when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
<Sm>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dm>	Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
    case esize of
        when 16
            S[d] = Zeros(16) : FPRoundInt(S[m]<15:0>, FPSCR[], rounding, exact);
        when 32
            S[d] = FPRoundInt(S[m], FPSCR[], rounding, exact);
        when 64
            D[d] = FPRoundInt(D[m], FPSCR[], rounding, exact);
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VRSHL

Vector Rounding Shift Left takes each element in a vector, shifts them by a value from the least significant byte of the corresponding element of a second vector, and places the results in the destination vector. If the shift value is positive, the operation is a left shift. If the shift value is negative, it is a rounding right shift. For a truncating shift, see VSHL.

The first operand and result elements are the same data type, and can be any one of:

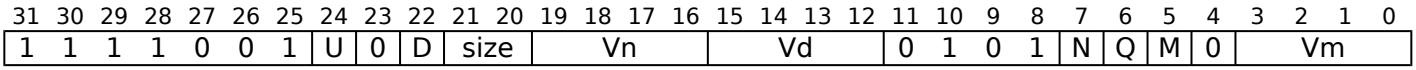
- 8-bit, 16-bit, 32-bit, or 64-bit signed integers.
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

The second operand is always a signed integer of the same size.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



64-bit SIMD vector (Q == 0)

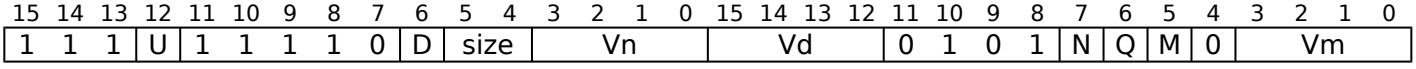
```
VRSHL{<c>}{<q>}.<dt> {<Dd>,} <Dm>, <Dn>
```

128-bit SIMD vector (Q == 1)

```
VRSHL{<c>}{<q>}.<dt> {<Qd>,} <Qm>, <Qn>
```

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1' || Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;
```

T1



64-bit SIMD vector (Q == 0)

```
VRSHL{<c>}{<q>}.<dt> {<Dd>,} <Dm>, <Dn>
```

128-bit SIMD vector (Q == 1)

```
VRSHL{<c>}{<q>}.<dt> {<Qd>,} <Qm>, <Qn>
```

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1' || Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>	For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional. For encoding T1: see *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<dt>	Is the data type for the elements of the vectors, encoded in “U:size”:

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
0	11	S64
1	00	U8
1	01	U16
1	10	U32
1	11	U64

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            shift = SInt(Elem[D[n+r],e,esize]<7:0>);
            round_const = 1 << (-shift-1); // 0 for left shift, 2^(n-1) for right shift
            result = (Int(Elem[D[m+r],e,esize], unsigned) + round_const) << shift;
            Elem[D[d+r],e,esize] = result<esize-1:0>;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VRSHR

Vector Rounding Shift Right takes each element in a vector, right shifts them by an immediate value, and places the rounded results in the destination vector. For truncated results, see *VSHR*.

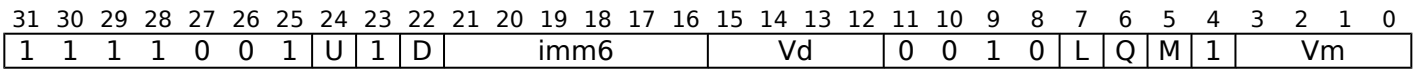
The operand and result elements must be the same size, and can be any one of:

- 8-bit, 16-bit, 32-bit, or 64-bit signed integers.
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



64-bit SIMD vector (!(imm6 == 000xxx && L == 0) && Q == 0)

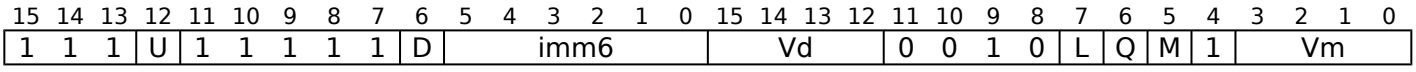
```
VRSHR{<c>}{<q>}.<type><size> {<Dd>,} <Dm>, #<imm>
```

128-bit SIMD vector (!(imm6 == 000xxx && L == 0) && Q == 1)

```
VRSHR{<c>}{<q>}.<type><size> {<Qd>,} <Qm>, #<imm>
```

```
if (L:imm6) == '0000xxx' then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
case L:imm6 of
    when '0001xxx'  esize = 8;  elements = 8;  shift_amount = 16 - UInt(imm6);
    when '001xxxx'  esize = 16;  elements = 4;  shift_amount = 32 - UInt(imm6);
    when '01xxxxx'  esize = 32;  elements = 2;  shift_amount = 64 - UInt(imm6);
    when '1xxxxxx'  esize = 64;  elements = 1;  shift_amount = 64 - UInt(imm6);
unsigned = (U == '1');  d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1



64-bit SIMD vector (!(imm6 == 000xxx && L == 0) && Q == 0)

```
VRSHR{<c>}{<q>}.<type><size> {<Dd>,} <Dm>, #<imm>
```

128-bit SIMD vector (!(imm6 == 000xxx && L == 0) && Q == 1)

```
VRSHR{<c>}{<q>}.<type><size> {<Qd>,} <Qm>, #<imm>
```

```
if (L:imm6) == '0000xxx' then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
case L:imm6 of
    when '0001xxx'  esize = 8;  elements = 8;  shift_amount = 16 - UInt(imm6);
    when '001xxxx'  esize = 16;  elements = 4;  shift_amount = 32 - UInt(imm6);
    when '01xxxxx'  esize = 32;  elements = 2;  shift_amount = 64 - UInt(imm6);
    when '1xxxxxx'  esize = 64;  elements = 1;  shift_amount = 64 - UInt(imm6);
unsigned = (U == '1');  d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Related encodings: See *Advanced SIMD one register and modified immediate* for the T32 instruction set, or *Advanced SIMD one register and modified immediate* for the A32 instruction set.

Assembler Symbols

<c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.
For encoding T1: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<type> Is the data type for the elements of the vectors, encoded in “U”:

U	<type>
0	S
1	U

<size> Is the data size for the elements of the vectors, encoded in “L:imm6<5:3>”:

L	imm6<5:3>	<size>
0	001	8
0	01x	16
0	1xx	32
1	xxx	64

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

<imm> Is an immediate value, in the range 1 to <size>, encoded in the "imm6" field as <size> - <imm>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    round_const = 1 << (shift_amount - 1);
    for r = 0 to regs-1
        for e = 0 to elements-1
            result = (Int(Elem[D[m+r],e,esize], unsigned) + round_const) >> shift_amount;
            Elem[D[d+r],e,esize] = result<esize-1:0>;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VRSHR (zero)

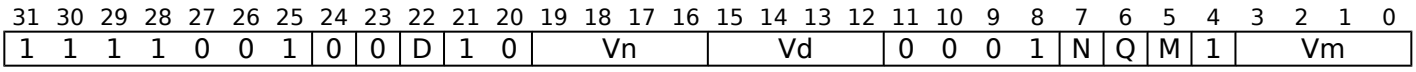
Vector Rounding Shift Right copies the contents of one SIMD register to another

This is a pseudo-instruction of VORR (register). This means:

- The encodings in this description are named to match the encodings of VORR (register).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of VORR (register) gives the operational pseudocode for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



64-bit SIMD vector (Q == 0)

```
VRSHR{<c>}{<q>}.<dt> <Dd>, <Dm>, #0
```

is equivalent to

```
VORR{<c>}{<q>}{.<dt>} <Dd>, <Dm>, <Dm>
```

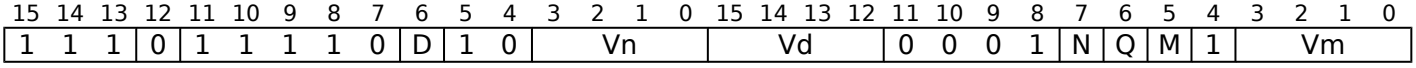
128-bit SIMD vector (Q == 1)

```
VRSHR{<c>}{<q>}.<dt> <Qd>, <Qm>, #0
```

is equivalent to

```
VORR{<c>}{<q>}{.<dt>} <Qd>, <Qm>, <Qm>
```

T1



64-bit SIMD vector (Q == 0)

```
VRSHR{<c>}{<q>}.<dt> <Dd>, <Dm>, #0
```

is equivalent to

```
VORR{<c>}{<q>}{.<dt>} <Dd>, <Dm>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VRSHR{<c>}{<q>}.<dt> <Qd>, <Qm>, #0
```

is equivalent to

```
VORR{<c>}{<q>}{.<dt>} <Qd>, <Qm>, <Qm>
```

Assembler Symbols

<c>	For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional. For encoding T1: see *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.

<dt>	Is the data type for the elements of the vectors, and must be one of: S8, S16, S32, S64, U8, U16, U32 or U64.
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qm>	Is the 128-bit name of the SIMD&FP source register, encoded in the "N:Vn" and "M:Vm" field as <Qm>*2.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dm>	Is the 64-bit name of the SIMD&FP source register, encoded in the "N:Vn" and "M:Vm" field.

Operation

The description of VORR (register) gives the operational pseudocode for this instruction.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VRSHRN

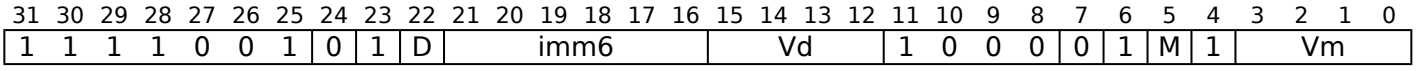
Vector Rounding Shift Right and Narrow takes each element in a vector, right shifts them by an immediate value, and places the rounded results in the destination vector. For truncated results, see *VSHRN*.

The operand elements can be 16-bit, 32-bit, or 64-bit integers. There is no distinction between signed and unsigned integers. The destination elements are half the size of the source elements.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

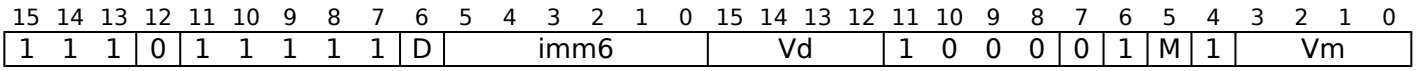


A1 (imm6 != 000xxx)

```
VRSHRN{<c>}{<q>}.I<size> <Dd>, <Qm>, #<imm>
```

```
if imm6 == '000xxx' then SEE "Related encodings";
if Vm<0> == '1' then UNDEFINED;
case imm6 of
    when '001xxx'  esize = 8;  elements = 8;  shift_amount = 16 - UInt(imm6);
    when '01xxxx'  esize = 16;  elements = 4;  shift_amount = 32 - UInt(imm6);
    when '1xxxxx'  esize = 32;  elements = 2;  shift_amount = 64 - UInt(imm6);
d = UInt(D:Vd);  m = UInt(M:Vm);
```

T1



T1 (imm6 != 000xxx)

```
VRSHRN{<c>}{<q>}.I<size> <Dd>, <Qm>, #<imm>
```

```
if imm6 == '000xxx' then SEE "Related encodings";
if Vm<0> == '1' then UNDEFINED;
case imm6 of
    when '001xxx'  esize = 8;  elements = 8;  shift_amount = 16 - UInt(imm6);
    when '01xxxx'  esize = 16;  elements = 4;  shift_amount = 32 - UInt(imm6);
    when '1xxxxx'  esize = 32;  elements = 2;  shift_amount = 64 - UInt(imm6);
d = UInt(D:Vd);  m = UInt(M:Vm);
```

Related encodings: See *Advanced SIMD one register and modified immediate* for the T32 instruction set, or *Advanced SIMD one register and modified immediate* for the A32 instruction set.

Assembler Symbols

<c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<size> Is the data size for the elements of the vectors, encoded in “imm6<5:3>”:

imm6<5:3>	<size>
001	16
01x	32
1xx	64

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<imm> Is an immediate value, in the range 1 to <size>/2, encoded in the "imm6" field as <size>/2 - <imm>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    round_const = 1 << (shift_amount-1);
    for e = 0 to elements-1
        result = LSR(Elem[Qin[m>>1],e,2*esize] + round_const, shift_amount);
        Elem[D[d],e,esize] = result<esize-1:0>;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VRSHRN (zero)

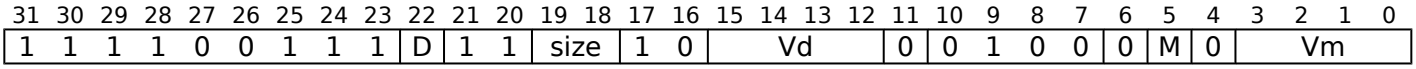
Vector Rounding Shift Right and Narrow takes each element in a vector, right shifts them by an immediate value, and places the rounded results in the destination vector

This is a pseudo-instruction of VMOVN. This means:

- The encodings in this description are named to match the encodings of VMOVN.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of VMOVN gives the operational pseudocode for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



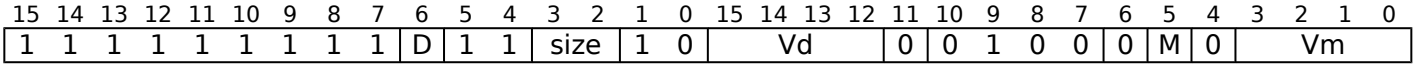
A1

```
VRSHRN{<c>}{<q>}.<dt> <Dd>, <Qm>, #0
```

is equivalent to

```
VMOVN{<c>}{<q>}.<dt> <Dd>, <Qm>
```

T1



T1

```
VRSHRN{<c>}{<q>}.<dt> <Dd>, <Qm>, #0
```

is equivalent to

```
VMOVN{<c>}{<q>}.<dt> <Dd>, <Qm>
```

Assembler Symbols

<c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.
For encoding T1: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the operand, encoded in “size”:

size	<dt>
00	I16
01	I32
10	I64
11	RESERVED

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

The description of VMOVN gives the operational pseudocode for this instruction.

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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.

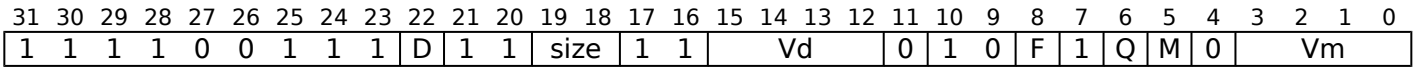
The operand and result elements are the same type, and can be floating-point numbers or unsigned integers.

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



64-bit SIMD vector (Q == 0)

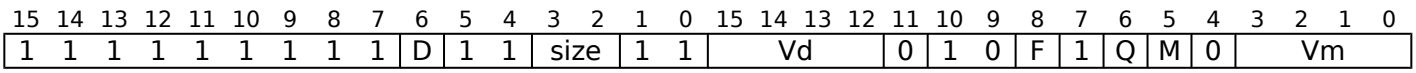
```
VRSQRTE{<c>}{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VRSQRTE{<c>}{<q>}.<dt> <Qd>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && (!HaveFP16Ext() || F == '0')) || size IN {'00', '11'} then UNDEFINED;
floating_point = (F == '1');
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



64-bit SIMD vector (Q == 0)

```
VRSQRTE{<c>}{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VRSQRTE{<c>}{<q>}.<dt> <Qd>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && (!HaveFP16Ext() || F == '0')) || size IN {'00', '11'} then UNDEFINED;
if size == '01' && InITBlock()  then UNPREDICTABLE;
floating_point = (F == '1');
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_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VRSQRTS

Vector Reciprocal Square Root Step multiplies the elements of one vector by the corresponding elements of another vector, subtracts each of the products from 3.0, divides these results by 2.0, and places the results into the elements of the destination vector.

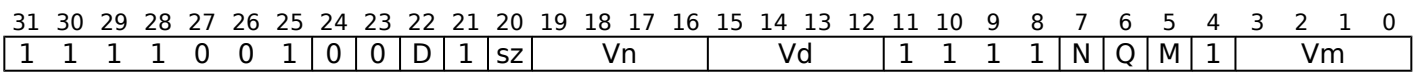
The operand and result elements are floating-point numbers.

For details of the operation performed by this instruction see *Floating-point reciprocal estimate and step*.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



64-bit SIMD vector (Q == 0)

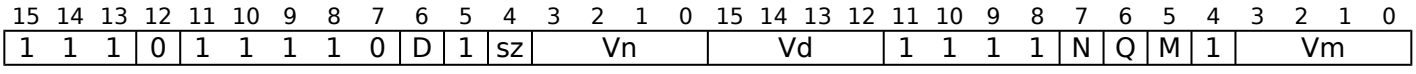
```
VRSQRTS{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VRSQRTS{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1



64-bit SIMD vector (Q == 0)

```
VRSQRTS{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VRSQRTS{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.
For encoding T1: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the vectors, encoded in “sz”:

sz	<dt>
0	F32
1	F16

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Newton-Raphson iteration
For details of the operation performed and how it can be used in a Newton-Raphson iteration to calculate the reciprocal of the square root of a number, see *Floating-point reciprocal estimate and step*.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            Elem[D[d+r],e,esize] = FPRSqrtStep(Elem[D[n+r],e,esize], Elem[D[m+r],e,esize]);
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VRSRA

Vector Rounding Shift Right and Accumulate takes each element in a vector, right shifts them by an immediate value, and accumulates the rounded results into the destination vector.For truncated results, see *VSRA*.

The operand and result elements must all be the same type, and can be any one of:

- 8-bit, 16-bit, 32-bit, or 64-bit signed integers.
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	1	D	imm6						Vd				0	0	1	1	L	Q	M	1	Vm			

64-bit SIMD vector (!(imm6 == 000xxx && L == 0) && Q == 0)

```
VRSRA{<c>}{<q>}.<type><size> {<Dd>,} <Dm>, #<imm>
```

128-bit SIMD vector (!(imm6 == 000xxx && L == 0) && Q == 1)

```
VRSRA{<c>}{<q>}.<type><size> {<Qd>,} <Qm>, #<imm>
```

```
if (L:imm6) == '0000xxx' then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
case L:imm6 of
    when '0001xxx'  esize = 8;  elements = 8;  shift_amount = 16 - UInt(imm6);
    when '001xxxx'  esize = 16;  elements = 4;  shift_amount = 32 - UInt(imm6);
    when '01xxxxx'  esize = 32;  elements = 2;  shift_amount = 64 - UInt(imm6);
    when '1xxxxxx'  esize = 64;  elements = 1;  shift_amount = 64 - UInt(imm6);
unsigned = (U == '1');  d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	1	D	imm6						Vd				0	0	1	1	L	Q	M	1	Vm			

64-bit SIMD vector (!(imm6 == 000xxx && L == 0) && Q == 0)

```
VRSRA{<c>}{<q>}.<type><size> {<Dd>,} <Dm>, #<imm>
```

128-bit SIMD vector (!(imm6 == 000xxx && L == 0) && Q == 1)

```
VRSRA{<c>}{<q>}.<type><size> {<Qd>,} <Qm>, #<imm>
```

```
if (L:imm6) == '0000xxx' then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
case L:imm6 of
    when '0001xxx'  esize = 8;  elements = 8;  shift_amount = 16 - UInt(imm6);
    when '001xxxx'  esize = 16;  elements = 4;  shift_amount = 32 - UInt(imm6);
    when '01xxxxx'  esize = 32;  elements = 2;  shift_amount = 64 - UInt(imm6);
    when '1xxxxxx'  esize = 64;  elements = 1;  shift_amount = 64 - UInt(imm6);
unsigned = (U == '1');  d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Related encodings: See *Advanced SIMD one register and modified immediate* for the T32 instruction set, or *Advanced SIMD one register and modified immediate* for the A32 instruction set.

Assembler Symbols

<c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.
For encoding T1: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<type> Is the data type for the elements of the vectors, encoded in “U”:

U	<type>
0	S
1	U

<size> Is the data size for the elements of the vectors, encoded in “L:imm6<5:3>”:

L	imm6<5:3>	<size>
0	001	8
0	01x	16
0	1xx	32
1	xxx	64

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

<imm> Is an immediate value, in the range 1 to <size>, encoded in the "imm6" field as <size> - <imm>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    round_const = 1 << (shift_amount - 1);
    for r = 0 to regs-1
        for e = 0 to elements-1
            result = (Int(Elem[D[m+r],e,esize], unsigned) + round_const) >> shift_amount;
            Elem[D[d+r],e,esize] = Elem[D[d+r],e,esize] + result;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VRSUBHN

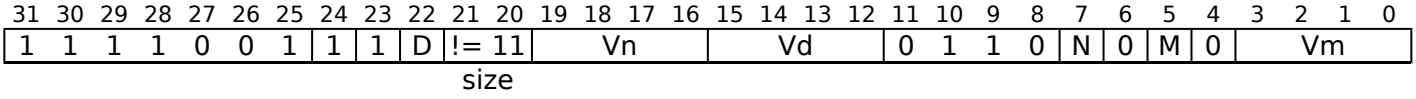
Vector Rounding Subtract and Narrow, returning High Half subtracts the elements of one quadword vector from the corresponding elements of another quadword vector, takes the most significant half of each result, and places the final results in a doubleword vector. The results are rounded. For truncated results, see *VSUBHN*.

The operand elements can be 16-bit, 32-bit, or 64-bit integers. There is no distinction between signed and unsigned integers.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

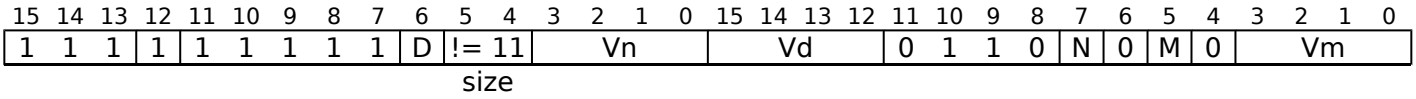


A1

```
VRSUBHN{<c>}{<q>}.<dt> <Dd>, <Qn>, <Qm>
```

```
if size == '11' then SEE "Related encodings";
if Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

T1



T1

```
VRSUBHN{<c>}{<q>}.<dt> <Dd>, <Qn>, <Qm>
```

```
if size == '11' then SEE "Related encodings";
if Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

Related encodings: See *Advanced SIMD data-processing* for the T32 instruction set, or *Advanced SIMD data-processing* for the A32 instruction set.

Assembler Symbols

<c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the operands, encoded in “size”:

size	<dt>
00	I16
01	I32
10	I64

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    round_const = 1 << (esize-1);
    for e = 0 to elements-1
        result = Elem[Qin[n>>1],e,2*esize] - Elem[Qin[m>>1],e,2*esize] + round_const;
        Elem[D[d],e,esize] = result<2*esize-1:esize>;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

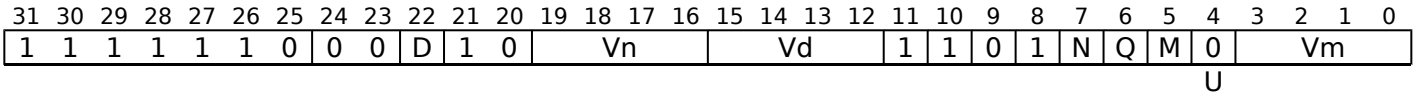
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VSDOT (vector)

Dot Product vector form with signed integers. This instruction performs the dot product of the four 8-bit elements in each 32-bit element of the first source register with the four 8-bit elements of the corresponding 32-bit element in the second source register, accumulating the result into the corresponding 32-bit element of the destination register.
In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.
ID_ISAR6.DP indicates whether this instruction is supported.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1
(FEAT_DotProd)



64-bit SIMD vector (Q == 0)

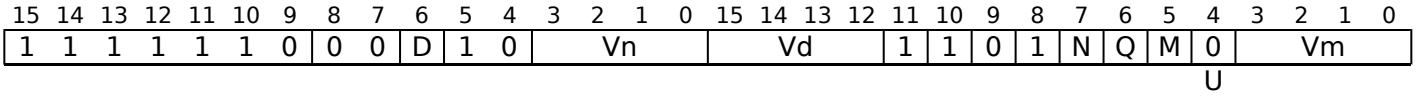
```
VSDOT{<q>}.S8 <Dd>, <Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VSDOT{<q>}.S8 <Qd>, <Qn>, <Qm>
```

```
if !HaveDOTPExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
boolean signed = U=='0';
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(M:Vm);
integer esize = 32;
integer regs = if Q == '1' then 2 else 1;
```

T1
(FEAT_DotProd)



64-bit SIMD vector (Q == 0)

```
VSDOT{<q>}.S8 <Dd>, <Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VSDOT{<q>}.S8 <Qd>, <Qn>, <Qm>
```

```
if InITBlock() then UNPREDICTABLE;
if !HaveDOTPExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
boolean signed = U=='0';
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(M:Vm);
integer esize = 32;
integer regs = if Q == '1' then 2 else 1;
```

Assembler Symbols

<q>	See *Standard assembler syntax fields*.
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qn>	Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
<Qm>	Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
bits(64) operand1;
bits(64) operand2;
bits(64) result;
CheckAdvSIMDEnabled();
for r = 0 to regs-1
    operand1 = D[n+r];
    operand2 = D[m+r];
    result = D[d+r];
    integer element1, element2;
    for e = 0 to 1
        integer res = 0;
        for i = 0 to 3
            if signed then
                element1 = SInt(Elem[operand1, 4 * e + i, esize DIV 4]);
                element2 = SInt(Elem[operand2, 4 * e + i, esize DIV 4]);
            else
                element1 = UInt(Elem[operand1, 4 * e + i, esize DIV 4]);
                element2 = UInt(Elem[operand2, 4 * e + i, esize DIV 4]);
            res = res + element1 * element2;
        Elem[result, e, esize] = Elem[result, e, esize] + res;
    D[d+r] = result;
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VSDOT (by element)

Dot Product index form with signed integers. This instruction performs the dot product of the four 8-bit elements in each 32-bit element of the first source register with the four 8-bit elements of an indexed 32-bit element in the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

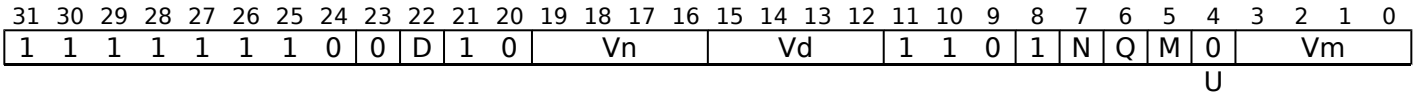
In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

ID_ISAR6.DP indicates whether this instruction is supported.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

(FEAT_DotProd)



64-bit SIMD vector (Q == 0)

```
VSDOT{<q>}.S8 <Dd>, <Dn>, <Dm>[<index>]
```

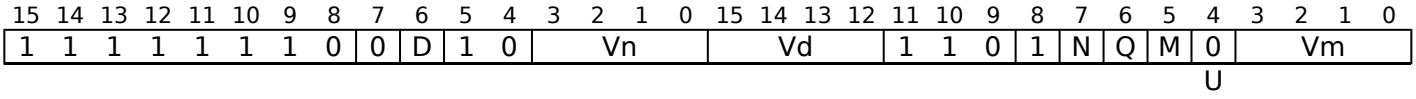
128-bit SIMD vector (Q == 1)

```
VSDOT{<q>}.S8 <Qd>, <Qn>, <Dm>[<index>]
```

```
if !HaveDOTPExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
boolean signed = (U=='0');
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(Vm<3:0>);
integer index = UInt(M);
integer esize = 32;
integer regs = if Q == '1' then 2 else 1;
```

T1

(FEAT_DotProd)



64-bit SIMD vector (Q == 0)

```
VSDOT{<q>}.S8 <Dd>, <Dn>, <Dm>[<index>]
```

128-bit SIMD vector (Q == 1)

```
VSDOT{<q>}.S8 <Qd>, <Qn>, <Dm>[<index>]
```

```
if InITBlock() then UNPREDICTABLE;
if !HaveDOTPExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
boolean signed = (U=='0');
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(Vm<3:0>);
integer index = UInt(M);
integer esize = 32;
integer regs = if Q == '1' then 2 else 1;
```

Assembler Symbols

<q>	See *Standard assembler syntax fields*.
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qn>	Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "Vm" field.
<index>	Is the element index in the range 0 to 1, encoded in the "M" field.

Operation

```
bits(64) operand1;
bits(64) operand2 = D[m];
bits(64) result;
CheckAdvSIMDEnabled();
for r = 0 to regs-1
    operand1 = D[n+r];
    result = D[d+r];
    integer element1, element2;
    for e = 0 to 1
        integer res = 0;
        for i = 0 to 3
            if signed then
                element1 = SInt(Elem[operand1, 4 * e + i, esize DIV 4]);
                element2 = SInt(Elem[operand2, 4 * index + i, esize DIV 4]);
            else
                element1 = UInt(Elem[operand1, 4 * e + i, esize DIV 4]);
                element2 = UInt(Elem[operand2, 4 * index + i, esize DIV 4]);
            res = res + element1 * element2;
        Elem[result, e, esize] = Elem[result, e, esize] + res;
    D[d+r] = result;
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

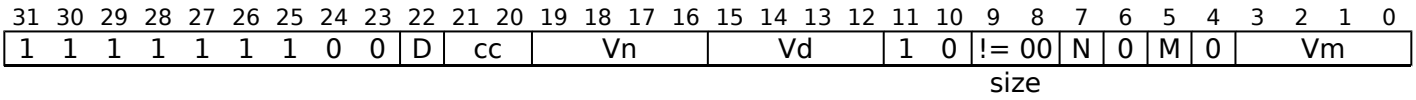
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VSELEQ, VSELGE, VSELGT, VSELVS

Floating-point conditional select allows the destination register to take the value in either one or the other source register according to the condition codes in the *APSR*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



VSELEQ,doubleprec (cc == 00 && size == 11)

```
VSELEQ.F64 <Dd>, <Dn>, <Dm> // (Cannot be conditional)
```

VSELEQ,halfprec (cc == 00 && size == 01)
(FEAT_FP16)

```
VSELEQ.F16 <Sd>, <Sn>, <Sm> // (Cannot be conditional)
```

VSELEQ,singleprec (cc == 00 && size == 10)

```
VSELEQ.F32 <Sd>, <Sn>, <Sm> // (Cannot be conditional)
```

VSELGE,doubleprec (cc == 10 && size == 11)

```
VSELGE.F64 <Dd>, <Dn>, <Dm> // (Cannot be conditional)
```

VSELGE,halfprec (cc == 10 && size == 01)
(FEAT_FP16)

```
VSELGE.F16 <Sd>, <Sn>, <Sm> // (Cannot be conditional)
```

VSELGE,singleprec (cc == 10 && size == 10)

```
VSELGE.F32 <Sd>, <Sn>, <Sm> // (Cannot be conditional)
```

VSELGT,doubleprec (cc == 11 && size == 11)

```
VSELGT.F64 <Dd>, <Dn>, <Dm> // (Cannot be conditional)
```

VSELGT,halfprec (cc == 11 && size == 01)
(FEAT_FP16)

```
VSELGT.F16 <Sd>, <Sn>, <Sm> // (Cannot be conditional)
```

VSELGT,singleprec (cc == 11 && size == 10)

```
VSELGT.F32 <Sd>, <Sn>, <Sm> // (Cannot be conditional)
```

VSELVS,doubleprec (cc == 01 && size == 11)

```
VSELVS.F64 <Dd>, <Dn>, <Dm> // (Cannot be conditional)
```

VSELVS,halfprec (cc == 01 && size == 01)
(FEAT_FP16)

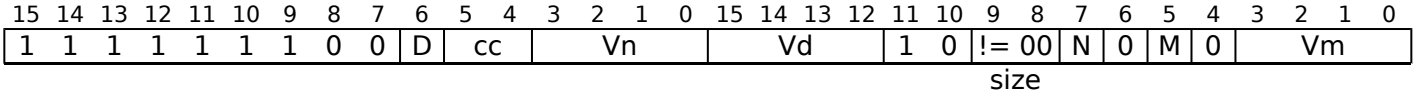
```
VSELVS.F16 <Sd>, <Sn>, <Sm> // (Cannot be conditional)
```

VSELVS,singleprec (cc == 01 && size == 10)

```
VSELVS.F32 <Sd>, <Sn>, <Sm> // (Cannot be conditional)
```

```
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);
cond = cc:(cc<1> EOR cc<0>):'0';
```

T1


15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 1 1 1 1 0 0 D cc Vn Vd 1 0 != 00 N 0 M 0 Vm
size

VSELEQ,doubleprec (cc == 00 && size == 11)

```
VSELEQ.F64 <Dd>, <Dn>, <Dm> // (Not permitted in IT block)
```

VSELEQ,halfprec (cc == 00 && size == 01)
(FEAT_FP16)

```
VSELEQ.F16 <Sd>, <Sn>, <Sm> // (Not permitted in IT block)
```

VSELEQ,singleprec (cc == 00 && size == 10)

```
VSELEQ.F32 <Sd>, <Sn>, <Sm> // (Not permitted in IT block)
```

VSELGE,doubleprec (cc == 10 && size == 11)

```
VSELGE.F64 <Dd>, <Dn>, <Dm> // (Not permitted in IT block)
```

VSELGE,halfprec (cc == 10 && size == 01)
(FEAT_FP16)

```
VSELGE.F16 <Sd>, <Sn>, <Sm> // (Not permitted in IT block)
```

VSELGE,singleprec (cc == 10 && size == 10)

```
VSELGE.F32 <Sd>, <Sn>, <Sm> // (Not permitted in IT block)
```

VSELGT,doubleprec (cc == 11 && size == 11)

```
VSELGT.F64 <Dd>, <Dn>, <Dm> // (Not permitted in IT block)
```

VSELGT,halfprec (cc == 11 && size == 01)
(FEAT_FP16)

```
VSELGT.F16 <Sd>, <Sn>, <Sm> // (Not permitted in IT block)
```

VSELGT,singleprec (cc == 11 && size == 10)

```
VSELGT.F32 <Sd>, <Sn>, <Sm> // (Not permitted in IT block)
```

VSELVS,doubleprec (cc == 01 && size == 11)

```
VSELVS.F64 <Dd>, <Dn>, <Dm> // (Not permitted in IT block)
```

VSELVS,halfprec (cc == 01 && size == 01)
(FEAT_FP16)

```
VSELVS.F16 <Sd>, <Sn>, <Sm> // (Not permitted in IT block)
```

VSELVS,singleprec (cc == 01 && size == 10)

```
VSELVS.F32 <Sd>, <Sn>, <Sm> // (Not permitted in IT block)
```

```
if InITBlock() then UNPREDICTABLE;
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);
cond = cc:(cc<1> EOR cc<0>):'0';
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
<Sn>	Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.
<Sm>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.

Operation

```
EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
case esize of
    when 16
        S[d] = Zeros(16) : (if ConditionHolds(cond) then S[n] else S[m])<15:0>;
    when 32
        S[d] = if ConditionHolds(cond) then S[n] else S[m];
    when 64
        D[d] = if ConditionHolds(cond) then D[n] else D[m];
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VSHL (immediate)

Vector Shift Left (immediate) takes each element in a vector of integers, left shifts them by an immediate value, and places the results in the destination vector.

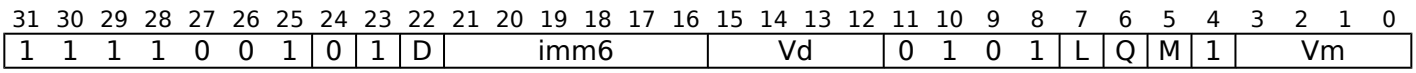
Bits shifted out of the left of each element are lost.

The elements must all be the same size, and can be 8-bit, 16-bit, 32-bit, or 64-bit integers. There is no distinction between signed and unsigned integers.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



64-bit SIMD vector (!(imm6 == 000xxx && L == 0) && Q == 0)

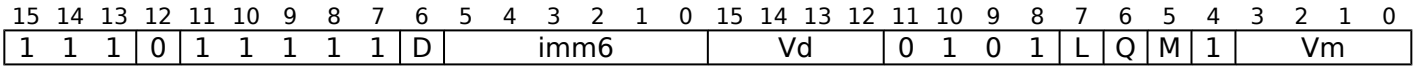
```
VSHL{<c>}{<q>}.I<size> {<Dd>,} <Dm>, #<imm>
```

128-bit SIMD vector (!(imm6 == 000xxx && L == 0) && Q == 1)

```
VSHL{<c>}{<q>}.I<size> {<Qd>,} <Qm>, #<imm>
```

```
if L:imm6 == '0000xxx' then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
case L:imm6 of
    when '0001xxx'  esize = 8;  elements = 8;  shift_amount = UInt(imm6) - 8;
    when '001xxxx'  esize = 16;  elements = 4;  shift_amount = UInt(imm6) - 16;
    when '01xxxxx'  esize = 32;  elements = 2;  shift_amount = UInt(imm6) - 32;
    when '1xxxxxx'  esize = 64;  elements = 1;  shift_amount = UInt(imm6);
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1



64-bit SIMD vector (!(imm6 == 000xxx && L == 0) && Q == 0)

```
VSHL{<c>}{<q>}.I<size> {<Dd>,} <Dm>, #<imm>
```

128-bit SIMD vector (!(imm6 == 000xxx && L == 0) && Q == 1)

```
VSHL{<c>}{<q>}.I<size> {<Qd>,} <Qm>, #<imm>
```

```
if L:imm6 == '0000xxx' then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
case L:imm6 of
    when '0001xxx'  esize = 8;  elements = 8;  shift_amount = UInt(imm6) - 8;
    when '001xxxx'  esize = 16;  elements = 4;  shift_amount = UInt(imm6) - 16;
    when '01xxxxx'  esize = 32;  elements = 2;  shift_amount = UInt(imm6) - 32;
    when '1xxxxxx'  esize = 64;  elements = 1;  shift_amount = UInt(imm6);
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Related encodings: See *Advanced SIMD one register and modified immediate* for the T32 instruction set, or *Advanced SIMD one register and modified immediate* for the A32 instruction set.

Assembler Symbols

<c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<size> Is the data size for the elements of the vectors, encoded in “L:imm6<5:3>”:

L	imm6<5:3>	<size>
0	001	8
0	01x	16
0	1xx	32
1	xxx	64

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

<imm> Is an immediate value, in the range 0 to <size>-1, encoded in the "imm6" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            Elem[D[d+r],e,esize] = LSL(Elem[D[m+r],e,esize], shift_amount);
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VSHL (register)

Vector Shift Left (register) takes each element in a vector, shifts them by a value from the least significant byte of the corresponding element of a second vector, and places the results in the destination vector. If the shift value is positive, the operation is a left shift. If the shift value is negative, it is a truncating right shift.

For a rounding shift, see *VRSHL*.

The first operand and result elements are the same data type, and can be any one of:

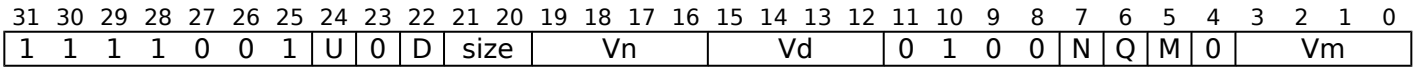
- 8-bit, 16-bit, 32-bit, or 64-bit signed integers.
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

The second operand is always a signed integer of the same size.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



64-bit SIMD vector (Q == 0)

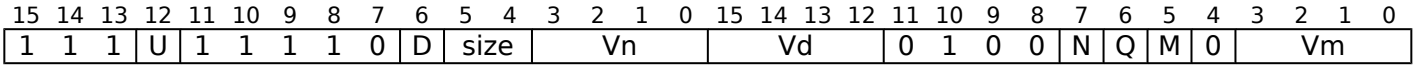
```
VSHL{<c>}{<q>}.<dt> {<Dd>,} <Dm>, <Dn>
```

128-bit SIMD vector (Q == 1)

```
VSHL{<c>}{<q>}.<dt> {<Qd>,} <Qm>, <Qn>
```

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1' || Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;
```

T1



64-bit SIMD vector (Q == 0)

```
VSHL{<c>}{<q>}.<dt> {<Dd>,} <Dm>, <Dn>
```

128-bit SIMD vector (Q == 1)

```
VSHL{<c>}{<q>}.<dt> {<Qd>,} <Qm>, <Qn>
```

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1' || Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>	For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional. For encoding T1: see *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the vectors, encoded in “U:size”:

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
0	11	S64
1	00	U8
1	01	U16
1	10	U32
1	11	U64

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            shift = SInt(Elem[D[n+r],e,esize]<7:0>);
            result = Int(Elem[D[m+r],e,esize], unsigned) << shift;
            Elem[D[d+r],e,esize] = result<esize-1:0>;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VSHLL

Vector Shift Left Long takes each element in a doubleword vector, left shifts them by an immediate value, and places the results in a quadword vector.

The operand elements can be:

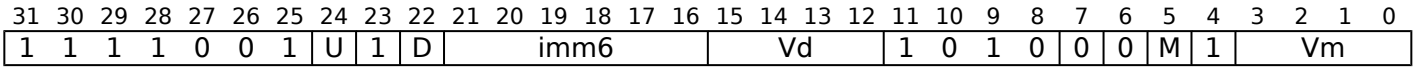
- 8-bit, 16-bit, or 32-bit signed integers.
- 8-bit, 16-bit, or 32-bit unsigned integers.
- 8-bit, 16-bit, or 32-bit untyped integers, maximum shift only.

The result elements are twice the length of the operand elements.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2) .

A1

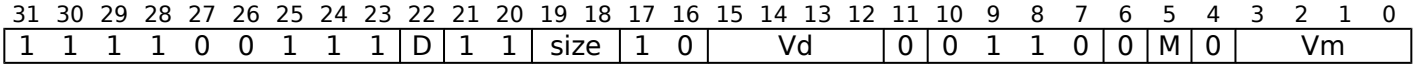


A1 (imm6 != 000xxx)

```
VSHLL{<c>}{<q>}.<type><size> <Qd>, <Dm>, #<imm>
```

```
if imm6 == '000xxx' then SEE "Related encodings";
if Vd<0> == '1' then UNDEFINED;
case imm6 of
    when '001xxx'  esize = 8;  elements = 8;  shift_amount = UInt(imm6) - 8;
    when '01xxxx'  esize = 16;  elements = 4;  shift_amount = UInt(imm6) - 16;
    when '1xxxxx'  esize = 32;  elements = 2;  shift_amount = UInt(imm6) - 32;
if shift_amount == 0 then SEE "VMOVL";
unsigned = (U == '1');  d = UInt(D:Vd);  m = UInt(M:Vm);
```

A2

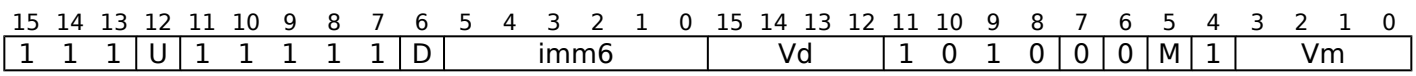


A2

```
VSHLL{<c>}{<q>}.<type><size> <Qd>, <Dm>, #<imm>
```

```
if size == '11' || Vd<0> == '1' then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;  shift_amount = esize;
unsigned = FALSE;  // Or TRUE without change of functionality
d = UInt(D:Vd);  m = UInt(M:Vm);
```

T1



T1 (imm6 != 000xxx)

```
VSHLL{<c>}{<q>}.<type><size> <Qd>, <Dm>, #<imm>
```

```
if imm6 == '000xxx' then SEE "Related encodings";
if Vd<0> == '1' then UNDEFINED;
case imm6 of
    when '001xxx'  esize = 8;  elements = 8;  shift_amount = UInt(imm6) - 8;
    when '01xxxx'  esize = 16;  elements = 4;  shift_amount = UInt(imm6) - 16;
    when '1xxxxx'  esize = 32;  elements = 2;  shift_amount = UInt(imm6) - 32;
if shift_amount == 0 then SEE "VMOVL";
unsigned = (U == '1');  d = UInt(D:Vd);  m = UInt(M:Vm);
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size		1	0	Vd				0	0	1	1	0	0	M	0	Vm			

T2

```
VSHLL{<c>}{<q>}.<type><size> <Qd>, <Dm>, #<imm>
```

```
if size == '11' || Vd<0> == '1' then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;  shift_amount = esize;
unsigned = FALSE;  // Or TRUE without change of functionality
d = UInt(D:Vd);  m = UInt(M:Vm);
```

Related encodings: See *Advanced SIMD one register and modified immediate* for the T32 instruction set, or *Advanced SIMD one register and modified immediate* for the A32 instruction set.

Assembler Symbols

<c> For encoding A1 and A2: see *Standard assembler syntax fields*. This encoding must be unconditional.

For encoding T1 and T2: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<type> The data type for the elements of the operand. It must be one of:

S
Signed. In encoding T1/A1, encoded as U = 0.

U
Unsigned. In encoding T1/A1, encoded as U = 1.

I
Untyped integer, Available only in encoding T2/A2.

<size> The data size for the elements of the operand. The following table shows the permitted values and their encodings:

<size>	Encoding T1/A1	Encoding T2/A2
8	Encoded as imm6<5:3> = 0b001	Encoded as size = 0b00
16	Encoded as imm6<5:4> = 0b01	Encoded as size = 0b01
32	Encoded as imm6<5> = 1	Encoded as size = 0b10

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

<imm> The immediate value. <imm> must lie in the range 1 to <size>, and:

- If <size> == <imm>, the encoding is T2/A2.
- Otherwise, the encoding is T1/A1, and:
  - If <size> == 8, <imm> is encoded in imm6<2:0>.
  - If <size> == 16, <imm> is encoded in imm6<3:0>.
  - If <size> == 32, <imm> is encoded in imm6<4:0>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for e = 0 to elements-1
        result = Int(Elem[Din[m],e,esize], unsigned) << shift_amount;
        Elem[Q[d>>1],e,2*esize] = result<2*esize-1:0>;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VSHR

Vector Shift Right takes each element in a vector, right shifts them by an immediate value, and places the truncated results in the destination vector. For rounded results, see *VRSHR*.

The operand and result elements must be the same size, and can be any one of:

- 8-bit, 16-bit, 32-bit, or 64-bit signed integers.
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	1	D	imm6						Vd				0	0	0	0	L	Q	M	1	Vm			

64-bit SIMD vector (!(imm6 == 000xxx && L == 0) && Q == 0)

```
VSHR{<c>}{<q>}.<type><size> {<Dd>,} <Dm>, #<imm>
```

128-bit SIMD vector (!(imm6 == 000xxx && L == 0) && Q == 1)

```
VSHR{<c>}{<q>}.<type><size> {<Qd>,} <Qm>, #<imm>
```

```
if (L:imm6) == '0000xxx' then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
case L:imm6 of
    when '0001xxx'  esize = 8;  elements = 8;  shift_amount = 16 - UInt(imm6);
    when '001xxxx'  esize = 16;  elements = 4;  shift_amount = 32 - UInt(imm6);
    when '01xxxxx'  esize = 32;  elements = 2;  shift_amount = 64 - UInt(imm6);
    when '1xxxxxx'  esize = 64;  elements = 1;  shift_amount = 64 - UInt(imm6);
unsigned = (U == '1');  d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	U	1	1	1	1	1	D	imm6						Vd				0	0	0	0	L	Q	M	1	Vm			

64-bit SIMD vector (!(imm6 == 000xxx && L == 0) && Q == 0)

```
VSHR{<c>}{<q>}.<type><size> {<Dd>,} <Dm>, #<imm>
```

128-bit SIMD vector (!(imm6 == 000xxx && L == 0) && Q == 1)

```
VSHR{<c>}{<q>}.<type><size> {<Qd>,} <Qm>, #<imm>
```

```
if (L:imm6) == '0000xxx' then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
case L:imm6 of
    when '0001xxx'  esize = 8;  elements = 8;  shift_amount = 16 - UInt(imm6);
    when '001xxxx'  esize = 16;  elements = 4;  shift_amount = 32 - UInt(imm6);
    when '01xxxxx'  esize = 32;  elements = 2;  shift_amount = 64 - UInt(imm6);
    when '1xxxxxx'  esize = 64;  elements = 1;  shift_amount = 64 - UInt(imm6);
unsigned = (U == '1');  d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Related encodings: See *Advanced SIMD one register and modified immediate* for the T32 instruction set, or *Advanced SIMD one register and modified immediate* for the A32 instruction set.

Assembler Symbols

<c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.
For encoding T1: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<type> Is the data type for the elements of the vectors, encoded in “U”:

U	<type>
0	S
1	U

<size> Is the data size for the elements of the vectors, encoded in “L:imm6<5:3>”:

L	imm6<5:3>	<size>
0	001	8
0	01x	16
0	1xx	32
1	xxx	64

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

<imm> Is an immediate value, in the range 1 to <size>, encoded in the "imm6" field as <size> - <imm>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            result = Int(Elem[D[m+r],e,esize], unsigned) >> shift_amount;
            Elem[D[d+r],e,esize] = result<esize-1:0>;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VSHR (zero)

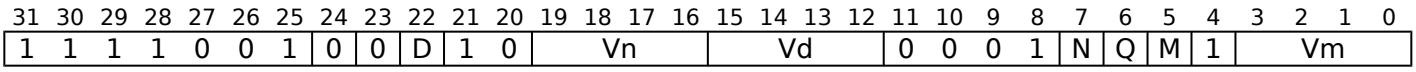
Vector Shift Right copies the contents of one SIMD register to another

This is a pseudo-instruction of VORR (register). This means:

- The encodings in this description are named to match the encodings of VORR (register).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of VORR (register) gives the operational pseudocode for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



64-bit SIMD vector (Q == 0)

```
VSHR{<c>}{<q>}.<dt> <Dd>, <Dm>, #0
```

is equivalent to

```
VORR{<c>}{<q>}{.<dt>} <Dd>, <Dm>, <Dm>
```

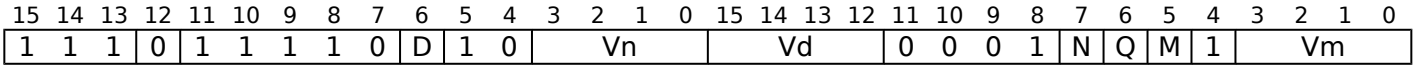
128-bit SIMD vector (Q == 1)

```
VSHR{<c>}{<q>}.<dt> <Qd>, <Qm>, #0
```

is equivalent to

```
VORR{<c>}{<q>}{.<dt>} <Qd>, <Qm>, <Qm>
```

T1



64-bit SIMD vector (Q == 0)

```
VSHR{<c>}{<q>}.<dt> <Dd>, <Dm>, #0
```

is equivalent to

```
VORR{<c>}{<q>}{.<dt>} <Dd>, <Dm>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VSHR{<c>}{<q>}.<dt> <Qd>, <Qm>, #0
```

is equivalent to

```
VORR{<c>}{<q>}{.<dt>} <Qd>, <Qm>, <Qm>
```

Assembler Symbols

<c>	For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional. For encoding T1: see *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.

<dt>	Is the data type for the elements of the vectors, and must be one of: S8, S16, S32, S64, U8, U16, U32 or U64.
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qm>	Is the 128-bit name of the SIMD&FP source register, encoded in the "N:Vn" and "M:Vm" field as <Qm>*2.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dm>	Is the 64-bit name of the SIMD&FP source register, encoded in the "N:Vn" and "M:Vm" field.

Operation

The description of VORR (register) gives the operational pseudocode for this instruction.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VSHRN

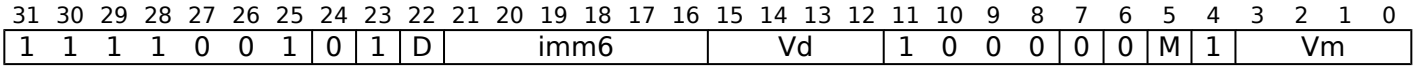
Vector Shift Right Narrow takes each element in a vector, right shifts them by an immediate value, and places the truncated results in the destination vector. For rounded results, see *VRSHRN*.

The operand elements can be 16-bit, 32-bit, or 64-bit integers. There is no distinction between signed and unsigned integers. The destination elements are half the size of the source elements.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

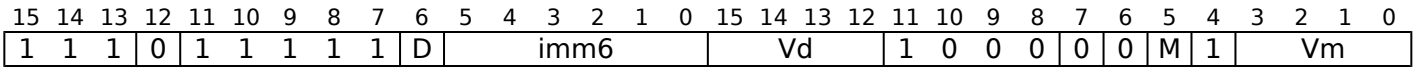


A1 (imm6 != 000xxx)

```
VSHRN{<c>}{<q>}.I<size> <Dd>, <Qm>, #<imm>
```

```
if imm6 == '000xxx' then SEE "Related encodings";
if Vm<0> == '1' then UNDEFINED;
case imm6 of
    when '001xxx'  esize = 8;  elements = 8;  shift_amount = 16 - UInt(imm6);
    when '01xxxx'  esize = 16;  elements = 4;  shift_amount = 32 - UInt(imm6);
    when '1xxxxx'  esize = 32;  elements = 2;  shift_amount = 64 - UInt(imm6);
d = UInt(D:Vd);  m = UInt(M:Vm);
```

T1



T1 (imm6 != 000xxx)

```
VSHRN{<c>}{<q>}.I<size> <Dd>, <Qm>, #<imm>
```

```
if imm6 == '000xxx' then SEE "Related encodings";
if Vm<0> == '1' then UNDEFINED;
case imm6 of
    when '001xxx'  esize = 8;  elements = 8;  shift_amount = 16 - UInt(imm6);
    when '01xxxx'  esize = 16;  elements = 4;  shift_amount = 32 - UInt(imm6);
    when '1xxxxx'  esize = 32;  elements = 2;  shift_amount = 64 - UInt(imm6);
d = UInt(D:Vd);  m = UInt(M:Vm);
```

Related encodings: See *Advanced SIMD one register and modified immediate* for the T32 instruction set, or *Advanced SIMD one register and modified immediate* for the A32 instruction set.

Assembler Symbols

<c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<size> Is the data size for the elements of the vectors, encoded in “imm6<5:3>”:

imm6<5:3>	<size>
001	16
01x	32
1xx	64

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<imm> Is an immediate value, in the range 1 to <size>/2, encoded in the "imm6" field as <size>/2 - <imm>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for e = 0 to elements-1
        result = LSR(Elem[Qin[m>>1],e,2*esize], shift_amount);
        Elem[D[d],e,esize] = result<esize-1:0>;
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VSHRN (zero)

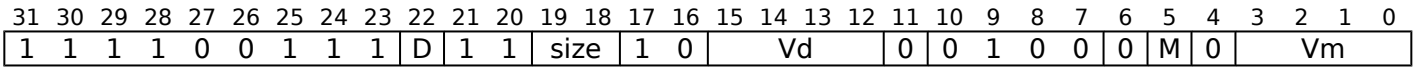
Vector Shift Right Narrow takes each element in a vector, right shifts them by an immediate value, and places the truncated results in the destination vector

This is a pseudo-instruction of VMOVN. This means:

- The encodings in this description are named to match the encodings of VMOVN.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of VMOVN gives the operational pseudocode for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



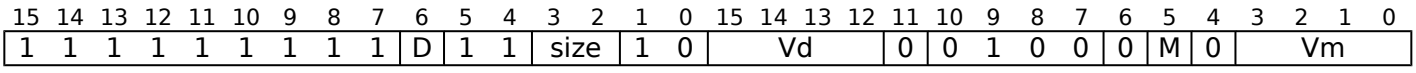
A1

```
VSHRN{<c>}{<q>}.<dt> <Dd>, <Qm>, #0
```

is equivalent to

```
VMOVN{<c>}{<q>}.<dt> <Dd>, <Qm>
```

T1



T1

```
VSHRN{<c>}{<q>}.<dt> <Dd>, <Qm>, #0
```

is equivalent to

```
VMOVN{<c>}{<q>}.<dt> <Dd>, <Qm>
```

Assembler Symbols

<c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.
For encoding T1: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the operand, encoded in “size”:

size	<dt>
00	I16
01	I32
10	I64
11	RESERVED

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

The description of VMOVN gives the operational pseudocode for this instruction.

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VSLI

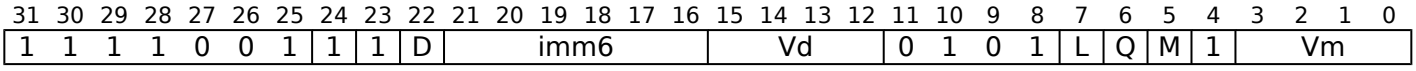
Vector Shift Left and Insert takes each element in the operand vector, left shifts them by an immediate value, and inserts the results in the destination vector. Bits shifted out of the left of each element are lost.

The elements must all be the same size, and can be 8-bit, 16-bit, 32-bit, or 64-bit. There is no distinction between data types.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



64-bit SIMD vector (!(imm6 == 000xxx && L == 0) && Q == 0)

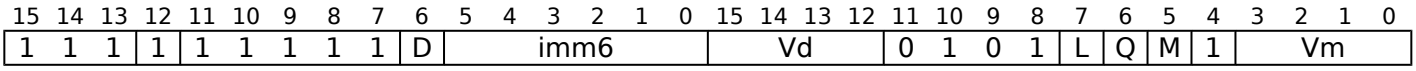
```
VSLI{<c>}{<q>}.<size> {<Dd>,} <Dm>, #<imm>
```

128-bit SIMD vector (!(imm6 == 000xxx && L == 0) && Q == 1)

```
VSLI{<c>}{<q>}.<size> {<Qd>,} <Qm>, #<imm>
```

```
if (L:imm6) == '0000xxx' then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
case L:imm6 of
    when '0001xxx'  esize = 8;  elements = 8;  shift_amount = UInt(imm6) - 8;
    when '001xxxx'  esize = 16;  elements = 4;  shift_amount = UInt(imm6) - 16;
    when '01xxxxx'  esize = 32;  elements = 2;  shift_amount = UInt(imm6) - 32;
    when '1xxxxxx'  esize = 64;  elements = 1;  shift_amount = UInt(imm6);
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1



64-bit SIMD vector (!(imm6 == 000xxx && L == 0) && Q == 0)

```
VSLI{<c>}{<q>}.<size> {<Dd>,} <Dm>, #<imm>
```

128-bit SIMD vector (!(imm6 == 000xxx && L == 0) && Q == 1)

```
VSLI{<c>}{<q>}.<size> {<Qd>,} <Qm>, #<imm>
```

```
if (L:imm6) == '0000xxx' then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
case L:imm6 of
    when '0001xxx'  esize = 8;  elements = 8;  shift_amount = UInt(imm6) - 8;
    when '001xxxx'  esize = 16;  elements = 4;  shift_amount = UInt(imm6) - 16;
    when '01xxxxx'  esize = 32;  elements = 2;  shift_amount = UInt(imm6) - 32;
    when '1xxxxxx'  esize = 64;  elements = 1;  shift_amount = UInt(imm6);
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Related encodings: See *Advanced SIMD one register and modified immediate* for the T32 instruction set, or *Advanced SIMD one register and modified immediate* for the A32 instruction set.

Assembler Symbols

<c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<size> Is the data size for the elements of the vectors, encoded in “L:imm6<5:3>”:

L	imm6<5:3>	<size>
0	001	8
0	01x	16
0	1xx	32
1	xxx	64

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

<imm> Is an immediate value, in the range 0 to <size>-1, encoded in the "imm6" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    mask = LSL(Ones(esize), shift_amount);
    for r = 0 to regs-1
        for e = 0 to elements-1
            shifted_op = LSL(Elem[D[m+r],e,esize], shift_amount);
            Elem[D[d+r],e,esize] = (Elem[D[d+r],e,esize] AND NOT(mask)) OR shifted_op;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VSMMLA

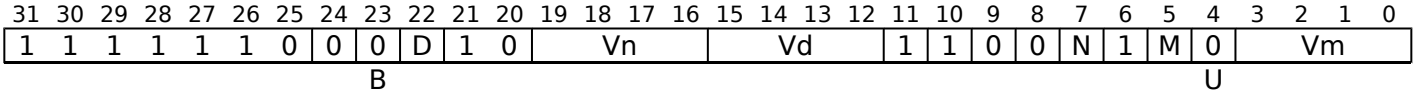
The widening integer matrix multiply-accumulate instruction multiplies the 2x8 matrix of signed 8-bit integer values held in the first source vector by the 8x2 matrix of signed 8-bit integer values in the second source vector. The resulting 2x2 32-bit integer matrix product is destructively added to the 32-bit integer matrix accumulator held in the destination vector. This is equivalent to performing an 8-way dot product per destination element.

From Armv8.2, this is an OPTIONAL instruction. *ID_ISAR6*.I8MM indicates whether this instruction is supported in the T32 and A32 instruction sets.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

(FEAT_AA32I8MM)



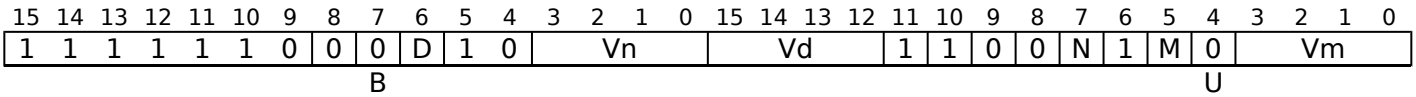
A1

```
VSMMLA{<q>}.S8 <Qd>, <Qn>, <Qm>
```

```
if !HaveAArch32Int8MatMulExt() then UNDEFINED;
case B:U of
    when '00' op1_unsigned = FALSE; op2_unsigned = FALSE;
    when '01' op1_unsigned = TRUE;  op2_unsigned = TRUE;
    when '10' op1_unsigned = TRUE;  op2_unsigned = FALSE;
    when '11' UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(M:Vm);
```

T1

(FEAT_AA32I8MM)



T1

```
VSMMLA{<q>}.S8 <Qd>, <Qn>, <Qm>
```

```
if InITBlock() then UNPREDICTABLE;
if !HaveAArch32Int8MatMulExt() then UNDEFINED;
case B:U of
    when '00' op1_unsigned = FALSE; op2_unsigned = FALSE;
    when '01' op1_unsigned = TRUE;  op2_unsigned = TRUE;
    when '10' op1_unsigned = TRUE;  op2_unsigned = FALSE;
    when '11' UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(M:Vm);
```

Assembler Symbols

<q>	See *Standard assembler syntax fields*.
<Qd>	Is the 128-bit name of the SIMD&FP third source and destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
CheckAdvSIMDEnabled();
bits(128) operand1 = Q[n>>1];
bits(128) operand2 = Q[m>>1];
bits(128) addend   = Q[d>>1];

Q[d>>1] = MatMulAdd(addend, operand1, operand2, op1_unsigned, op2_unsigned);
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

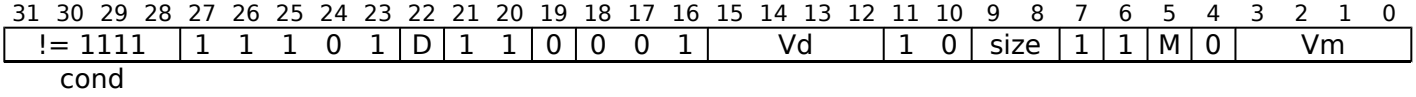
VSQRT

Square Root calculates the square root of the value in a floating-point register and writes the result to another floating-point register.

Depending on settings in the *CPACR*, *NSACR*, *HCPTR*, and *FPEXC* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



Half-precision scalar (size == 01)
(FEAT_FP16)

```
VSQRT{<c>}{<q>}.F16 <Sd>, <Sm>
```

Single-precision scalar (size == 10)

```
VSQRT{<c>}{<q>}.F32 <Sd>, <Sm>
```

Double-precision scalar (size == 11)

```
VSQRT{<c>}{<q>}.F64 <Dd>, <Dm>
```

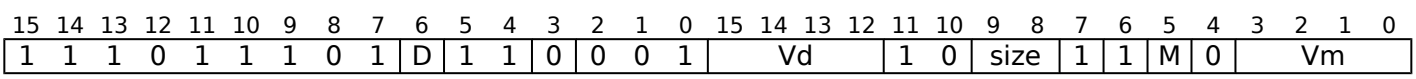
```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
case size of
    when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); 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



Half-precision scalar (size == 01)
(FEAT_FP16)

```
VSQRT{<c>}{<q>}.F16 <Sd>, <Sm>
```

Single-precision scalar (size == 10)

```
VSQRT{<c>}{<q>}.F32 <Sd>, <Sm>
```

Double-precision scalar (size == 11)

```
VSQRT{<c>}{<q>}.F64 <Dd>, <Dm>
```

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock()  then UNPREDICTABLE;
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
case size of
    when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
<Sm>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dm>	Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    case esize of
        when 16 S[d] = Zeros(16) : FPSqrt(S[m]<15:0>, FPSCR[]);
        when 32 S[d] = FPSqrt(S[m], FPSCR[]);
        when 64 D[d] = FPSqrt(D[m], FPSCR[]);
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VSRA

Vector Shift Right and Accumulate takes each element in a vector, right shifts them by an immediate value, and accumulates the truncated results into the destination vector. For rounded results, see *VRSRA*.

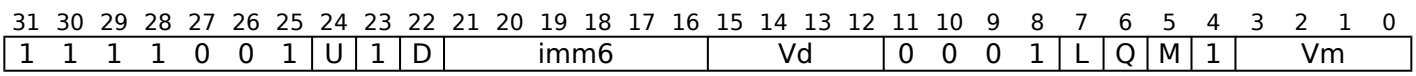
The operand and result elements must all be the same type, and can be any one of:

- 8-bit, 16-bit, 32-bit, or 64-bit signed integers.
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



64-bit SIMD vector (!(imm6 == 000xxx && L == 0) && Q == 0)

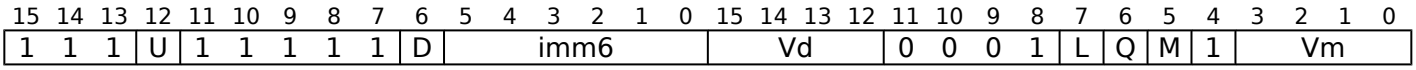
```
VSRA{<c>}{<q>}.<type><size> {<Dd>,} <Dm>, #<imm>
```

128-bit SIMD vector (!(imm6 == 000xxx && L == 0) && Q == 1)

```
VSRA{<c>}{<q>}.<type><size> {<Qd>,} <Qm>, #<imm>
```

```
if (L:imm6) == '0000xxx' then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
case L:imm6 of
    when '0001xxx'  esize = 8;  elements = 8;  shift_amount = 16 - UInt(imm6);
    when '001xxxx'  esize = 16;  elements = 4;  shift_amount = 32 - UInt(imm6);
    when '01xxxxx'  esize = 32;  elements = 2;  shift_amount = 64 - UInt(imm6);
    when '1xxxxxx'  esize = 64;  elements = 1;  shift_amount = 64 - UInt(imm6);
unsigned = (U == '1');  d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1



64-bit SIMD vector (!(imm6 == 000xxx && L == 0) && Q == 0)

```
VSRA{<c>}{<q>}.<type><size> {<Dd>,} <Dm>, #<imm>
```

128-bit SIMD vector (!(imm6 == 000xxx && L == 0) && Q == 1)

```
VSRA{<c>}{<q>}.<type><size> {<Qd>,} <Qm>, #<imm>
```

```
if (L:imm6) == '0000xxx' then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
case L:imm6 of
    when '0001xxx'  esize = 8;  elements = 8;  shift_amount = 16 - UInt(imm6);
    when '001xxxx'  esize = 16;  elements = 4;  shift_amount = 32 - UInt(imm6);
    when '01xxxxx'  esize = 32;  elements = 2;  shift_amount = 64 - UInt(imm6);
    when '1xxxxxx'  esize = 64;  elements = 1;  shift_amount = 64 - UInt(imm6);
unsigned = (U == '1');  d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Related encodings: See *Advanced SIMD one register and modified immediate* for the T32 instruction set, or *Advanced SIMD one register and modified immediate* for the A32 instruction set.

Assembler Symbols

<c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.
For encoding T1: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<type> Is the data type for the elements of the vectors, encoded in “U”:

U	<type>
0	S
1	U

<size> Is the data size for the elements of the vectors, encoded in “L:imm6<5:3>”:

L	imm6<5:3>	<size>
0	001	8
0	01x	16
0	1xx	32
1	xxx	64

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

<imm> Is an immediate value, in the range 1 to <size>, encoded in the "imm6" field as <size> - <imm>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            result = Int(Elem[D[m+r],e,esize], unsigned) >> shift_amount;
            Elem[D[d+r],e,esize] = Elem[D[d+r],e,esize] + result;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VSRI

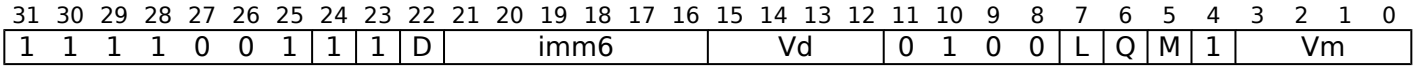
Vector Shift Right and Insert takes each element in the operand vector, right shifts them by an immediate value, and inserts the results in the destination vector. Bits shifted out of the right of each element are lost.

The elements must all be the same size, and can be 8-bit, 16-bit, 32-bit, or 64-bit. There is no distinction between data types.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



64-bit SIMD vector (!(imm6 == 000xxx && L == 0) && Q == 0)

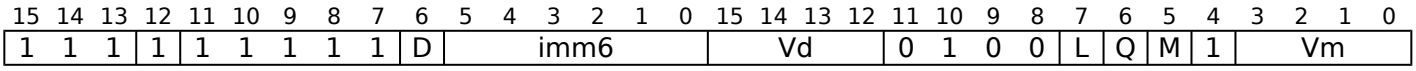
```
VSRI{<c>}{<q>}.<size> {<Dd>,} <Dm>, #<imm>
```

128-bit SIMD vector (!(imm6 == 000xxx && L == 0) && Q == 1)

```
VSRI{<c>}{<q>}.<size> {<Qd>,} <Qm>, #<imm>
```

```
if (L:imm6) == '0000xxx' then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
case L:imm6 of
    when '0001xxx'  esize = 8;  elements = 8;  shift_amount = 16 - UInt(imm6);
    when '001xxxx'  esize = 16;  elements = 4;  shift_amount = 32 - UInt(imm6);
    when '01xxxxx'  esize = 32;  elements = 2;  shift_amount = 64 - UInt(imm6);
    when '1xxxxxx'  esize = 64;  elements = 1;  shift_amount = 64 - UInt(imm6);
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1



64-bit SIMD vector (!(imm6 == 000xxx && L == 0) && Q == 0)

```
VSRI{<c>}{<q>}.<size> {<Dd>,} <Dm>, #<imm>
```

128-bit SIMD vector (!(imm6 == 000xxx && L == 0) && Q == 1)

```
VSRI{<c>}{<q>}.<size> {<Qd>,} <Qm>, #<imm>
```

```
if (L:imm6) == '0000xxx' then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
case L:imm6 of
    when '0001xxx'  esize = 8;  elements = 8;  shift_amount = 16 - UInt(imm6);
    when '001xxxx'  esize = 16;  elements = 4;  shift_amount = 32 - UInt(imm6);
    when '01xxxxx'  esize = 32;  elements = 2;  shift_amount = 64 - UInt(imm6);
    when '1xxxxxx'  esize = 64;  elements = 1;  shift_amount = 64 - UInt(imm6);
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Related encodings: See *Advanced SIMD one register and modified immediate* for the T32 instruction set, or *Advanced SIMD one register and modified immediate* for the A32 instruction set.

Assembler Symbols

<c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<size> Is the data size for the elements of the vectors, encoded in “L:imm6<5:3>”:

L	imm6<5:3>	<size>
0	001	8
0	01x	16
0	1xx	32
1	xxx	64

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

<imm> Is an immediate value, in the range 1 to <size>, encoded in the "imm6" field as <size> - <imm>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    mask = LSR(Ones(esize), shift_amount);
    for r = 0 to regs-1
        for e = 0 to elements-1
            shifted_op = LSR(Elem[D[m+r],e,esize], shift_amount);
            Elem[D[d+r],e,esize] = (Elem[D[d+r],e,esize] AND NOT(mask)) OR shifted_op;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

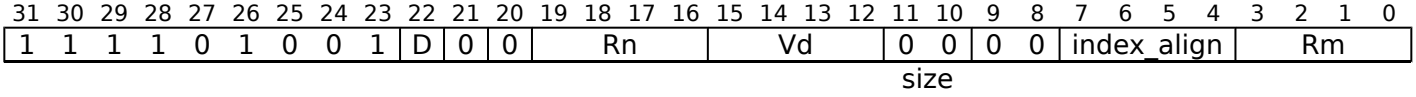
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VST1 (single element from one lane)

Store single element from one lane of one register stores one element to memory from one element of a register. For details of the addressing mode see *Advanced SIMD addressing mode*.
Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1 , A2 and A3) and T32 (T1 , T2 and T3) .

A1



Offset (Rm == 1111)

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

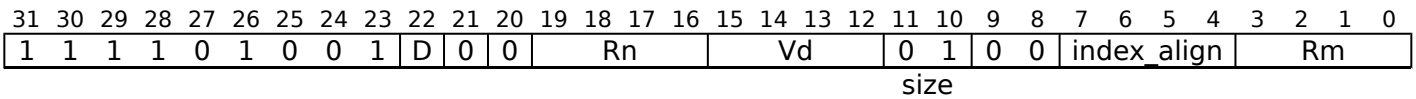
```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

```
if size == '11' then UNDEFINED;
if index_align<0> != '0' then UNDEFINED;
ebytes = 1;  index = UInt(index_align<3:1>);  alignment = 1;
d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 then UNPREDICTABLE;
```

A2



Offset (Rm == 1111)

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

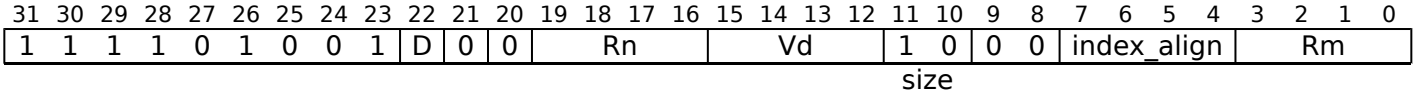
```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

```
if size == '11' then UNDEFINED;
if index_align<1> != '0' then UNDEFINED;
ebytes = 2;  index = UInt(index_align<3:2>);
alignment = if index_align<0> == '0' then 1 else 2;
d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 then UNPREDICTABLE;
```

A3



Offset (Rm == 1111)

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

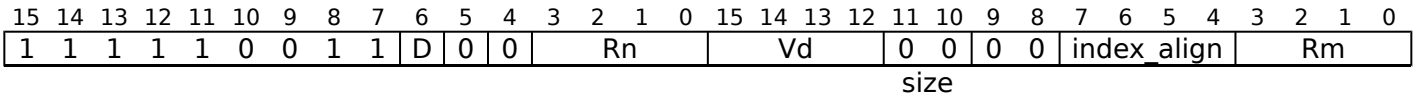
```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

```
if size == '11' then UNDEFINED;
if index_align<2> != '0' then UNDEFINED;
if index_align<1:0> != '00' && index_align<1:0> != '11' then UNDEFINED;
ebytes = 4;  index = UInt(index_align<3>);
alignment = if index_align<1:0> == '00' then 1 else 4;
d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 then UNPREDICTABLE;
```

T1



Offset (Rm == 1111)

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

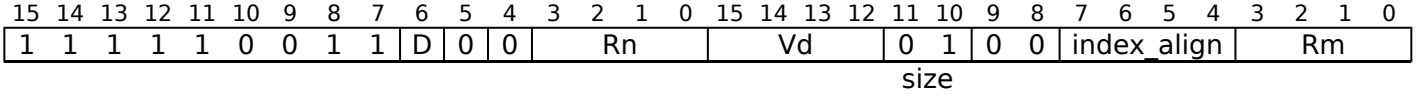
```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

```
if size == '11' then UNDEFINED;
if index_align<0> != '0' then UNDEFINED;
ebytes = 1;  index = UInt(index_align<3:1>);  alignment = 1;
d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 then UNPREDICTABLE;
```

T2



Offset (Rm == 1111)

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

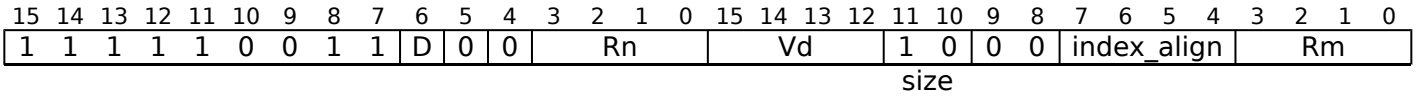
```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

```
if size == '11' then UNDEFINED;
if index_align<1> != '0' then UNDEFINED;
ebytes = 2;  index = UInt(index_align<3:2>);
alignment = if index_align<0> == '0' then 1 else 2;
d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 then UNPREDICTABLE;
```

T3



Offset (Rm == 1111)

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

```
if size == '11' then UNDEFINED;
if index_align<2> != '0' then UNDEFINED;
if index_align<1:0> != '00' && index_align<1:0> != '11' then UNDEFINED;
ebytes = 4;  index = UInt(index_align<3>);
alignment = if index_align<1:0> == '00' then 1 else 4;
d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 then UNPREDICTABLE;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	For encoding A1, A2 and A3: see *Standard assembler syntax fields*. This encoding must be unconditional. For encoding T1, T2 and T3: see *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<size>	Is the data size, encoded in "size":

size	<size>
00	8
01	16
10	32

<list> Is a list containing the single 64-bit name of the SIMD&FP register holding the element.
The list must be { <Dd>[<index>] }.
The register <Dd> is encoded in the "D:Vd" field.
The permitted values and encoding of <index> depend on <size>:

<size> == 8
<index> is in the range 0 to 7, encoded in the "index_align<3:1>" field.

<size> == 16
<index> is in the range 0 to 3, encoded in the "index_align<3:2>" field.

<size> == 32
<index> is 0 or 1, encoded in the "index_align<3>" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

<align> When <size> == 8, <align> must be omitted, otherwise it is the optional alignment.
Whenever <align> is omitted, the standard alignment is used, see *Unaligned data access*, and the encoding depends on <size>:

<size> == 8
Encoded in the "index_align<0>" field as 0.

<size> == 16
Encoded in the "index_align<1:0>" field as 0b00.

<size> == 32
Encoded in the "index_align<2:0>" field as 0b000.

Whenever <align> is present, the permitted values and encoding depend on <size>:

<size> == 16
<align> is 16, meaning 16-bit alignment, encoded in the "index_align<1:0>" field as 0b01.

<size> == 32
<align> is 32, meaning 32-bit alignment, encoded in the "index_align<2:0>" field as 0b011.

: is the preferred separator before the <align> value, but the alignment can be specified as @<align>, see *Advanced SIMD addressing mode*.

<Rm> Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see *Advanced SIMD addressing mode*.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    address = R[n];  iswrite = TRUE;
    - = AArch32.CheckAlignment(address, alignment, AccType_VEC, iswrite);
    MemU[address,ebytes] = Elem[D[d],index];
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + ebytes;
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

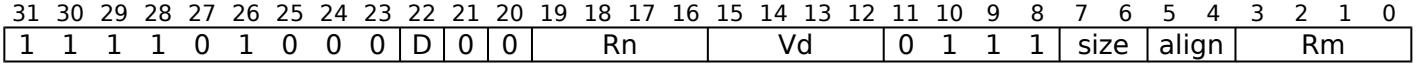
VST1 (multiple single elements)

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



Offset (Rm == 1111)

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

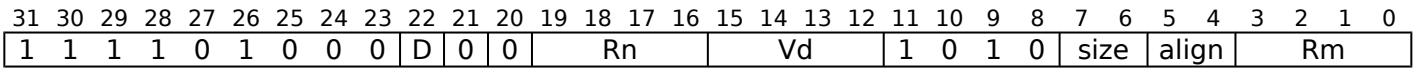
```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

```
regs = 1;  if align<1> == '1' then UNDEFINED;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 then UNPREDICTABLE;
```

A2



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>
```

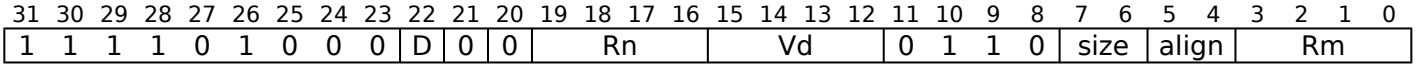
```
regs = 2;  if align == '11' then UNDEFINED;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UNPREDICTABLE;
```

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



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>
```

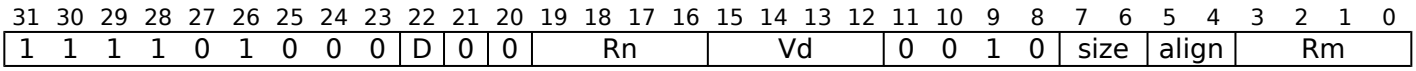
```
regs = 3;  if align<1> == '1' then UNDEFINED;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UNPREDICTABLE;
```

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



Offset (Rm == 1111)

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

```
regs = 4;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d+regs > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	0	D	0	0	Rn				Vd				0	1	1	1	size		align		Rm			

Offset (Rm == 1111)

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

```
regs = 1;  if align<1> == '1' then UNDEFINED;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 then UNPREDICTABLE;
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	0	D	0	0	Rn				Vd				1	0	1	0	size		align		Rm			

Offset (Rm == 1111)

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

```
regs = 2;  if align == '11' then UNDEFINED;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d+regs > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

T3

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	0	D	0	0	Rn				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>
```

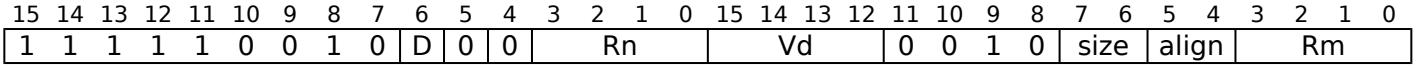
```
regs = 3;  if align<1> == '1' then UNDEFINED;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
d = UInt(D:Vd);  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UNPREDICTABLE;
```

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.

T4



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(AccType_NORMAL) then data<63:32> else data<31:0>;
                MemU[address+4,4] = if BigEndian(AccType_NORMAL) then data<31:0> else data<63:32>;
            address = address + ebytes;
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 8*regs;
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

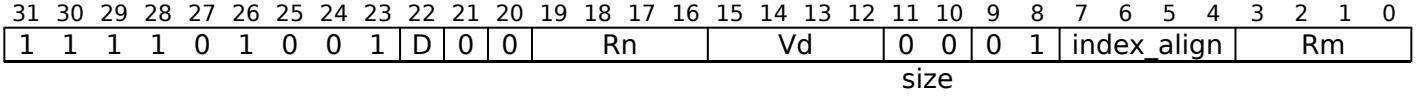
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VST2 (single 2-element structure from one lane)

Store single 2-element structure from one lane of two registers stores one 2-element structure to memory from corresponding elements of two registers. For details of the addressing mode see *Advanced SIMD addressing mode*. Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1 , A2 and A3) and T32 (T1 , T2 and T3) .

A1



Offset (Rm == 1111)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

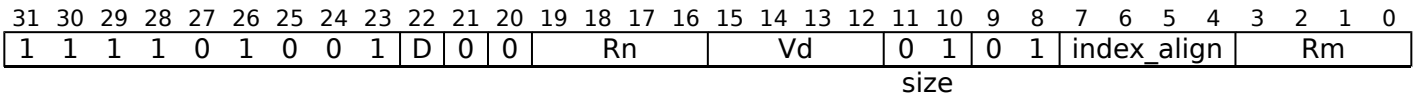
```
if size == '11' then UNDEFINED;
ebytes = 1;  index = UInt(index_align<3:1>);  inc = 1;
alignment = if index_align<0> == '0' then 1 else 2;
d = UInt(D:Vd);  d2 = d + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d2 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

A2



Offset (Rm == 1111)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

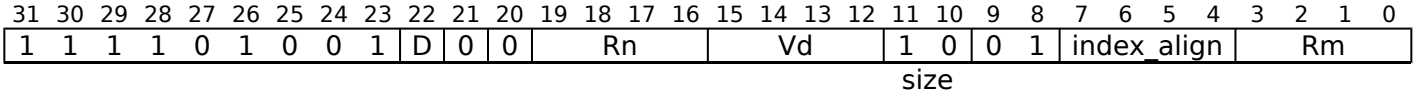
```
if size == '11' then UNDEFINED;
ebytes = 2;  index = UInt(index_align<3:2>);
inc = if index_align<1> == '0' then 1 else 2;
alignment = if index_align<0> == '0' then 1 else 4;
d = UInt(D:Vd);  d2 = d + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d2 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

A3



Offset (Rm == 1111)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

```
if size == '11' then UNDEFINED;
if index_align<1> != '0' then UNDEFINED;
ebytes = 4;  index = UInt(index_align<3>);
inc = if index_align<2> == '0' then 1 else 2;
alignment = if index_align<0> == '0' then 1 else 8;
d = UInt(D:Vd);  d2 = d + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UNPREDICTABLE;
```

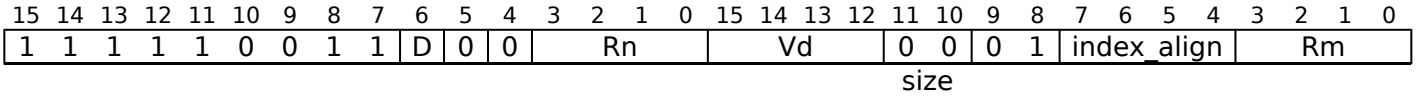
CONSTRAINED UNPREDICTABLE behavior

If `d2 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.

- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

T1



Offset (Rm == 1111)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

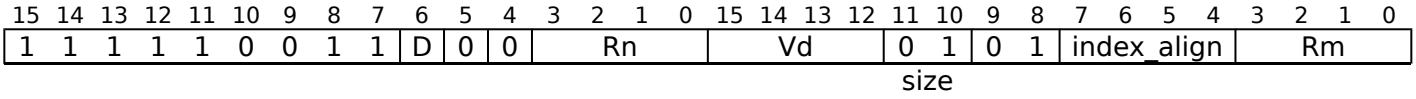
```
if size == '11' then UNDEFINED;
ebytes = 1;  index = UInt(index_align<3:1>);  inc = 1;
alignment = if index_align<0> == '0' then 1 else 2;
d = UInt(D:Vd);  d2 = d + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d2 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

T2



Offset (Rm == 1111)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

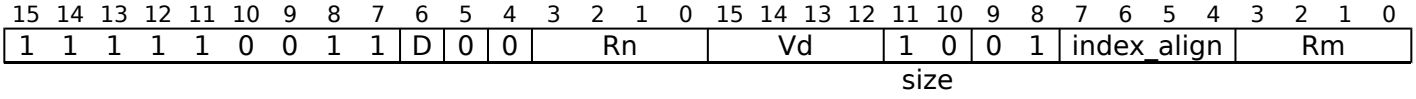
```
if size == '11' then UNDEFINED;
ebytes = 2;  index = UInt(index_align<3:2>);
inc = if index_align<1> == '0' then 1 else 2;
alignment = if index_align<0> == '0' then 1 else 4;
d = UInt(D:Vd);  d2 = d + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d2 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

T3



Offset (Rm == 1111)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

```
if size == '11' then UNDEFINED;
if index_align<1> != '0' then UNDEFINED;
ebytes = 4;  index = UInt(index_align<3>);
inc = if index_align<2> == '0' then 1 else 2;
alignment = if index_align<0> == '0' then 1 else 8;
d = UInt(D:Vd);  d2 = d + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d2 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *VST2 (single 2-element structure from one lane)*.

Assembler Symbols

<c> For encoding A1, A2 and A3: see *Standard assembler syntax fields*. This encoding must be unconditional.

For encoding T1, T2 and T3: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<size> Is the data size, encoded in “size”:

size	<size>
00	8
01	16
10	32

<list> Is a list containing the 64-bit names of the two SIMD&FP registers holding the element.
The list must be one of:

{ <Dd>[<index>], <Dd+1>[<index>] }
Single-spaced registers, encoded as "spacing" = 0.

{ <Dd>[<index>], <Dd+2>[<index>] }
Double-spaced registers, encoded as "spacing" = 1. Not permitted when <size> == 8.

The encoding of "spacing" depends on <size>:

<size> == 16
"spacing" is encoded in the "index_align<1>" field.

<size> == 32
"spacing" is encoded in the "index_align<2>" field.

The register <Dd> is encoded in the "D:Vd" field.
The permitted values and encoding of <index> depend on <size>:

<size> == 8
<index> is in the range 0 to 7, encoded in the "index_align<3:1>" field.

<size> == 16
<index> is in the range 0 to 3, encoded in the "index_align<3:2>" field.

<size> == 32
<index> is 0 or 1, encoded in the "index_align<3>" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

<align> Is the optional alignment.
Whenever <align> is omitted, the standard alignment is used, see *Unaligned data access*, and the encoding depends on <size>:

<size> == 8
Encoded in the "index_align<0>" field as 0.

<size> == 16
Encoded in the "index_align<0>" field as 0.

<size> == 32
Encoded in the "index_align<1:0>" field as 0b00.

Whenever <align> is present, the permitted values and encoding depend on <size>:

<size> == 8
<align> is 16, meaning 16-bit alignment, encoded in the "index_align<0>" field as 1.

<size> == 16
<align> is 32, meaning 32-bit alignment, encoded in the "index_align<0>" field as 1.

<size> == 32
<align> is 64, meaning 64-bit alignment, encoded in the "index_align<1:0>" field as 0b01.

: is the preferred separator before the <align> value, but the alignment can be specified as @<align>, see *Advanced SIMD addressing mode*.

<Rm> Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see *Advanced SIMD addressing mode*.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    address = R[n];  iswrite = TRUE;
    - = AArch32.CheckAlignment(address, alignment, AccType_VEC, iswrite);
    MemU[address,       ebytes] = Elem[D[d], index];
    MemU[address+ebytes,ebytes] = Elem[D[d2],index];
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 2*ebytes;
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

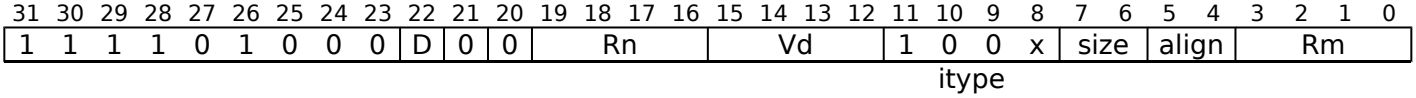
VST2 (multiple 2-element structures)

Store multiple 2-element structures from two or four registers stores multiple 2-element structures from two or four registers to memory, with interleaving. For more information, see *Element and structure load/store instructions*. Every element of each register is saved. For details of the addressing mode see *Advanced SIMD addressing mode*.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2) .

A1



Offset (Rm == 1111)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

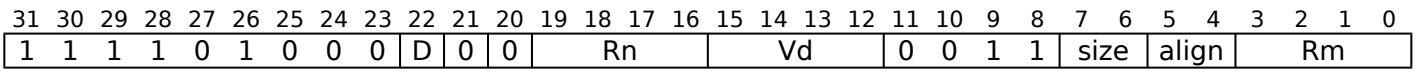
```
pairs = 1;  if align == '11' then UNDEFINED;
if size == '11' then UNDEFINED;
inc = if itype == '1001' then 2 else 1;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
d = UInt(D:Vd);  d2 = d + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d2+pairs > 32 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d2+pairs > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

A2



Offset (Rm == 1111)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

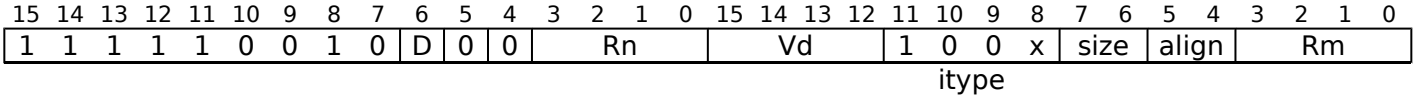
```
pairs = 2;  inc = 2;
if size == '11' then UNDEFINED;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
d = UInt(D:Vd);  d2 = d + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d2+pairs > 32 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d2+pairs > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

T1



Offset (Rm == 1111)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

```
pairs = 1;  if align == '11' then UNDEFINED;
if size == '11' then UNDEFINED;
inc = if itype == '1001' then 2 else 1;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
d = UInt(D:Vd);  d2 = d + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d2+pairs > 32 then UNPREDICTABLE;
```

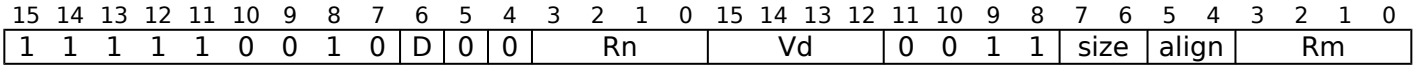
CONSTRAINED UNPREDICTABLE behavior

If `d2+pairs > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

T2



Offset (Rm == 1111)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

```
pairs = 2;  inc = 2;
if size == '11' then UNDEFINED;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
d = UInt(D:Vd);  d2 = d + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d2+pairs > 32 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d2+pairs > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *VST2 (multiple 2-element structures)*.

Related encodings: See *Advanced SIMD element or structure load/store* for the T32 instruction set, or *Advanced SIMD element or structure load/store* for the A32 instruction set.

Assembler Symbols

<c> For encoding A1 and A2: see *Standard assembler syntax fields*. This encoding must be unconditional.

For encoding T1 and T2: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<size> Is the data size, encoded in “size”:

size	<size>
00	8
01	16
10	32
11	RESERVED

<list> Is a list containing the 64-bit names of the SIMD&FP registers.

The list must be one of:

{ <Dd>, <Dd+1> }

Two single-spaced registers. Selects the A1 and T1 encodings of the instruction, and encoded in the "itype" field as 0b1000.

{ <Dd>, <Dd+2> }
Two double-spaced registers. Selects the A1 and T1 encodings of the instruction, and encoded in the "itype" field as 0b1001.

{ <Dd>, <Dd+1>, <Dd+2>, <Dd+3> }
Three single-spaced registers. Selects the A2 and T2 encodings of the instruction.

The register <Dd> is encoded in the "D:Vd" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

<align> Is the optional alignment.
Whenever <align> is omitted, the standard alignment is used, see *Unaligned data access*, and is encoded in the "align" field as 0b00.
Whenever <align> is present, the permitted values are:

64
64-bit alignment, encoded in the "align" field as 0b01.

128
128-bit alignment, encoded in the "align" field as 0b10.

256
256-bit alignment, encoded in the "align" field as 0b11. Available only if <list> contains four registers.

: is the preferred separator before the <align> value, but the alignment can be specified as @<align>, see *Advanced SIMD addressing mode*.

<Rm> Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see *Advanced SIMD addressing mode*.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    address = R[n];  iswrite = TRUE;
    - = AArch32.CheckAlignment(address, alignment, AccType_VEC, iswrite);
    for r = 0 to pairs-1
        for e = 0 to elements-1
            MemU[address,       ebytes] = Elem[D[d+r], e];
            MemU[address+ebytes,ebytes] = Elem[D[d2+r],e];
            address = address + 2*ebytes;
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 16*pairs;
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

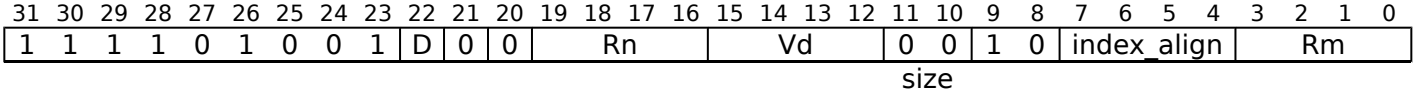
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VST3 (single 3-element structure from one lane)

Store single 3-element structure from one lane of three registers stores one 3-element structure to memory from corresponding elements of three registers. For details of the addressing mode see *Advanced SIMD addressing mode*. Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1 , A2 and A3) and T32 (T1 , T2 and T3) .

A1



Offset (Rm == 1111)

```
VST3{<c>}{<q>}.<size> <list>, [<Rn>]
```

Post-indexed (Rm == 1101)

```
VST3{<c>}{<q>}.<size> <list>, [<Rn>]!
```

Post-indexed (Rm != 11x1)

```
VST3{<c>}{<q>}.<size> <list>, [<Rn>], <Rm>
```

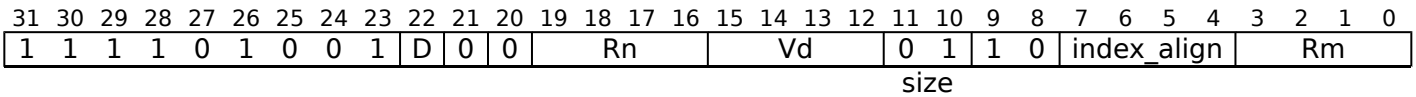
```
if size == '11' then UNDEFINED;
if index_align<0> != '0' then UNDEFINED;
ebytes = 1;  index = UInt(index_align<3:1>);  inc = 1;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d3 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

A2



Offset (Rm == 1111)

```
VST3{<c>}{<q>}.<size> <list>, [<Rn>]
```

Post-indexed (Rm == 1101)

```
VST3{<c>}{<q>}.<size> <list>, [<Rn>]!
```

Post-indexed (Rm != 11x1)

```
VST3{<c>}{<q>}.<size> <list>, [<Rn>], <Rm>
```

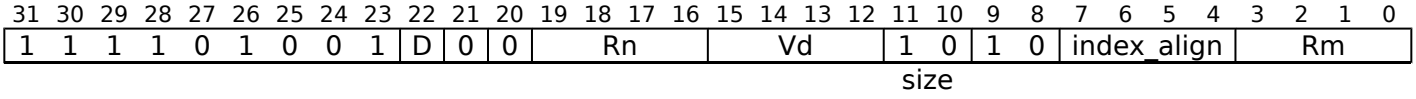
```
if size == '11' then UNDEFINED;
if index_align<0> != '0' then UNDEFINED;
ebytes = 2;  index = UInt(index_align<3:2>);
inc = if index_align<1> == '0' then 1 else 2;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d3 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

A3



Offset (Rm == 1111)

```
VST3{<c>}{<q>}.<size> <list>, [<Rn>]
```

Post-indexed (Rm == 1101)

```
VST3{<c>}{<q>}.<size> <list>, [<Rn>]!
```

Post-indexed (Rm != 11x1)

```
VST3{<c>}{<q>}.<size> <list>, [<Rn>], <Rm>
```

```
if size == '11' then UNDEFINED;
if index_align<1:0> != '00' then UNDEFINED;
ebytes = 4;  index = UInt(index_align<3>);
inc = if index_align<2> == '0' then 1 else 2;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;
```

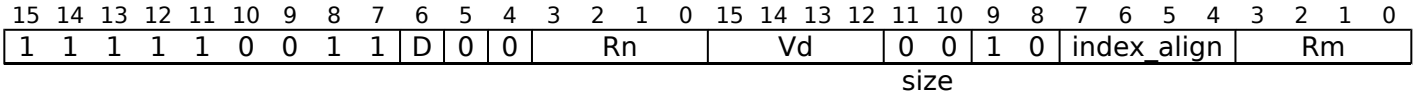
CONSTRAINED UNPREDICTABLE behavior

If `d3 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

T1



Offset (Rm == 1111)

```
VST3{<c>}{<q>}.<size> <list>, [<Rn>]
```

Post-indexed (Rm == 1101)

```
VST3{<c>}{<q>}.<size> <list>, [<Rn>]!
```

Post-indexed (Rm != 11x1)

```
VST3{<c>}{<q>}.<size> <list>, [<Rn>], <Rm>
```

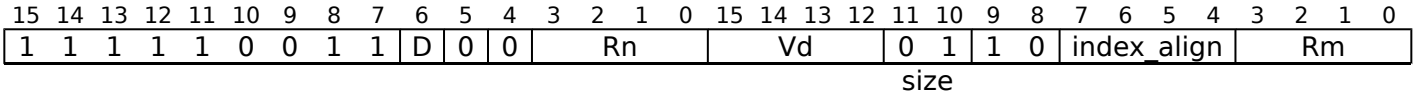
```
if size == '11' then UNDEFINED;
if index_align<0> != '0' then UNDEFINED;
ebytes = 1;  index = UInt(index_align<3:1>);  inc = 1;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d3 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

T2



Offset (Rm == 1111)

```
VST3{<c>}{<q>}.<size> <list>, [<Rn>]
```

Post-indexed (Rm == 1101)

```
VST3{<c>}{<q>}.<size> <list>, [<Rn>]!
```

Post-indexed (Rm != 11x1)

```
VST3{<c>}{<q>}.<size> <list>, [<Rn>], <Rm>
```

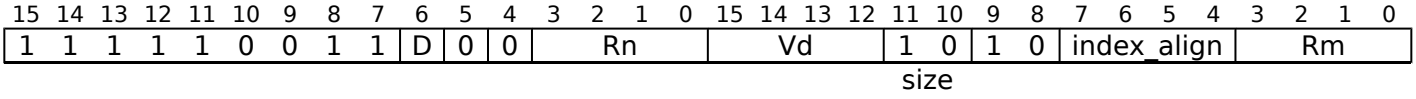
```
if size == '11' then UNDEFINED;
if index_align<0> != '0' then UNDEFINED;
ebytes = 2;  index = UInt(index_align<3:2>);
inc = if index_align<1> == '0' then 1 else 2;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d3 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

T3



Offset (Rm == 1111)

```
VST3{<c>}{<q>}.<size> <list>, [<Rn>]
```

Post-indexed (Rm == 1101)

```
VST3{<c>}{<q>}.<size> <list>, [<Rn>]!
```

Post-indexed (Rm != 11x1)

```
VST3{<c>}{<q>}.<size> <list>, [<Rn>], <Rm>
```

```
if size == '11' then UNDEFINED;
if index_align<1:0> != '00' then UNDEFINED;
ebytes = 4;  index = UInt(index_align<3>);
inc = if index_align<2> == '0' then 1 else 2;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d3 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *VST3 (single 3-element structure from one lane)*.

Assembler Symbols

<c> For encoding A1, A2 and A3: see *Standard assembler syntax fields*. This encoding must be unconditional.

For encoding T1, T2 and T3: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<size> Is the data size, encoded in “size”:

size	<size>
00	8
01	16
10	32

<list> Is a list containing the 64-bit names of the three SIMD&FP registers holding the element.
The list must be one of:

{ <Dd>[<index>], <Dd+1>[<index>], <Dd+2>[<index>] }
Single-spaced registers, encoded as "spacing" = 0.

{ <Dd>[<index>], <Dd+2>[<index>], <Dd+4>[<index>] }
Double-spaced registers, encoded as "spacing" = 1. Not permitted when <size> == 8.

The encoding of "spacing" depends on <size>:

<size> == 8
"spacing" is encoded in the "index_align<0>" field.

<size> == 16
"spacing" is encoded in the "index_align<1>" field, and "index_align<0>" is set to 0.

<size> == 32
"spacing" is encoded in the "index_align<2>" field, and "index_align<1:0>" is set to 0b00.

The register <Dd> is encoded in the "D:Vd" field.
The permitted values and encoding of <index> depend on <size>:

<size> == 8
<index> is in the range 0 to 7, encoded in the "index_align<3:1>" field.

<size> == 16
<index> is in the range 0 to 3, encoded in the "index_align<3:2>" field.

<size> == 32
<index> is 0 or 1, encoded in the "index_align<3>" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

<Rm> Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see *Advanced SIMD addressing mode*.

Alignment

Standard alignment rules apply, see *Alignment support*.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    address = R[n];
    MemU[address,         ebytes] = Elem[D[d], index];
    MemU[address+ebytes,  ebytes] = Elem[D[d2],index];
    MemU[address+2*ebytes,ebytes] = Elem[D[d3],index];
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 3*ebytes;
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

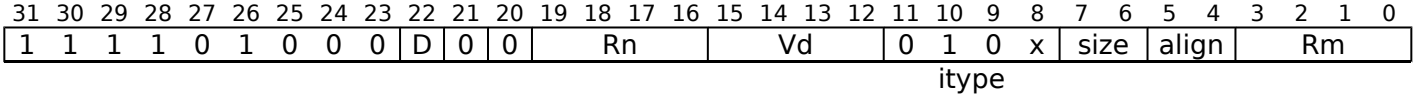
VST3 (multiple 3-element structures)

Store multiple 3-element structures from three registers stores multiple 3-element structures to memory from three registers, with interleaving. For more information, see *Element and structure load/store instructions*. Every element of each register is saved. For details of the addressing mode see *Advanced SIMD addressing mode*.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



Offset (Rm == 1111)

```
VST3{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

```
VST3{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VST3{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

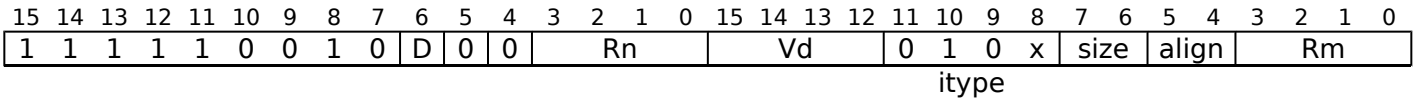
```
if size == '11' || align<1> == '1' then UNDEFINED;
case itype of
    when '0100'
        inc = 1;
    when '0101'
        inc = 2;
    otherwise
        SEE "Related encodings";
alignment = if align<0> == '0' then 1 else 8;
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d3 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

T1



Offset (Rm == 1111)

```
VST3{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

```
VST3{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VST3{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

```
if size == '11' || align<1> == '1' then UNDEFINED;
case itype of
    when '0100'
        inc = 1;
    when '0101'
        inc = 2;
    otherwise
        SEE "Related encodings";
alignment = if align<0> == '0' then 1 else 8;
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d3 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *VST3 (multiple 3-element structures)*.

Related encodings: See *Advanced SIMD element or structure load/store* for the T32 instruction set, or *Advanced SIMD element or structure load/store* for the A32 instruction set.

Assembler Symbols

<c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<size> Is the data size, encoded in “size”:

size	<size>
00	8
01	16
10	32
11	RESERVED

<list> Is a list containing the 64-bit names of the SIMD&FP registers.

The list must be one of:

{ <Dd>, <Dd+1>, <Dd+2> }

Single-spaced registers, encoded in the "itype" field as 0b0100.

{ <Dd>, <Dd+2>, <Dd+4> }

Double-spaced registers, encoded in the "itype" field as 0b0101.

The register <Dd> is encoded in the "D:Vd" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

<align> Is the optional alignment.

Whenever <align> is omitted, the standard alignment is used, see *Unaligned data access*, and is encoded in the "align" field as 0b00.

Whenever <align> is present, the only permitted values is 64, meaning 64-bit alignment, encoded in the "align" field as 0b01.

: is the preferred separator before the <align> value, but the alignment can be specified as @<align>, see *Advanced SIMD addressing mode*.

<Rm> Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see *Advanced SIMD addressing mode*.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    address = R[n];  iswrite = TRUE;
    - = AArch32.CheckAlignment(address, alignment, AccType_VEC, iswrite);
    for e = 0 to elements-1
        MemU[address,         ebytes] = Elem[D[d], e];
        MemU[address+ebytes,  ebytes] = Elem[D[d2],e];
        MemU[address+2*ebytes,ebytes] = Elem[D[d3],e];
        address = address + 3*ebytes;
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 24;
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

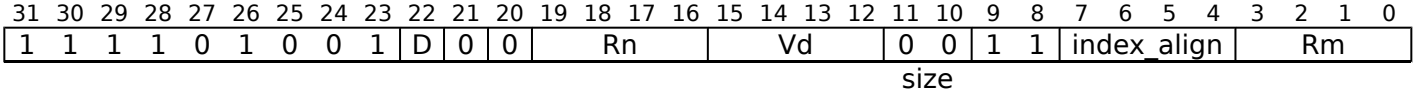
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VST4 (single 4-element structure from one lane)

Store single 4-element structure from one lane of four registers stores one 4-element structure to memory from corresponding elements of four registers. For details of the addressing mode see *Advanced SIMD addressing mode*. Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1 , A2 and A3) and T32 (T1 , T2 and T3) .

A1



Offset (Rm == 1111)

```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

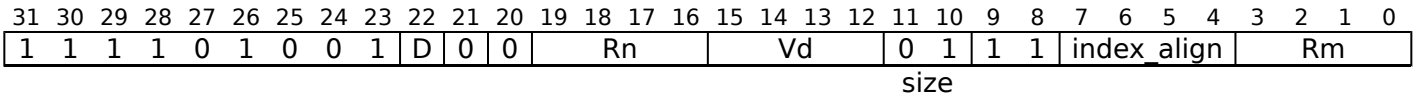
```
if size == '11' then UNDEFINED;
if size != '00' then SEE "Related encodings";
ebytes = 1;  index = UInt(index_align<3:1>);  inc = 1;
alignment = if index_align<0> == '0' then 1 else 4;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  d4 = d3 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d4 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

A2



Offset (Rm == 1111)

```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

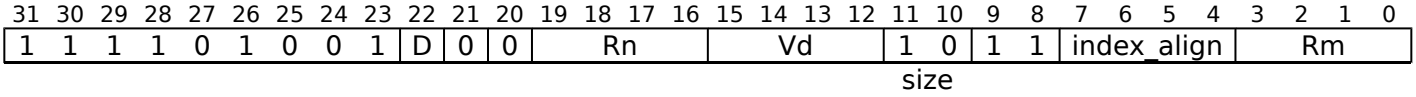
```
if size == '11' then UNDEFINED;
if size != '01' then SEE "Related encodings";
ebytes = 2;  index = UInt(index_align<3:2>);
inc = if index_align<1> == '0' then 1 else 2;
alignment = if index_align<0> == '0' then 1 else 8;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  d4 = d3 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d4 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

A3



Offset (Rm == 1111)

```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

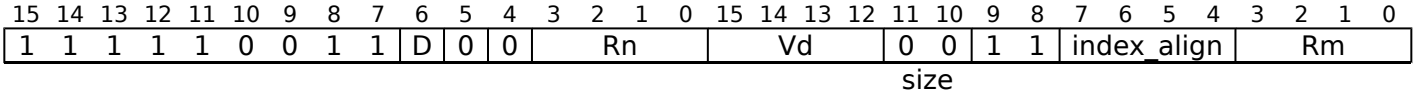
```
if size == '11' then UNDEFINED;
if size != '10' then SEE "Related encodings";
if index_align<1:0> == '11' then UNDEFINED;
ebytes = 4;  index = UInt(index_align<3>);
inc = if index_align<2> == '0' then 1 else 2;
alignment = if index_align<1:0> == '00' then 1 else 4 << UInt(index_align<1:0>);
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  d4 = d3 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d4 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

T1



Offset (Rm == 1111)

```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

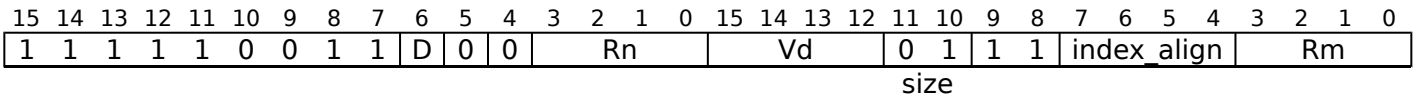
```
if size == '11' then UNDEFINED;
if size != '00' then SEE "Related encodings";
ebytes = 1;  index = UInt(index_align<3:1>);  inc = 1;
alignment = if index_align<0> == '0' then 1 else 4;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  d4 = d3 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d4 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

T2



Offset (Rm == 1111)

```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

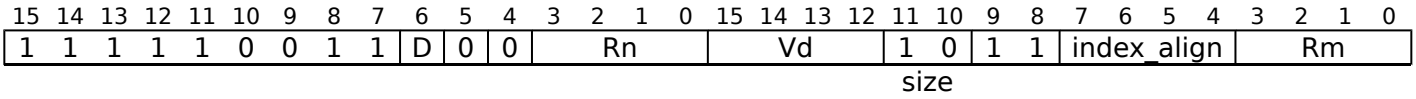
```
if size == '11' then UNDEFINED;
if size != '01' then SEE "Related encodings";
ebytes = 2;  index = UInt(index_align<3:2>);
inc = if index_align<1> == '0' then 1 else 2;
alignment = if index_align<0> == '0' then 1 else 8;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  d4 = d3 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d4 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

T3



Offset (Rm == 1111)

```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm != 11x1)

```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Post-indexed (Rm == 1101)

```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

```
if size == '11' then UNDEFINED;
if size != '10' then SEE "Related encodings";
if index_align<1:0> == '11' then UNDEFINED;
ebytes = 4;  index = UInt(index_align<3>);
inc = if index_align<2> == '0' then 1 else 2;
alignment = if index_align<1:0> == '00' then 1 else 4 << UInt(index_align<1:0>);
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  d4 = d3 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d4 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *VST4 (single 4-element structure from one lane)*.

Assembler Symbols

<c> For encoding A1, A2 and A3: see *Standard assembler syntax fields*. This encoding must be unconditional.

For encoding T1, T2 and T3: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<size> Is the data size, encoded in “size”:

size	<size>
00	8
01	16
10	32

<list> Is a list containing the 64-bit names of the four SIMD&FP registers holding the element.
The list must be one of:

{ <Dd>[<index>], <Dd+1>[<index>], <Dd+2>[<index>], <Dd+3>[<index>] }
Single-spaced registers, encoded as "spacing" = 0.

{ <Dd>[<index>], <Dd+2>[<index>], <Dd+4>[<index>], <Dd+6>[<index>] }
Double-spaced registers, encoded as "spacing" = 1. Not permitted when <size> == 8.

The encoding of "spacing" depends on <size>:

<size> == 16
"spacing" is encoded in the "index_align<1>" field.

<size> == 32
"spacing" is encoded in the "index_align<2>" field.

The register <Dd> is encoded in the "D:Vd" field.
The permitted values and encoding of <index> depend on <size>:

<size> == 8
<index> is in the range 0 to 7, encoded in the "index_align<3:1>" field.

<size> == 16
<index> is in the range 0 to 3, encoded in the "index_align<3:2>" field.

<size> == 32
<index> is 0 or 1, encoded in the "index_align<3>" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

<align> Is the optional alignment.
Whenever <align> is omitted, the standard alignment is used, see *Unaligned data access*, and the encoding depends on <size>:

<size> == 8
Encoded in the "index_align<0>" field as 0.

<size> == 16
Encoded in the "index_align<0>" field as 0.

<size> == 32
Encoded in the "index_align<1:0>" field as 0b00.

Whenever <align> is present, the permitted values and encoding depend on <size>:

<size> == 8
<align> is 32, meaning 32-bit alignment, encoded in the "index_align<0>" field as 1.

<size> == 16
<align> is 64, meaning 64-bit alignment, encoded in the "index_align<0>" field as 1.

<size> == 32
<align> can be 64 or 128. 64-bit alignment is encoded in the "index_align<1:0>" field as 0b01, and 128-bit alignment is encoded in the "index_align<1:0>" field as 0b10.

: is the preferred separator before the <align> value, but the alignment can be specified as @<align>, see *Advanced SIMD addressing mode*.

<Rm> Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see *Advanced SIMD addressing mode*.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    address = R[n];  iswrite = TRUE;
    - = AArch32.CheckAlignment(address, alignment, AccType_VEC, iswrite);
    MemU[address,         ebytes] = Elem[D[d], index];
    MemU[address+ebytes,  ebytes] = Elem[D[d2],index];
    MemU[address+2*ebytes,ebytes] = Elem[D[d3],index];
    MemU[address+3*ebytes,ebytes] = Elem[D[d4],index];
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 4*ebytes;
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

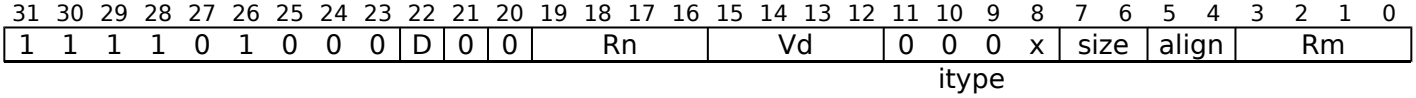
VST4 (multiple 4-element structures)

Store multiple 4-element structures from four registers stores multiple 4-element structures to memory from four registers, with interleaving. For more information, see *Element and structure load/store instructions*. Every element of each register is saved. For details of the addressing mode see *Advanced SIMD addressing mode*.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



Offset (Rm == 1111)

```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

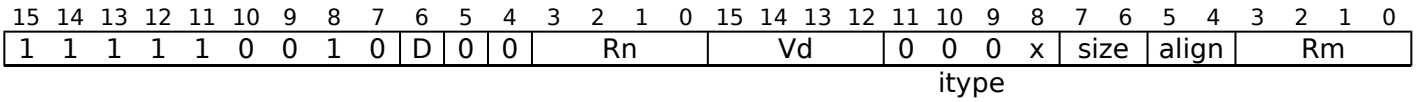
```
if size == '11' then UNDEFINED;
case itype of
    when '0000'
        inc = 1;
    when '0001'
        inc = 2;
    otherwise
        SEE "Related encodings";
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  d4 = d3 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d4 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

T1



Offset (Rm == 1111)

```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

```
if size == '11' then UNDEFINED;
case itype of
    when '0000'
        inc = 1;
    when '0001'
        inc = 2;
    otherwise
        SEE "Related encodings";
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size);  elements = 8 DIV ebytes;
d = UInt(D:Vd);  d2 = d + inc;  d3 = d2 + inc;  d4 = d3 + inc;  n = UInt(Rn);  m = UInt(Rm);
wback = (m != 15);  register_index = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d4 > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The memory locations specified by the instruction and the number of registers specified by the instruction become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *VST4 (multiple 4-element structures)*.

Related encodings: See *Advanced SIMD element or structure load/store* for the T32 instruction set, or *Advanced SIMD element or structure load/store* for the A32 instruction set.

Assembler Symbols

<c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<size> Is the data size, encoded in “size”:

size	<size>
00	8
01	16
10	32
11	RESERVED

<list> Is a list containing the 64-bit names of the SIMD&FP registers.

The list must be one of:

{ <Dd>, <Dd+1>, <Dd+2>, <Dd+3> }

Single-spaced registers, encoded in the "itype" field as 0b0000.

{ <Dd>, <Dd+2>, <Dd+4>, <Dd+6> }

Double-spaced registers, encoded in the "itype" field as 0b0001.

The register <Dd> is encoded in the "D:Vd" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

<align> Is the optional alignment.

Whenever <align> is omitted, the standard alignment is used, see *Unaligned data access*, and is encoded in the "align" field as 0b00.

Whenever <align> is present, the permitted values are:

64
64-bit alignment, encoded in the "align" field as 0b01.

128
128-bit alignment, encoded in the "align" field as 0b10.

256
256-bit alignment, encoded in the "align" field as 0b11.

: is the preferred separator before the <align> value, but the alignment can be specified as @<align>, see *Advanced SIMD addressing mode*.

<Rm> Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see *Advanced SIMD addressing mode*.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    address = R[n];  iswrite = TRUE;
    - = AArch32.CheckAlignment(address, alignment, AccType_VEC, iswrite);
    for e = 0 to elements-1
        MemU[address,         ebytes] = Elem[D[d], e];
        MemU[address+ebytes,  ebytes] = Elem[D[d2],e];
        MemU[address+2*ebytes,ebytes] = Elem[D[d3],e];
        MemU[address+3*ebytes,ebytes] = Elem[D[d4],e];
        address = address + 4*ebytes;
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 32;
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

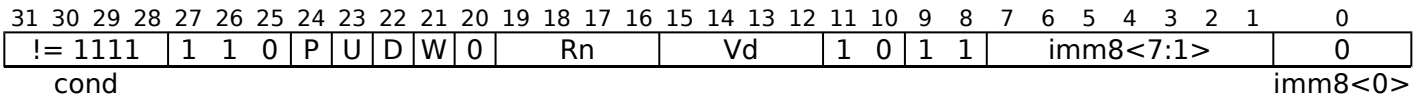
VSTM, VSTMDB, VSTMIA

Store multiple SIMD&FP registers stores multiple registers from the Advanced SIMD and floating-point register file to consecutive memory locations using an address from a general-purpose register.
Depending on settings in the *CPACR*, *NSACR*, *HCPTR*, and *FPEXC* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

This instruction is used by the alias VPUSH.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2) .

A1



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

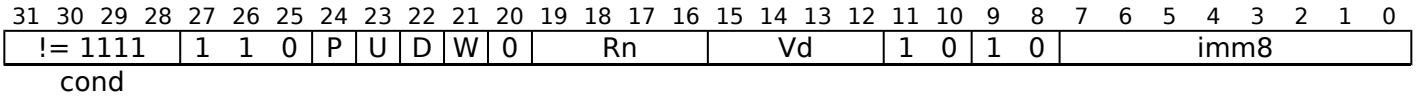
If `regs == 0`, then one of the following behaviors must occur:

- 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



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

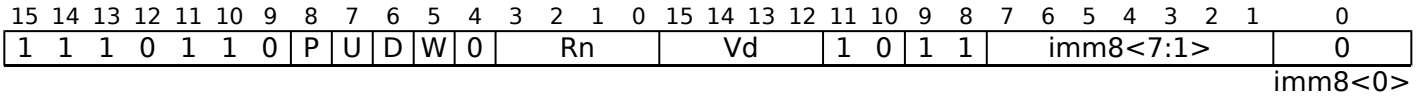
If `regs == 0`, then one of the following behaviors must occur:

- 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



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

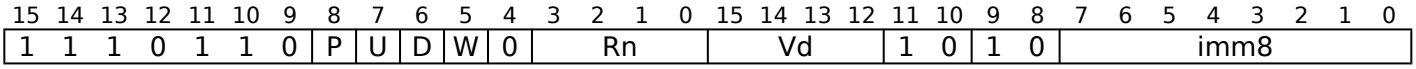
If `regs == 0`, then one of the following behaviors must occur:

- 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.

T2



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();  CheckVFPEnabled(TRUE);
    address = if add then R[n] else R[n]-imm32;
    for r = 0 to regs-1
        if single_regs then
            MemA[address,4] = S[d+r];  address = address+4;
        else
            // Store as two word-aligned words in the correct order for current endianness.
            MemA[address,4] = if BigEndian(AccType_ATOMIC) then D[d+r]<63:32> else D[d+r]<31:0>;
            MemA[address+4,4] = if BigEndian(AccType_ATOMIC) then D[d+r]<31:0> else D[d+r]<63:32>;
            address = address+8;
    if wback then R[n] = if add then R[n]+imm32 else R[n]-imm32;
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

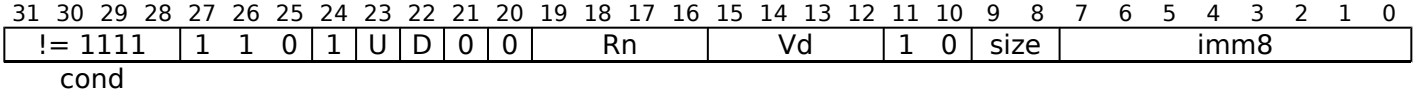
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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



Half-precision scalar (size == 01)
(FEAT_FP16)

```
VSTR{<c>}{<q>}.16 <Sd>, [<Rn>{, #{+/-}<imm>}]
```

Single-precision scalar (size == 10)

```
VSTR{<c>}{<q>}{.32} <Sd>, [<Rn>{, #{+/-}<imm>}]
```

Double-precision scalar (size == 11)

```
VSTR{<c>}{<q>}{.64} <Dd>, [<Rn>{, #{+/-}<imm>}]
```

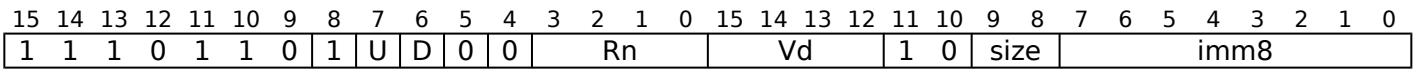
```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
esize = 8 << UInt(size);  add = (U == '1');
imm32 = if esize == 16 then ZeroExtend(imm8:'0', 32) else ZeroExtend(imm8:'00', 32);
case size of
    when '01' d = UInt(Vd:D);
    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



Half-precision scalar (size == 01)
(FEAT_FP16)

```
VSTR{<c>}{<q>}.16 <Sd>, [<Rn>{, #{+/-}<imm>}]
```

Single-precision scalar (size == 10)

```
VSTR{<c>}{<q>}{.32} <Sd>, [<Rn>{, #{+/-}<imm>}]
```

Double-precision scalar (size == 11)

```
VSTR{<c>}{<q>}{.64} <Dd>, [<Rn>{, #{+/-}<imm>}]
```

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock()  then UNPREDICTABLE;
esize = 8 << UInt(size);  add = (U == '1');
imm32 = if esize == 16 then ZeroExtend(imm8:'0', 32) else ZeroExtend(imm8:'00', 32);
case size of
    when '01' d = UInt(Vd:D);
    when '10' d = UInt(Vd:D);
    when '11' d = UInt(D:Vd);
n = UInt(Rn);
if n == 15 && CurrentInstrSet() != InstrSet_A32 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c>	See *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
.64	Is an optional data size specifier for 64-bit memory accesses that can be used in the assembler source code, but is otherwise ignored.
<Dd>	Is the 64-bit name of the SIMD&FP source register, encoded in the "D:Vd" field.
.32	Is an optional data size specifier for 32-bit memory accesses that can be used in the assembler source code, but is otherwise ignored.
<Sd>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Vd:D" field.
<Rn>	Is the general-purpose base register, encoded in the "Rn" field. The PC can be used, but this is deprecated.
+/-	Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<imm>	For the single-precision scalar or double-precision scalar variants: is the optional unsigned immediate byte offset, a multiple of 4, in the range 0 to 1020, defaulting to 0, and encoded in the "imm8" field as <imm>/4. For the half-precision scalar variant: is the optional unsigned immediate byte offset, a multiple of 2, in the range 0 to 510, defaulting to 0, and encoded in the "imm8" field as <imm>/2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    address = if add then (R[n] + imm32) else (R[n] - imm32);
    case esize of
        when 16
            MemA[address,2] = S[d]<15:0>;
        when 32
            MemA[address,4] = S[d];
        when 64
            // Store as two word-aligned words in the correct order for current endianness.
            MemA[address,4]   = if BigEndian(AccType_ATOMIC) then D[d]<63:32> else D[d]<31:0>;
            MemA[address+4,4] = if BigEndian(AccType_ATOMIC) then D[d]<31:0>  else D[d]<63:32>;
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

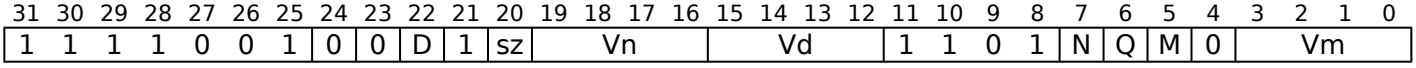
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VSUB (floating-point)

Vector Subtract (floating-point) subtracts the elements of one vector from the corresponding elements of another vector, and places the results in the destination vector.
Depending on settings in the *CPACR*, *NSACR*, *HCPTR*, and *FPEXC* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1 and A2) and T32 (T1 and T2) .

A1



64-bit SIMD vector (Q == 0)

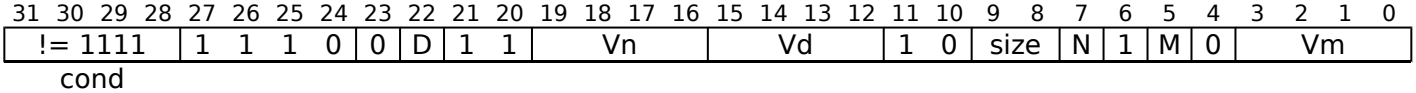
```
VSUB{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

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



Half-precision scalar (size == 01)
(FEAT_FP16)

```
VSUB{<c>}{<q>}.F16 {<Sd>,} <Sn>, <Sm>
```

Single-precision scalar (size == 10)

```
VSUB{<c>}{<q>}.F32 {<Sd>,} <Sn>, <Sm>
```

Double-precision scalar (size == 11)

```
VSUB{<c>}{<q>}.F64 {<Dd>,} <Dn>, <Dm>
```

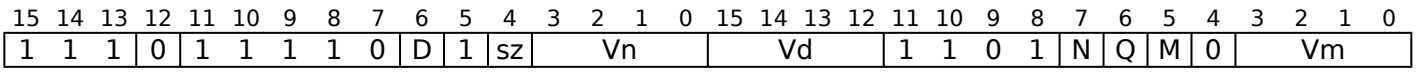
```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
advsimd = FALSE;
case size of
    when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '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)

```
VSUB{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VSUB{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

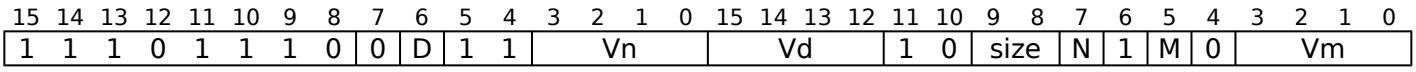
```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
advsimd = TRUE;
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd);  n = 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



Half-precision scalar (size == 01)
(FEAT_FP16)

```
VSUB{<c>}{<q>}.F16 {<Sd>,} <Sn>, <Sm>
```

Single-precision scalar (size == 10)

```
VSUB{<c>}{<q>}.F32 {<Sd>,} <Sn>, <Sm>
```

Double-precision scalar (size == 11)

```
VSUB{<c>}{<q>}.F64 {<Dd>,} <Dn>, <Dm>
```

```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock()  then UNPREDICTABLE;
advsimd = FALSE;
case size of
    when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.
For encoding A2, T1 and T2: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the vectors, encoded in “sz”:

sz	<dt>
0	F32
1	F16

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

<Sn> Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.

<Sm> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
    if advsimd then  // Advanced SIMD instruction
        for r = 0 to regs-1
            for e = 0 to elements-1
                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_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

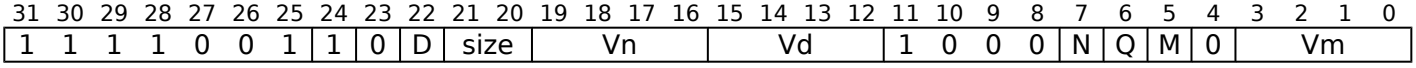
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VSUB (integer)

Vector Subtract (integer) subtracts the elements of one vector from the corresponding elements of another vector, and places the results in the destination vector.
Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



64-bit SIMD vector (Q == 0)

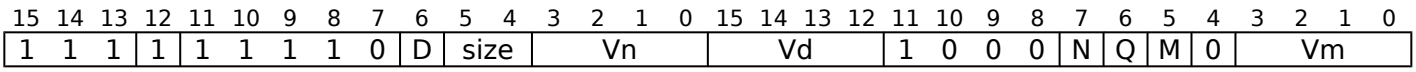
```
VSUB{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VSUB{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1



64-bit SIMD vector (Q == 0)

```
VSUB{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VSUB{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.
For encoding T1: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the vectors, encoded in “size”:

size	<dt>
00	I8
01	I16
10	I32
11	I64

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            Elem[D[d+r],e,esize] = Elem[D[n+r],e,esize] - Elem[D[m+r],e,esize];
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VSUBHN

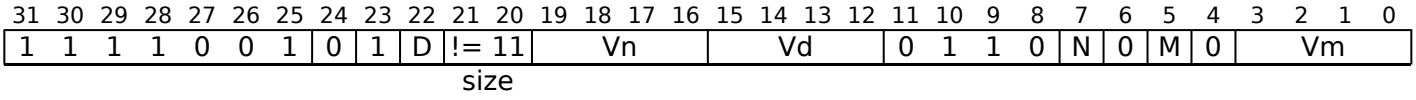
Vector Subtract and Narrow, returning High Half subtracts the elements of one quadword vector from the corresponding elements of another quadword vector, takes the most significant half of each result, and places the final results in a doubleword vector. The results are truncated. For rounded results, see *VRSUBHN*.

There is no distinction between signed and unsigned integers.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

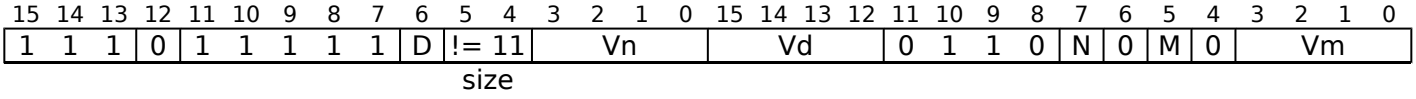


A1

```
VSUBHN{<c>}{<q>}.<dt> <Dd>, <Qn>, <Qm>
```

```
if size == '11' then SEE "Related encodings";
if Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

T1



T1

```
VSUBHN{<c>}{<q>}.<dt> <Dd>, <Qn>, <Qm>
```

```
if size == '11' then SEE "Related encodings";
if Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

Related encodings: See *Advanced SIMD data-processing* for the T32 instruction set, or *Advanced SIMD data-processing* for the A32 instruction set.

Assembler Symbols

<c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the operands, encoded in “size”:

size	<dt>
00	I16
01	I32
10	I64

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for e = 0 to elements-1
        result = Elem[Qin[n>>1],e,2*esize] - Elem[Qin[m>>1],e,2*esize];
        Elem[D[d],e,esize] = result<2*esize-1:esize>;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

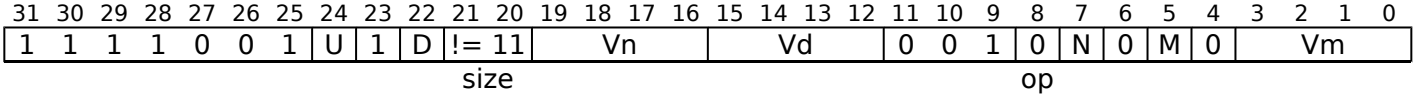
VSUBL

Vector Subtract Long subtracts the elements of one doubleword vector from the corresponding elements of another doubleword vector, and places the results in a quadword vector. Before subtracting, it sign-extends or zero-extends the elements of both operands.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

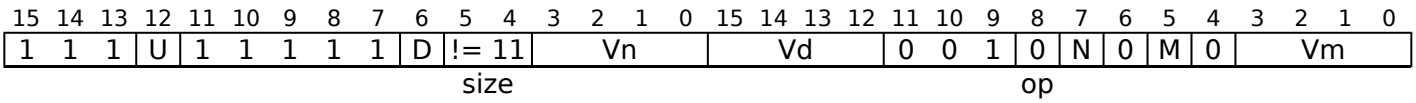


A1

```
VSUBL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>
```

```
if size == '11' then SEE "Related encodings";
if Vd<0> == '1' || (op == '1' && Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;  is_vsubw = (op == '1');
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

T1



T1

```
VSUBL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>
```

```
if size == '11' then SEE "Related encodings";
if Vd<0> == '1' || (op == '1' && Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;  is_vsubw = (op == '1');
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

Related encodings: See *Advanced SIMD data-processing* for the T32 instruction set, or *Advanced SIMD data-processing* for the A32 instruction set.

Assembler Symbols

<c>	For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional. For encoding T1: see *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<dt>	Is the data type for the elements of the second operand vector, encoded in “U:size”:

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	00	U8
1	01	U16
1	10	U32

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for e = 0 to elements-1
        if is_vsubw then
            op1 = Int(Elem[Qin[n>>1],e,2*esize], unsigned);
        else
            op1 = Int(Elem[Din[n],e,esize], unsigned);
        result = op1 - Int(Elem[Din[m],e,esize], unsigned);
        Elem[Q[d>>1],e,2*esize] = result<2*esize-1:0>;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

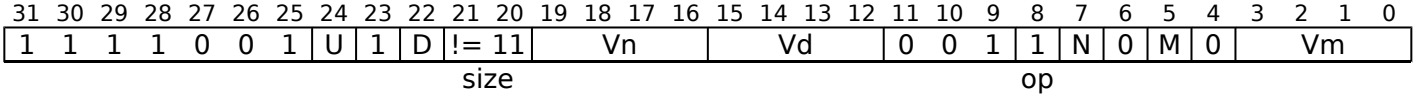
VSUBW

Vector Subtract Wide subtracts the elements of a doubleword vector from the corresponding elements of a quadword vector, and places the results in another quadword vector. Before subtracting, it sign-extends or zero-extends the elements of the doubleword operand.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

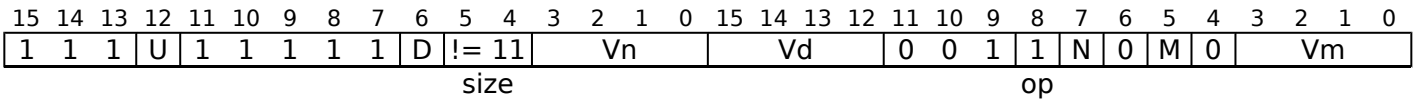


A1

```
VSUBW{<c>}{<q>}.<dt> {<Qd>,} <Qn>, <Dm>
```

```
if size == '11' then SEE "Related encodings";
if Vd<0> == '1' || (op == '1' && Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;  is_vsubw = (op == '1');
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

T1



T1

```
VSUBW{<c>}{<q>}.<dt> {<Qd>,} <Qn>, <Dm>
```

```
if size == '11' then SEE "Related encodings";
if Vd<0> == '1' || (op == '1' && Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;  is_vsubw = (op == '1');
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
```

Related encodings: See *Advanced SIMD data-processing* for the T32 instruction set, or *Advanced SIMD data-processing* for the A32 instruction set.

Assembler Symbols

<c>	For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional. For encoding T1: see *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<dt>	Is the data type for the elements of the second operand vector, encoded in “U:size”:

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	00	U8
1	01	U16
1	10	U32

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for e = 0 to elements-1
        if is_vsubw then
            op1 = Int(Elem[Qin[n>>1],e,2*esize], unsigned);
        else
            op1 = Int(Elem[Din[n],e,esize], unsigned);
        result = op1 - Int(Elem[Din[m],e,esize], unsigned);
        Elem[Q[d>>1],e,2*esize] = result<2*esize-1:0>;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

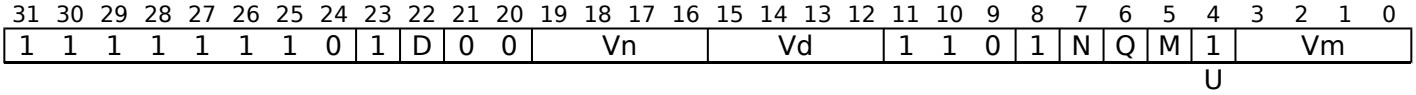
VSUDOT (by element)

Dot Product index form with signed and unsigned integers. This instruction performs the dot product of the four signed 8-bit integer values in each 32-bit element of the first source register with the four unsigned 8-bit integer values in an indexed 32-bit element of the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

From Armv8.2, this is an OPTIONAL instruction. *ID_ISAR6*.I8MM indicates whether this instruction is supported in the T32 and A32 instruction sets.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1
(FEAT_AA32I8MM)



64-bit SIMD vector (Q == 0)

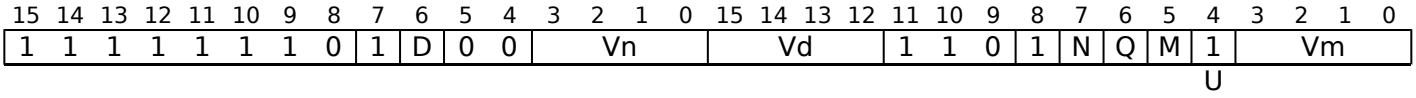
```
VSUDOT{<q>}.U8 <Dd>, <Dn>, <Dm>[<index>]
```

128-bit SIMD vector (Q == 1)

```
VSUDOT{<q>}.U8 <Qd>, <Qn>, <Dm>[<index>]
```

```
if !HaveAArch32Int8MatMulExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
boolean op1_unsigned = (U == '0');
boolean op2_unsigned = (U == '1');
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(Vm);
integer i = UInt(M);
integer regs = if Q == '1' then 2 else 1;
```

T1
(FEAT_AA32I8MM)



64-bit SIMD vector (Q == 0)

```
VSUDOT{<q>}.U8 <Dd>, <Dn>, <Dm>[<index>]
```

128-bit SIMD vector (Q == 1)

```
VSUDOT{<q>}.U8 <Qd>, <Qn>, <Dm>[<index>]
```

```
if InITBlock() then UNPREDICTABLE;
if !HaveAArch32Int8MatMulExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
boolean op1_unsigned = (U == '0');
boolean op2_unsigned = (U == '1');
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(Vm);
integer i = UInt(M);
integer regs = if Q == '1' then 2 else 1;
```

Assembler Symbols

<q>	See *Standard assembler syntax fields*.
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qn>	Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "Vm" field.
<index>	Is the element index in the range 0 to 1, encoded in the "M" field.

Operation

```
CheckAdvSIMDEnabled();
bits(64) operand1;
bits(64) operand2;
bits(64) result;

operand2 = Din[m];
for r = 0 to regs-1
    operand1 = Din[n+r];
    result = Din[d+r];
    for e = 0 to 1
        bits(32) res = Elem[result, e, 32];
        for b = 0 to 3
            element1 = Int(Elem[operand1, 4 * e + b, 8], op1_unsigned);
            element2 = Int(Elem[operand2, 4 * i + b, 8], op2_unsigned);
            res = res + element1 * element2;
        Elem[result, e, 32] = res;
    D[d+r] = result;
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

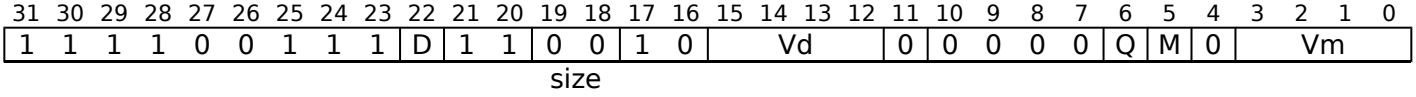
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VSWP

Vector Swap exchanges the contents of two vectors. The vectors can be either doubleword or quadword. There is no distinction between data types.
Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



64-bit SIMD vector (Q == 0)

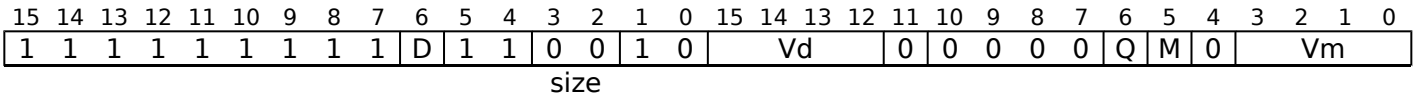
```
VSWP{<c>}{<q>}{.<dt>} <Dd>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VSWP{<c>}{<q>}{.<dt>} <Qd>, <Qm>
```

```
if size != '00' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1



64-bit SIMD vector (Q == 0)

```
VSWP{<c>}{<q>}{.<dt>} <Dd>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VSWP{<c>}{<q>}{.<dt>} <Qd>, <Qm>
```

```
if size != '00' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>	For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional. For encoding T1: see *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<dt>	An optional data type. It is ignored by assemblers, and does not affect the encoding.
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qm>	Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dm>	Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        if d == m then
            D[d+r] = bits(64) UNKNOWN;
        else
            D[d+r] = Din[m+r];
            D[m+r] = Din[d+r];
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

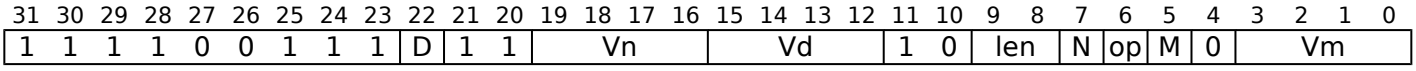
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VTBL, VTBX

Vector Table Lookup uses byte indexes in a control vector to look up byte values in a table and generate a new vector. Indexes out of range return 0.
Vector Table Extension works in the same way, except that indexes out of range leave the destination element unchanged.
Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



VTBL (op == 0)

```
VTBL{<c>}{<q>}.8 <Dd>, <list>, <Dm>
```

VTBX (op == 1)

```
VTBX{<c>}{<q>}.8 <Dd>, <list>, <Dm>
```

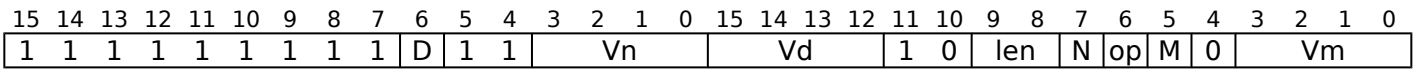
```
is_vtbl = (op == '0');  length = UInt(len)+1;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
if n+length > 32 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `n + length > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- One or more of the SIMD and floating-point registers are UNKNOWN. This behavior does not affect any general-purpose registers.

T1



VTBL (op == 0)

```
VTBL{<c>}{<q>}.8 <Dd>, <list>, <Dm>
```

VTBX (op == 1)

```
VTBX{<c>}{<q>}.8 <Dd>, <list>, <Dm>
```

```
is_vtbl = (op == '0');  length = UInt(len)+1;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);
if n+length > 32 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `n + length > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.

- The instruction executes as NOP.
- One or more of the SIMD and floating-point registers are UNKNOWN. This behavior does not affect any general-purpose registers.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

<c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.
For encoding T1: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<list> The vectors containing the table. It must be one of:

{<Dn>}
Encoded as len = 0b00.

{<Dn>, <Dn+1>}
Encoded as len = 0b01.

{<Dn>, <Dn+1>, <Dn+2>}
Encoded as len = 0b10.

{<Dn>, <Dn+1>, <Dn+2>, <Dn+3>}
Encoded as len = 0b11.

<Dm> Is the 64-bit name of the SIMD&FP source register holding the indices, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();

    // Create 256-bit = 32-byte table variable, with zeros in entries that will not be used.
    table3 = if length == 4 then D[n+3] else Zeros(64);
    table2 = if length >= 3 then D[n+2] else Zeros(64);
    table1 = if length >= 2 then D[n+1] else Zeros(64);
    table = table3 : table2 : table1 : D[n];

    for i = 0 to 7
        index = UInt(Elem[D[m],i,8]);
        if index < 8*length then
            Elem[D[d],i,8] = Elem[table,index,8];
        else
            if is_vtbl then
                Elem[D[d],i,8] = Zeros(8);
            // else Elem[D[d],i,8] unchanged
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

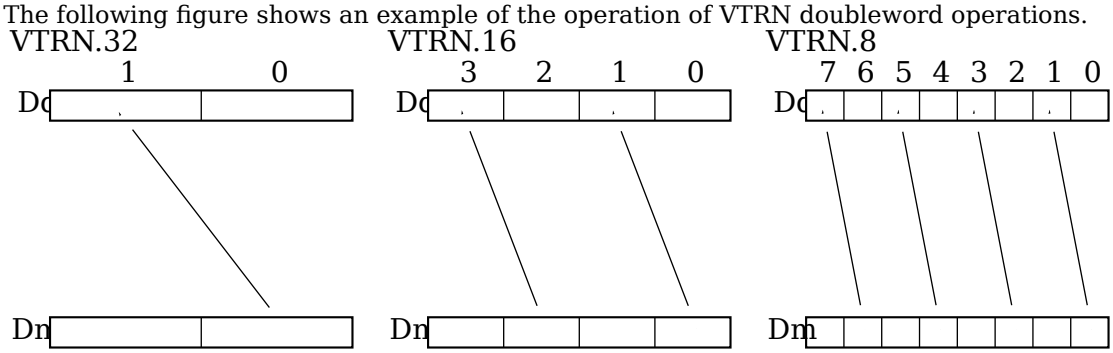
Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VTRN

Vector Transpose treats the elements of its operand vectors as elements of 2 x 2 matrices, and transposes the matrices.

The elements of the vectors can be 8-bit, 16-bit, or 32-bit. There is no distinction between data types.

The following figure shows an example of the operation of VTRN doubleword operations.



Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

This instruction is used by the pseudo-instructions VUZP (alias), and VZIP (alias).

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size		1	0	Vd				0	0	0	0	1	Q	M	0	Vm			

64-bit SIMD vector (Q == 0)

```
VTRN{<c>}{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VTRN{<c>}{<q>}.<dt> <Qd>, <Qm>
```

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size		1	0	Vd				0	0	0	0	1	Q	M	0	Vm			

64-bit SIMD vector (Q == 0)

```
VTRN{<c>}{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VTRN{<c>}{<q>}.<dt> <Qd>, <Qm>
```

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the vectors, encoded in “size”:

size	<dt>
00	8
01	16
10	32
11	RESERVED

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    h = elements DIV 2;

    for r = 0 to regs-1
        if d == m then
            D[d+r] = bits(64) UNKNOWN;
        else
            for e = 0 to h-1
                Elem[D[d+r],2*e+1,esize] = Elem[Din[m+r],2*e,esize];
                Elem[D[m+r],2*e,esize] = Elem[Din[d+r],2*e+1,esize];
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VTST

Vector Test Bits takes each element in a vector, and bitwise ANDs it with the corresponding element of a second vector. If the result is not zero, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

The operand vector elements can be any one of:

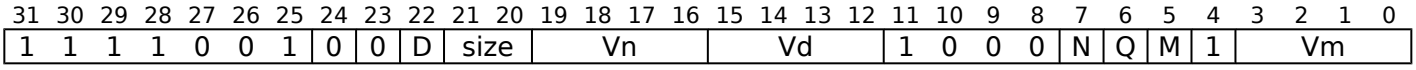
- 8-bit, 16-bit, or 32-bit fields.

The result vector elements are fields the same size as the operand vector elements.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



64-bit SIMD vector (Q == 0)

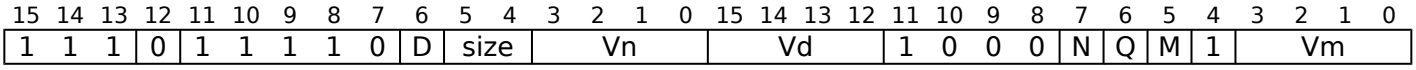
```
VTST{<c>}{<q>}.<dt> {<Dd>,} <Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VTST{<c>}{<q>}.<dt> {<Qd>,} <Qn>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

T1



64-bit SIMD vector (Q == 0)

```
VTST{<c>}{<q>}.<dt> {<Dd>,} <Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VTST{<c>}{<q>}.<dt> {<Qd>,} <Qn>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;
d = UInt(D:Vd);  n = UInt(N:Vn);  m = UInt(M:Vm);  regs = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c>	For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional. For encoding T1: see *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<dt>	Is the data type for the elements of the operands, encoded in “size”:

size	<dt>
00	8
01	16
10	32

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            if !IsZero(Elem[D[n+r],e,esize] AND Elem[D[m+r],e,esize]) then
                Elem[D[d+r],e,esize] = Ones(esize);
            else
                Elem[D[d+r],e,esize] = Zeros(esize);
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VUDOT (vector)

Dot Product vector form with unsigned integers. This instruction performs the dot product of the four 8-bit elements in each 32-bit element of the first source register with the four 8-bit elements of the corresponding 32-bit element in the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

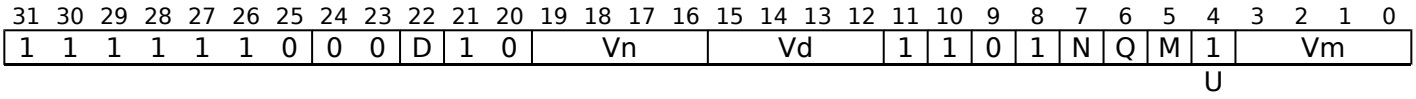
In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

ID_ISAR6.DP indicates whether this instruction is supported.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

(FEAT_DotProd)



64-bit SIMD vector (Q == 0)

```
VUDOT{<q>}.U8 <Dd>, <Dn>, <Dm>
```

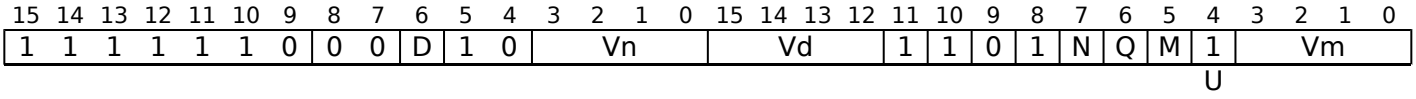
128-bit SIMD vector (Q == 1)

```
VUDOT{<q>}.U8 <Qd>, <Qn>, <Qm>
```

```
if !HaveDOTPExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
boolean signed = U=='0';
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(M:Vm);
integer esize = 32;
integer regs = if Q == '1' then 2 else 1;
```

T1

(FEAT_DotProd)



64-bit SIMD vector (Q == 0)

```
VUDOT{<q>}.U8 <Dd>, <Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VUDOT{<q>}.U8 <Qd>, <Qn>, <Qm>
```

```
if InITBlock() then UNPREDICTABLE;
if !HaveDOTPExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
boolean signed = U=='0';
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(M:Vm);
integer esize = 32;
integer regs = if Q == '1' then 2 else 1;
```

Assembler Symbols

<q>	See *Standard assembler syntax fields*.
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qn>	Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
<Qm>	Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
bits(64) operand1;
bits(64) operand2;
bits(64) result;
CheckAdvSIMDEnabled();
for r = 0 to regs-1
    operand1 = D[n+r];
    operand2 = D[m+r];
    result = D[d+r];
    integer element1, element2;
    for e = 0 to 1
        integer res = 0;
        for i = 0 to 3
            if signed then
                element1 = SInt(Elem[operand1, 4 * e + i, esize DIV 4]);
                element2 = SInt(Elem[operand2, 4 * e + i, esize DIV 4]);
            else
                element1 = UInt(Elem[operand1, 4 * e + i, esize DIV 4]);
                element2 = UInt(Elem[operand2, 4 * e + i, esize DIV 4]);
            res = res + element1 * element2;
        Elem[result, e, esize] = Elem[result, e, esize] + res;
    D[d+r] = result;
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VUDOT (by element)

Dot Product index form with unsigned integers. This instruction performs the dot product of the four 8-bit elements in each 32-bit element of the first source register with the four 8-bit elements of an indexed 32-bit element in the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

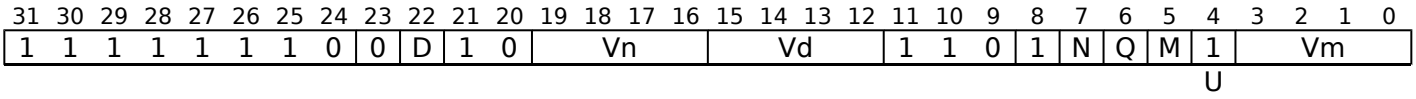
In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

ID_ISAR6.DP indicates whether this instruction is supported.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

(FEAT_DotProd)



64-bit SIMD vector (Q == 0)

```
VUDOT{<q>}.U8 <Dd>, <Dn>, <Dm>[<index>]
```

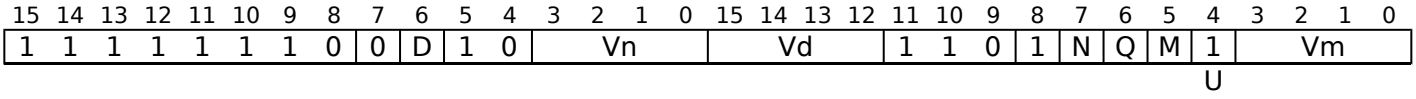
128-bit SIMD vector (Q == 1)

```
VUDOT{<q>}.U8 <Qd>, <Qn>, <Dm>[<index>]
```

```
if !HaveDOTPExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
boolean signed = (U=='0');
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(Vm<3:0>);
integer index = UInt(M);
integer esize = 32;
integer regs = if Q == '1' then 2 else 1;
```

T1

(FEAT_DotProd)



64-bit SIMD vector (Q == 0)

```
VUDOT{<q>}.U8 <Dd>, <Dn>, <Dm>[<index>]
```

128-bit SIMD vector (Q == 1)

```
VUDOT{<q>}.U8 <Qd>, <Qn>, <Dm>[<index>]
```

```
if InITBlock() then UNPREDICTABLE;
if !HaveDOTPExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
boolean signed = (U=='0');
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(Vm<3:0>);
integer index = UInt(M);
integer esize = 32;
integer regs = if Q == '1' then 2 else 1;
```

Assembler Symbols

<q>	See *Standard assembler syntax fields*.
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qn>	Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "Vm" field.
<index>	Is the element index in the range 0 to 1, encoded in the "M" field.

Operation

```
bits(64) operand1;
bits(64) operand2 = D[m];
bits(64) result;
CheckAdvSIMDEnabled();
for r = 0 to regs-1
    operand1 = D[n+r];
    result = D[d+r];
    integer element1, element2;
    for e = 0 to 1
        integer res = 0;
        for i = 0 to 3
            if signed then
                element1 = SInt(Elem[operand1, 4 * e + i, esize DIV 4]);
                element2 = SInt(Elem[operand2, 4 * index + i, esize DIV 4]);
            else
                element1 = UInt(Elem[operand1, 4 * e + i, esize DIV 4]);
                element2 = UInt(Elem[operand2, 4 * index + i, esize DIV 4]);
            res = res + element1 * element2;
        Elem[result, e, esize] = Elem[result, e, esize] + res;
    D[d+r] = result;
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VUMMLA

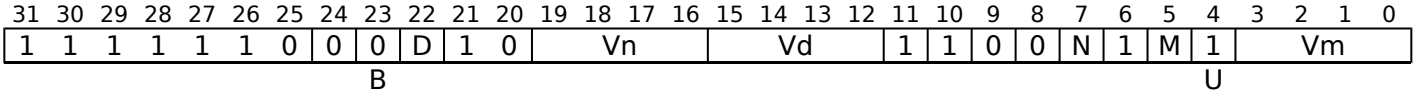
The widening integer matrix multiply-accumulate instruction multiplies the 2x8 matrix of unsigned 8-bit integer values held in the first source vector by the 8x2 matrix of unsigned 8-bit integer values in the second source vector. The resulting 2x2 32-bit integer matrix product is destructively added to the 32-bit integer matrix accumulator held in the destination vector. This is equivalent to performing an 8-way dot product per destination element.

From Armv8.2, this is an OPTIONAL instruction. *ID_ISAR6*.I8MM indicates whether this instruction is supported in the T32 and A32 instruction sets.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

(FEAT_AA32I8MM)



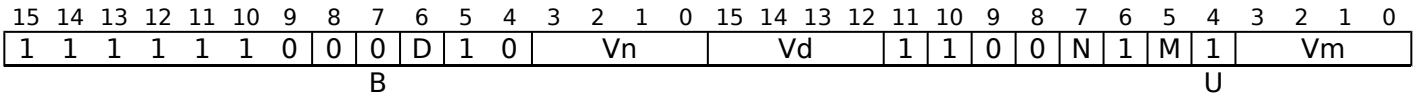
A1

```
VUMMLA{<q>}.U8 <Qd>, <Qn>, <Qm>
```

```
if !HaveAArch32Int8MatMulExt() then UNDEFINED;
case B:U of
    when '00' op1_unsigned = FALSE; op2_unsigned = FALSE;
    when '01' op1_unsigned = TRUE;  op2_unsigned = TRUE;
    when '10' op1_unsigned = TRUE;  op2_unsigned = FALSE;
    when '11' UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(M:Vm);
```

T1

(FEAT_AA32I8MM)



T1

```
VUMMLA{<q>}.U8 <Qd>, <Qn>, <Qm>
```

```
if InITBlock() then UNPREDICTABLE;
if !HaveAArch32Int8MatMulExt() then UNDEFINED;
case B:U of
    when '00' op1_unsigned = FALSE; op2_unsigned = FALSE;
    when '01' op1_unsigned = TRUE;  op2_unsigned = TRUE;
    when '10' op1_unsigned = TRUE;  op2_unsigned = FALSE;
    when '11' UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(M:Vm);
```

Assembler Symbols

<q>	See *Standard assembler syntax fields*.
<Qd>	Is the 128-bit name of the SIMD&FP third source and destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
CheckAdvSIMDEnabled();
bits(128) operand1 = Q[n>>1];
bits(128) operand2 = Q[m>>1];
bits(128) addend   = Q[d>>1];

Q[d>>1] = MatMulAdd(addend, operand1, operand2, op1_unsigned, op2_unsigned);
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VUSDOT (vector)

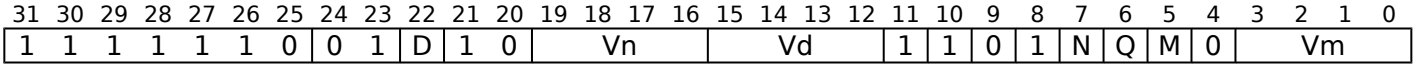
Dot Product vector form with mixed-sign integers. This instruction performs the dot product of the four unsigned 8-bit integer values in each 32-bit element of the first source register with the four signed 8-bit integer values in the corresponding 32-bit element of the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

From Armv8.2, this is an OPTIONAL instruction. *ID_ISAR6*.I8MM indicates whether this instruction is supported in the T32 and A32 instruction sets.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

(FEAT_AA32I8MM)



64-bit SIMD vector (Q == 0)

```
VUSDOT{<q>}.S8 <Dd>, <Dn>, <Dm>
```

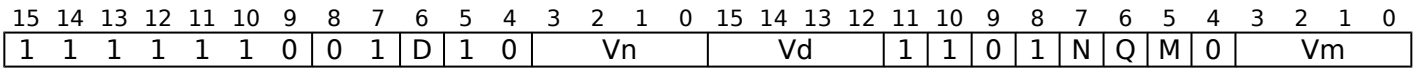
128-bit SIMD vector (Q == 1)

```
VUSDOT{<q>}.S8 <Qd>, <Qn>, <Qm>
```

```
if !HaveAArch32Int8MatMulExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(M:Vm);
integer regs = if Q == '1' then 2 else 1;
```

T1

(FEAT_AA32I8MM)



64-bit SIMD vector (Q == 0)

```
VUSDOT{<q>}.S8 <Dd>, <Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VUSDOT{<q>}.S8 <Qd>, <Qn>, <Qm>
```

```
if InITBlock() then UNPREDICTABLE;
if !HaveAArch32Int8MatMulExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(M:Vm);
integer regs = if Q == '1' then 2 else 1;
```

Assembler Symbols

<q>	See *Standard assembler syntax fields*.
<Qd>	Is the 128-bit name of the SIMD&FP third source and destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP third source and destination register, encoded in the "D:Vd" field.

<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
CheckAdvSIMDEnabled();
bits(64) operand1;
bits(64) operand2;
bits(64) result;

for r = 0 to regs-1
    operand1 = Din[n+r];
    operand2 = Din[m+r];
    result = Din[d+r];
    for e = 0 to 1
        bits(32) res = Elem[result, e, 32];
        for b = 0 to 3
            element1 = UInt(Elem[operand1, 4 * e + b, 8]);
            element2 = SInt(Elem[operand2, 4 * e + b, 8]);
            res = res + element1 * element2;
        Elem[result, e, 32] = res;
    D[d+r] = result;
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VUSDOT (by element)

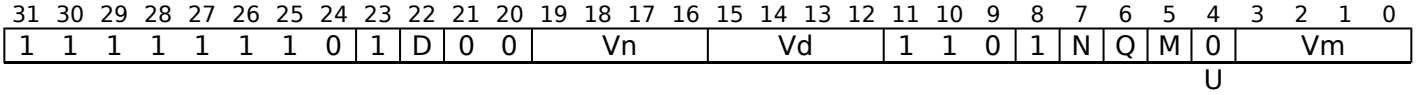
Dot Product index form with unsigned and signed integers. This instruction performs the dot product of the four unsigned 8-bit integer values in each 32-bit element of the first source register with the four signed 8-bit integer values in an indexed 32-bit element of the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

From Armv8.2, this is an OPTIONAL instruction. *ID_ISAR6*.I8MM indicates whether this instruction is supported in the T32 and A32 instruction sets.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

(FEAT_AA32I8MM)



64-bit SIMD vector (Q == 0)

```
VUSDOT{<q>}.S8 <Dd>, <Dn>, <Dm>[<index>]
```

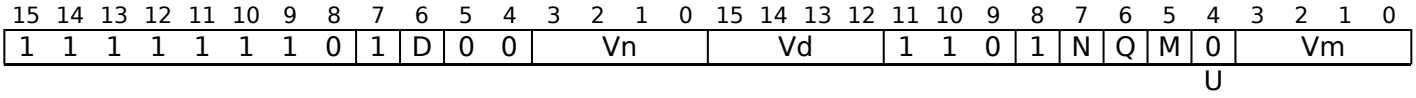
128-bit SIMD vector (Q == 1)

```
VUSDOT{<q>}.S8 <Qd>, <Qn>, <Dm>[<index>]
```

```
if !HaveAArch32Int8MatMulExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
boolean op1_unsigned = (U == '0');
boolean op2_unsigned = (U == '1');
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(Vm);
integer i = UInt(M);
integer regs = if Q == '1' then 2 else 1;
```

T1

(FEAT_AA32I8MM)



64-bit SIMD vector (Q == 0)

```
VUSDOT{<q>}.S8 <Dd>, <Dn>, <Dm>[<index>]
```

128-bit SIMD vector (Q == 1)

```
VUSDOT{<q>}.S8 <Qd>, <Qn>, <Dm>[<index>]
```

```
if InITBlock() then UNPREDICTABLE;
if !HaveAArch32Int8MatMulExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
boolean op1_unsigned = (U == '0');
boolean op2_unsigned = (U == '1');
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(Vm);
integer i = UInt(M);
integer regs = if Q == '1' then 2 else 1;
```

Assembler Symbols

<q>	See *Standard assembler syntax fields*.
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.
<Qn>	Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "Vm" field.
<index>	Is the element index in the range 0 to 1, encoded in the "M" field.

Operation

```
CheckAdvSIMDEnabled();
bits(64) operand1;
bits(64) operand2;
bits(64) result;

operand2 = Din[m];
for r = 0 to regs-1
    operand1 = Din[n+r];
    result = Din[d+r];
    for e = 0 to 1
        bits(32) res = Elem[result, e, 32];
        for b = 0 to 3
            element1 = Int(Elem[operand1, 4 * e + b, 8], op1_unsigned);
            element2 = Int(Elem[operand2, 4 * i + b, 8], op2_unsigned);
            res = res + element1 * element2;
        Elem[result, e, 32] = res;
    D[d+r] = result;
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VUSMMLA

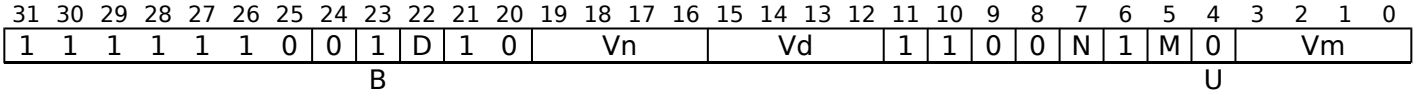
The widening integer matrix multiply-accumulate instruction multiplies the 2x8 matrix of unsigned 8-bit integer values held in the first source vector by the 8x2 matrix of signed 8-bit integer values in the second source vector. The resulting 2x2 32-bit integer matrix product is destructively added to the 32-bit integer matrix accumulator held in the destination vector. This is equivalent to performing an 8-way dot product per destination element.

From Armv8.2, this is an OPTIONAL instruction. *ID_ISAR6*.I8MM indicates whether this instruction is supported in the T32 and A32 instruction sets.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

(FEAT_AA32I8MM)



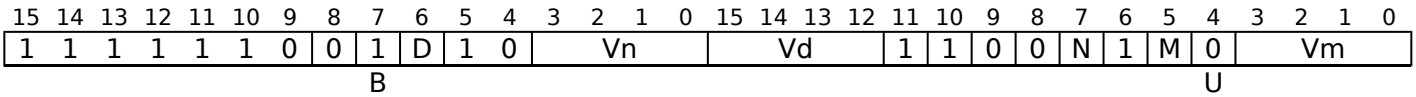
A1

```
VUSMMLA{<q>}.S8 <Qd>, <Qn>, <Qm>
```

```
if !HaveAArch32Int8MatMulExt() then UNDEFINED;
case B:U of
    when '00' op1_unsigned = FALSE; op2_unsigned = FALSE;
    when '01' op1_unsigned = TRUE;  op2_unsigned = TRUE;
    when '10' op1_unsigned = TRUE;  op2_unsigned = FALSE;
    when '11' UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(M:Vm);
```

T1

(FEAT_AA32I8MM)



T1

```
VUSMMLA{<q>}.S8 <Qd>, <Qn>, <Qm>
```

```
if InITBlock() then UNPREDICTABLE;
if !HaveAArch32Int8MatMulExt() then UNDEFINED;
case B:U of
    when '00' op1_unsigned = FALSE; op2_unsigned = FALSE;
    when '01' op1_unsigned = TRUE;  op2_unsigned = TRUE;
    when '10' op1_unsigned = TRUE;  op2_unsigned = FALSE;
    when '11' UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(M:Vm);
```

Assembler Symbols

<q>	See *Standard assembler syntax fields*.
<Qd>	Is the 128-bit name of the SIMD&FP third source and destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qn> Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.

<Qm> Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

Operation

```
CheckAdvSIMDEnabled();
bits(128) operand1 = Q[n>>1];
bits(128) operand2 = Q[m>>1];
bits(128) addend   = Q[d>>1];

Q[d>>1] = MatMulAdd(addend, operand1, operand2, op1_unsigned, op2_unsigned);
```

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VUZP

Vector Unzip de-interleaves the elements of two vectors.
The elements of the vectors can be 8-bit, 16-bit, or 32-bit. There is no distinction between data types.
The following figure shows an example of the operation of VUZP doubleword operation for data type 8.

VUZP.8, doubleword

	Register state before opera								Register state after operation							
Dd	A7	A6	A5	A4	A3	A2	A1	A0	B6	B4	B2	B0	A6	A4	A2	A0
Dm	B7	B6	B5	B4	B3	B2	B1	B0	B7	B5	B3	B1	A7	A5	A3	A1

The following figure shows an example of the operation of VUZP quadword operation for data type 32.

VUZP.32, quadword

	Register state before opera				Register state after operation			
Qd	A3	A2	A1	A0	B2	B0	A2	A0
Qm	B3	B2	B1	B0	B3	B1	A3	A1

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size		1	0	Vd				0	0	0	1	0	Q	M	0	Vm			

64-bit SIMD vector (Q == 0)

```
VUZP{<c>}{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VUZP{<c>}{<q>}.<dt> <Qd>, <Qm>

if size == '11' || (Q == '0' && size == '10') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
quadword_operation = (Q == '1');  esize = 8 << UInt(size);
d = UInt(D:Vd);  m = UInt(M:Vm);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size		1	0	Vd				0	0	0	1	0	Q	M	0	Vm			

64-bit SIMD vector (Q == 0)

```
VUZP{<c>}{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VUZP{<c>}{<q>}.<dt> <Qd>, <Qm>

if size == '11' || (Q == '0' && size == '10') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
quadword_operation = (Q == '1');  esize = 8 << UInt(size);
d = UInt(D:Vd);  m = UInt(M:Vm);
```

Assembler Symbols

<c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<dt> For the 64-bit SIMD vector variant: is the data type for the elements of the vectors, encoded in “size”:

size	<dt>
00	8
01	16
1x	RESERVED

For the 128-bit SIMD vector variant: is the data type for the elements of the vectors, encoded in “size”:

size	<dt>
00	8
01	16
10	32
11	RESERVED

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    if quadword_operation then
        if d == m then
            Q[d>>1] = bits(128) UNKNOWN;
        else
            zipped_q = Q[m>>1]:Q[d>>1];
            for e = 0 to (128 DIV esize) - 1
                Elem[Q[d>>1],e,esize] = Elem[zipped_q,2*e,esize];
                Elem[Q[m>>1],e,esize] = Elem[zipped_q,2*e+1,esize];
    else
        if d == m then
            D[d] = bits(64) UNKNOWN;
        else
            zipped_d = D[m]:D[d];
            for e = 0 to (64 DIV esize) - 1
                Elem[D[d],e,esize] = Elem[zipped_d,2*e,esize];
                Elem[D[m],e,esize] = Elem[zipped_d,2*e+1,esize];
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VUZP (alias)

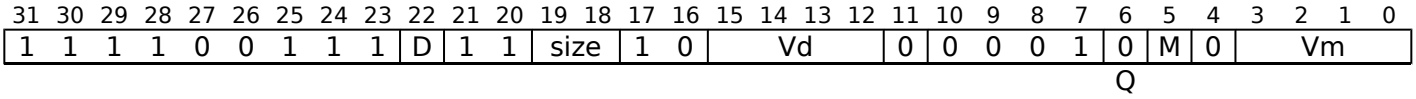
Vector Unzip de-interleaves the elements of two vectors

This is a pseudo-instruction of VTRN. This means:

- The encodings in this description are named to match the encodings of VTRN.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of VTRN gives the operational pseudocode for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



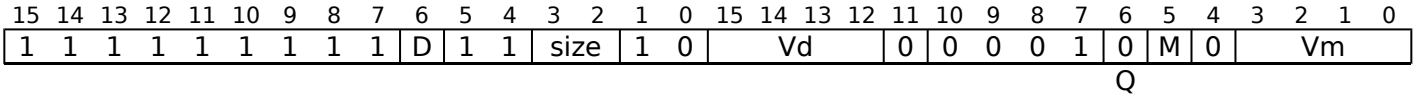
64-bit SIMD vector

```
VUZP{<c>}{<q>}.32 <Dd>, <Dm>
```

is equivalent to

```
VTRN{<c>}{<q>}.32 <Dd>, <Dm>
```

T1



64-bit SIMD vector

```
VUZP{<c>}{<q>}.32 <Dd>, <Dm>
```

is equivalent to

```
VTRN{<c>}{<q>}.32 <Dd>, <Dm>
```

Assembler Symbols

<c>	For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional. For encoding T1: see *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dm>	Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

The description of VTRN gives the operational pseudocode for this instruction.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VZIP

Vector Zip interleaves the elements of two vectors.

The elements of the vectors can be 8-bit, 16-bit, or 32-bit. There is no distinction between data types.

The following figure shows an example of the operation of VZIP doubleword operation for data type 8.

VZIP.8, doubleword

	Register state before opera								Register state after operation							
Dd	A7	A6	A5	A4	A3	A2	A1	A0	B3	A3	B2	A2	B1	A1	B0	A0
Dm	B7	B6	B5	B4	B3	B2	B1	B0	B7	A7	B6	A6	B5	A5	B4	A4

The following figure shows an example of the operation of VZIP quadword operation for data type 32.

VZIP.32, quadword

	Register state before opera				Register state after operation			
Qd	A3	A2	A1	A0	B1	A1	B0	A0
Qm	B3	B2	B1	B0	B3	A3	B2	A2

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size		1	0	Vd				0	0	0	1	1	Q	M	0	Vm			

64-bit SIMD vector (Q == 0)

```
VZIP{<c>}{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VZIP{<c>}{<q>}.<dt> <Qd>, <Qm>
```

```
if size == '11' || (Q == '0' && size == '10') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
quadword_operation = (Q == '1');  esize = 8 << UInt(size);
d = UInt(D:Vd);  m = UInt(M:Vm);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	D	1	1	size		1	0	Vd				0	0	0	1	1	Q	M	0	Vm			

64-bit SIMD vector (Q == 0)

```
VZIP{<c>}{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VZIP{<c>}{<q>}.<dt> <Qd>, <Qm>
```

```
if size == '11' || (Q == '0' && size == '10') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
quadword_operation = (Q == '1');  esize = 8 << UInt(size);
d = UInt(D:Vd);  m = UInt(M:Vm);
```

Assembler Symbols

<c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<dt> For the 64-bit SIMD vector variant: is the data type for the elements of the vectors, encoded in “size”:

size	<dt>
00	8
01	16
1x	RESERVED

For the 128-bit SIMD vector variant: is the data type for the elements of the vectors, encoded in “size”:

size	<dt>
00	8
01	16
10	32
11	RESERVED

<Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.

<Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    if quadword_operation then
        if d == m then
            Q[d>>1] = bits(128) UNKNOWN;
        else
            bits(256) zipped_q;
            for e = 0 to (128 DIV esize) - 1
                Elem[zipped_q,2*e,esize] = Elem[Q[d>>1],e,esize];
                Elem[zipped_q,2*e+1,esize] = Elem[Q[m>>1],e,esize];
            Q[d>>1] = zipped_q<127:0>;  Q[m>>1] = zipped_q<255:128>;
    else
        if d == m then
            D[d] = bits(64) UNKNOWN;
        else
            bits(128) zipped_d;
            for e = 0 to (64 DIV esize) - 1
                Elem[zipped_d,2*e,esize] = Elem[D[d],e,esize];
                Elem[zipped_d,2*e+1,esize] = Elem[D[m],e,esize];
            D[d] = zipped_d<63:0>;  D[m] = zipped_d<127:64>;
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VZIP (alias)

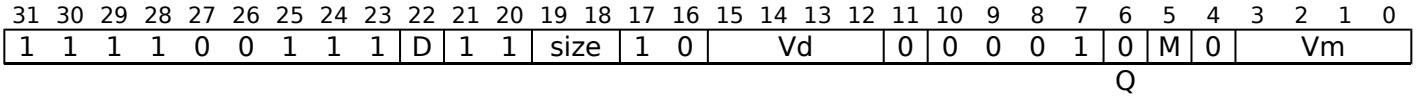
Vector Zip interleaves the elements of two vectors

This is a pseudo-instruction of VTRN. This means:

- The encodings in this description are named to match the encodings of VTRN.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of VTRN gives the operational pseudocode for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1



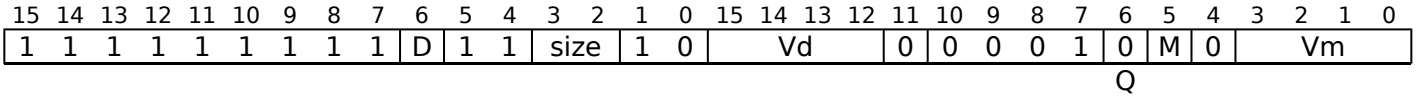
64-bit SIMD vector

```
VZIP{<c>}{<q>}.32 <Dd>, <Dm>
```

is equivalent to

```
VTRN{<c>}{<q>}.32 <Dd>, <Dm>
```

T1



64-bit SIMD vector

```
VZIP{<c>}{<q>}.32 <Dd>, <Dm>
```

is equivalent to

```
VTRN{<c>}{<q>}.32 <Dd>, <Dm>
```

Assembler Symbols

<c>	For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional. For encoding T1: see *Standard assembler syntax fields*.
<q>	See *Standard assembler syntax fields*.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dm>	Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

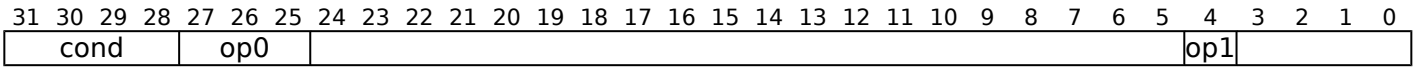
Operation

The description of VTRN gives the operational pseudocode for this instruction.

Internal version only: isa v01_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

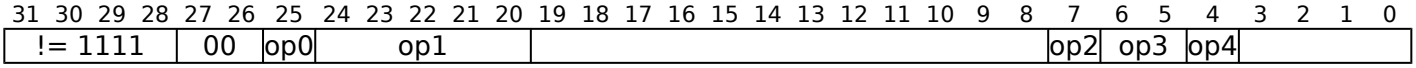
Top-level encodings for A32



Decode fields cond	op0	op1	Instruction details
!= 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

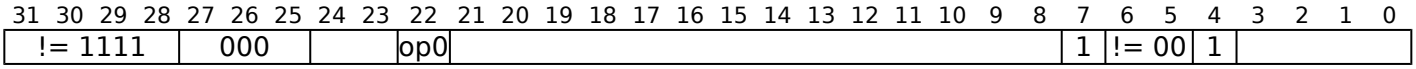
These instructions are under the top-level.



Decode fields op0	op1	op2	op3	op4	Instruction details
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

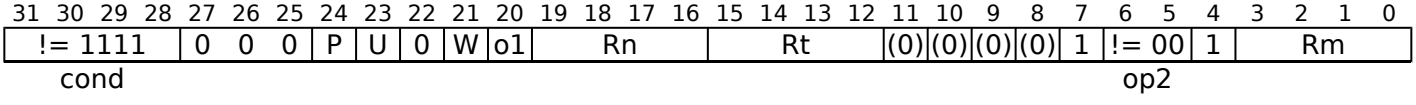
These instructions are under Data-processing and miscellaneous instructions.



Decode fields op0	Instruction details
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.

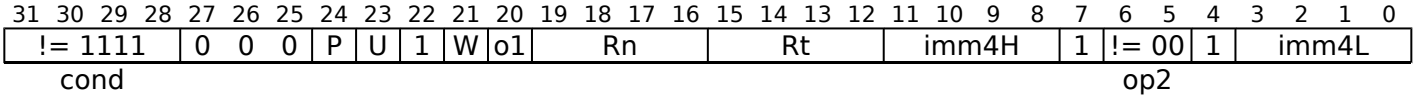


The following constraints also apply to this encoding: cond != 1111 && op2 != 00 && cond != 1111 && op2 != 00

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.



The following constraints also apply to this encoding: cond != 1111 && op2 != 00 && cond != 1111 && op2 != 00

Decode fields				Instruction Details
P:W	o1	Rn	op2	
	0	1111	10	LDRD (literal)
!= 01	1	1111	01	LDRH (literal)
!= 01	1	1111	10	LDRSB (literal)
!= 01	1	1111	11	LDRSH (literal)
00	0	!= 1111	10	LDRD (immediate) — post-indexed
00	0		01	STRH (immediate) — post-indexed
00	0		11	STRD (immediate) — post-indexed
00	1	!= 1111	01	LDRH (immediate) — post-indexed
00	1	!= 1111	10	LDRSB (immediate) — post-indexed
00	1	!= 1111	11	LDRSH (immediate) — post-indexed
01	0	!= 1111	10	UNALLOCATED
01	0		01	STRHT
01	0		11	UNALLOCATED
01	1		01	LDRHT
01	1		10	LDRSBT
01	1		11	LDRSHT
10	0	!= 1111	10	LDRD (immediate) — offset
10	0		01	STRH (immediate) — offset

Decode fields P:W	o1	Rn	op2	Instruction Details
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 opc	S	Instruction Details
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

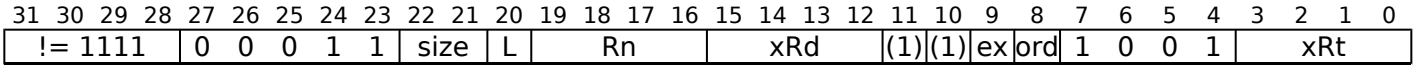
These instructions are under Data-processing and miscellaneous instructions.

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 op0	Instruction details
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.

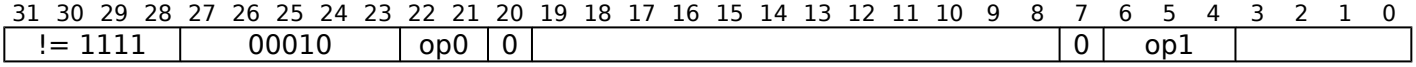


The following constraints also apply to this encoding: cond != 1111 && cond != 1111

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.

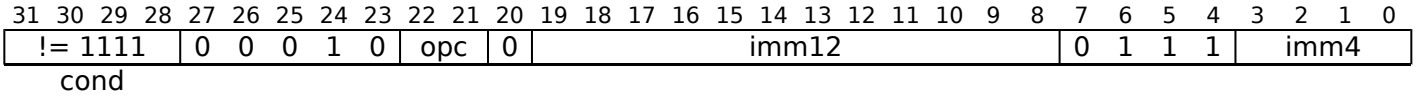


Decode fields		Instruction details
op0	op1	
00	001	UNALLOCATED
00	010	UNALLOCATED
00	011	UNALLOCATED
00	110	UNALLOCATED

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.

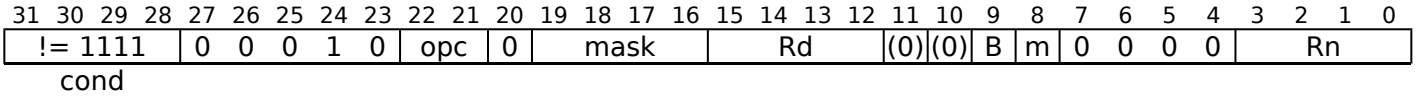


The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields opc	Instruction Details
00	HLT
01	BKPT
10	HVC
11	SMC

Move special register (register)

These instructions are under Miscellaneous.

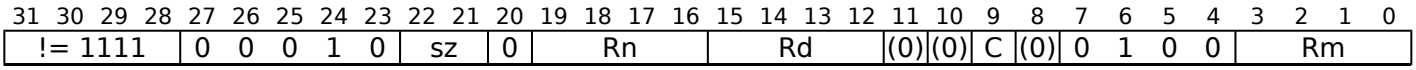


The following constraints also apply to this encoding: cond != 1111 && cond != 1111

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.



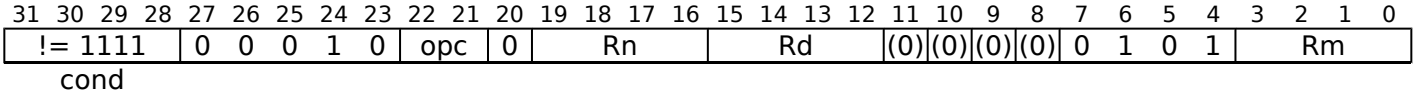
The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields sz	C	Instruction Details
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.

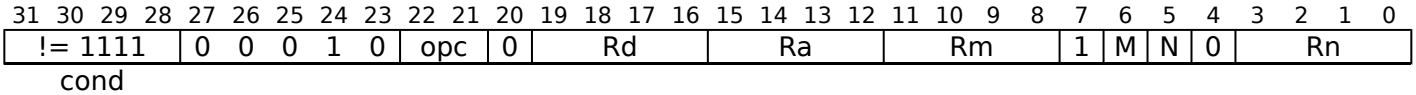


The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields opc	Instruction Details
00	QADD
01	QSUB
10	QDADD
11	QDSUB

Halfword Multiply and Accumulate

These instructions are under Data-processing and miscellaneous instructions.



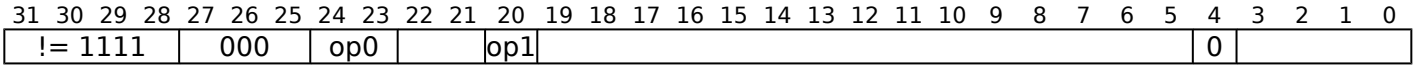
The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields opc	M	N	Instruction Details
00			SMLABB, SMLABT, SMLATB, SMLATT
01	0	0	SMLAWB, SMLAWT — SMLAWB
01	0	1	SMULWB, SMULWT — SMULWB

Decode fields opc	M	N	Instruction Details
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.

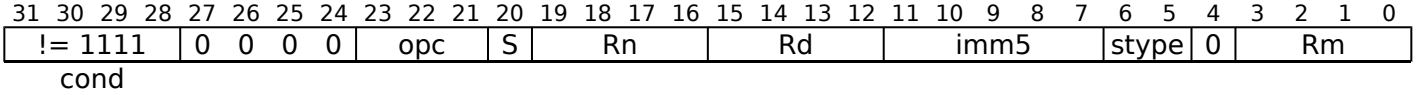


The following constraints also apply to this encoding: op0:op1 != 100

Decode fields op0	op1	Instruction details
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).



The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields opc	S	Rn	imm5:stype	Instruction Details
000			!= 0000011	AND, ANDS (register) — shift or rotate by value
000			0000011	AND, ANDS (register) — rotate right with extend
001			!= 0000011	EOR, EORS (register) — shift or rotate by value
001			0000011	EOR, EORS (register) — rotate right with extend
010	0	!= 1101	!= 0000011	SUB, SUBS (register) — SUB, shift or rotate by value
010	0	!= 1101	0000011	SUB, SUBS (register) — SUB, rotate right with extend
010	0	1101	!= 0000011	SUB, SUBS (SP minus register) — SUB, shift or rotate by value
010	0	1101	0000011	SUB, SUBS (SP minus register) — SUB, rotate right with extend
010	1	!= 1101	!= 0000011	SUB, SUBS (register) — SUBS, shift or rotate by value
010	1	!= 1101	0000011	SUB, SUBS (register) — SUBS, rotate right with extend
010	1	1101	!= 0000011	SUB, SUBS (SP minus register) — SUBS, shift or rotate by value
010	1	1101	0000011	SUB, SUBS (SP minus register) — SUBS, rotate right with extend
011			!= 0000011	RSB, RSBS (register) — shift or rotate by value
011			0000011	RSB, RSBS (register) — rotate right with extend
100	0	!= 1101	!= 0000011	ADD, ADDS (register) — ADD, shift or rotate by value
100	0	!= 1101	0000011	ADD, ADDS (register) — ADD, rotate right with extend
100	0	1101	!= 0000011	ADD, ADDS (SP plus register) — ADD, shift or rotate by value
100	0	1101	0000011	ADD, ADDS (SP plus register) — ADD, rotate right with extend

Decode fields				Instruction Details
opc	S	Rn	imm5:stype	
100	1	!= 1101	!= 0000011	ADD, ADDS (register) — ADDS, shift or rotate by value
100	1	!= 1101	0000011	ADD, ADDS (register) — ADDS, rotate right with extend
100	1	1101	!= 0000011	ADD, ADDS (SP plus register) — ADDS, shift or rotate by value
100	1	1101	0000011	ADD, ADDS (SP plus register) — ADDS, rotate right with extend
101			!= 0000011	ADC, ADCS (register) — shift or rotate by value
101			0000011	ADC, ADCS (register) — rotate right with extend
110			!= 0000011	SBC, SBCS (register) — shift or rotate by value
110			0000011	SBC, SBCS (register) — rotate right with extend
111			!= 0000011	RSC, RSCS (register) — shift or rotate by value
111			0000011	RSC, RSCS (register) — rotate right with extend

Integer Test and Compare (two register, immediate shift)

These instructions are under Data-processing register (immediate shift).

31 30 29 28	27 26 25 24 23	22 21	20	19 18 17 16	15 14 13 12	11 10 9 8 7	6 5	4	3 2 1 0
!= 1111	0 0 0 1 0	opc	1	Rn	(0)(0)(0)(0)	imm5	stype	0	Rm
cond									

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields		Instruction Details
opc	imm5:stype	
00	!= 0000011	TST (register) — shift or rotate by value
00	0000011	TST (register) — rotate right with extend
01	!= 0000011	TEQ (register) — shift or rotate by value
01	0000011	TEQ (register) — rotate right with extend
10	!= 0000011	CMP (register) — shift or rotate by value
10	0000011	CMP (register) — rotate right with extend
11	!= 0000011	CMN (register) — shift or rotate by value
11	0000011	CMN (register) — rotate right with extend

Logical Arithmetic (three register, immediate shift)

These instructions are under Data-processing register (immediate shift).

31 30 29 28	27 26 25 24 23	22 21	20	19 18 17 16	15 14 13 12	11 10 9 8 7	6 5	4	3 2 1 0
!= 1111	0 0 0 1 1	opc	S	Rn	Rd	imm5	stype	0	Rm
cond									

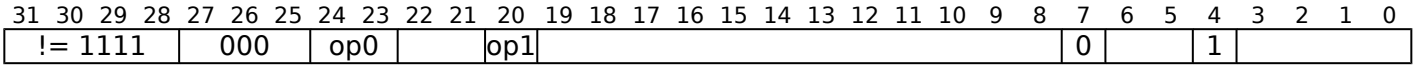
The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields		Instruction Details
opc	imm5:stype	
00	!= 0000011	ORR, ORRS (register) — shift or rotate by value
00	0000011	ORR, ORRS (register) — rotate right with extend
01	!= 0000011	MOV, MOVS (register) — shift or rotate by value
01	0000011	MOV, MOVS (register) — rotate right with extend
10	!= 0000011	BIC, BICS (register) — shift or rotate by value
10	0000011	BIC, BICS (register) — rotate right with extend
11	!= 0000011	MVN, MVNS (register) — shift or rotate by value

Decode fields opc	imm5:stype	Instruction Details
11	0000011	MVN, MVNS (register) — rotate right with extend

Data-processing register (register shift)

These instructions are under Data-processing and miscellaneous instructions.

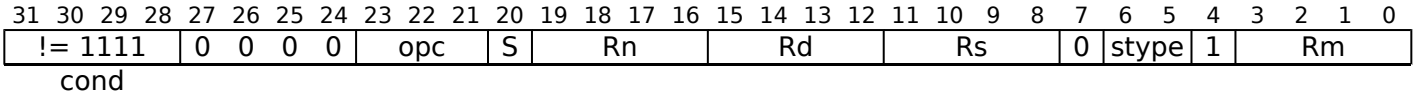


The following constraints also apply to this encoding: op0:op1 != 100

Decode fields op0	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).

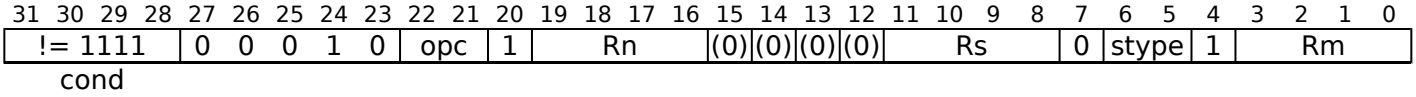


The following constraints also apply to this encoding: cond != 1111 && cond != 1111

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).



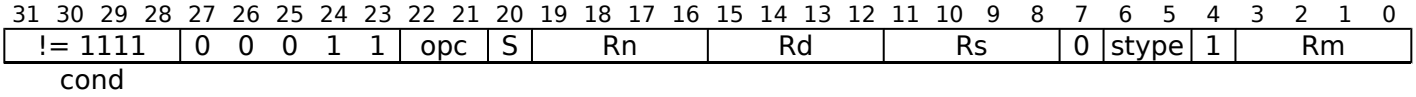
The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields opc	Instruction Details
00	TST (register-shifted register)
01	TEQ (register-shifted register)

Decode fields opc	Instruction Details
10	CMP (register-shifted register)
11	CMN (register-shifted register)

Logical Arithmetic (three register, register shift)

These instructions are under Data-processing register (register shift).

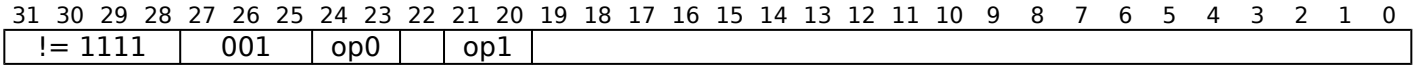


The following constraints also apply to this encoding: cond != 1111 && cond != 1111

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

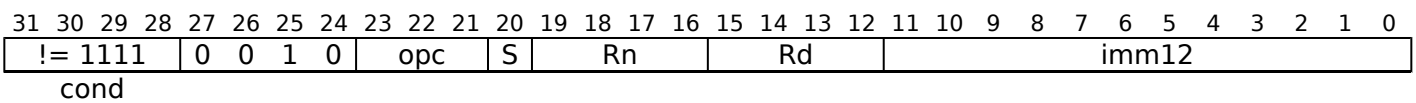
These instructions are under Data-processing and miscellaneous instructions.



Decode fields op0	op1	Instruction details
0x		Integer Data Processing (two register and immediate)
10	00	Move Halfword (immediate)
10	10	Move Special Register and Hints (immediate)
10	x1	Integer Test and Compare (one register and immediate)
11		Logical Arithmetic (two register and immediate)

Integer Data Processing (two register and immediate)

These instructions are under Data-processing immediate.



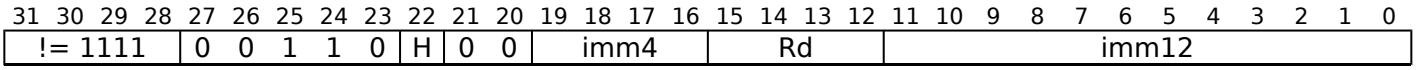
The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields opc	S	Rn	Instruction Details
000			AND, ANDS (immediate)
001			EOR, EORS (immediate)
010	0	!= 11x1	SUB, SUBS (immediate) — SUB
010	0	1101	SUB, SUBS (SP minus immediate) — SUB
010	0	1111	ADR — A2

Decode fields opc	S	Rn	Instruction Details
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.

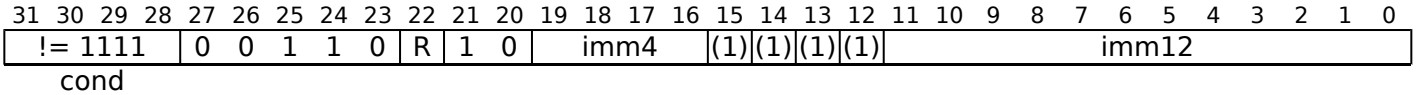


The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields H	Instruction Details
0	MOV, MOVS (immediate)
1	MOVT

Move Special Register and Hints (immediate)

These instructions are under Data-processing immediate.



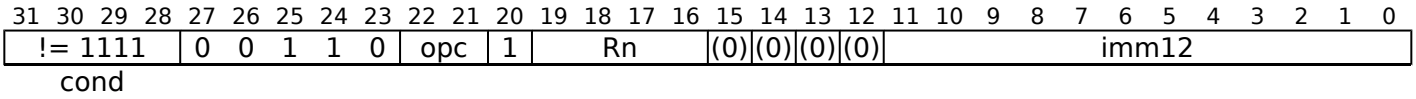
The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields R:imm4	imm12	Instruction Details	Feature
!= 00000		MSR (immediate)	-
00000	xxxx00000000	NOP	-
00000	xxxx00000001	YIELD	-
00000	xxxx00000010	WFE	-
00000	xxxx00000011	WFI	-
00000	xxxx00000100	SEV	-
00000	xxxx00000101	SEVL	-
00000	xxxx0000011x	Reserved hint, behaves as NOP	-
00000	xxxx00001xxx	Reserved hint, behaves as NOP	-
00000	xxxx00010000	ESB	FEAT_RAS
00000	xxxx00010001	Reserved hint, behaves as NOP	-
00000	xxxx00010010	TSB CSYNC	FEAT_TRF

Decode fields R:imm4	imm12	Instruction Details	Feature
00000	xxxx00010011	Reserved hint, behaves as NOP	-
00000	xxxx00010100	CSDB	-
00000	xxxx00010101	Reserved hint, behaves as NOP	-
00000	xxxx0001011x	Reserved hint, behaves as NOP	-
00000	xxxx00011xxx	Reserved hint, behaves as NOP	-
00000	xxxx001xxxxx	Reserved hint, behaves as NOP	-
00000	xxxx01xxxxxx	Reserved hint, behaves as NOP	-
00000	xxxx10xxxxxx	Reserved hint, behaves as NOP	-
00000	xxxx110xxxxx	Reserved hint, behaves as NOP	-
00000	xxxx1110xxxx	Reserved hint, behaves as NOP	-
00000	xxxx1111xxxx	DBG	-

Integer Test and Compare (one register and immediate)

These instructions are under Data-processing immediate.

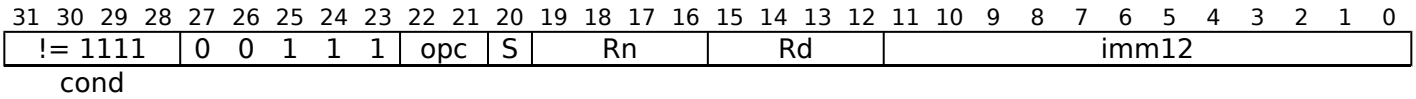


The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields opc	Instruction Details
00	TST (immediate)
01	TEQ (immediate)
10	CMP (immediate)
11	CMN (immediate)

Logical Arithmetic (two register and immediate)

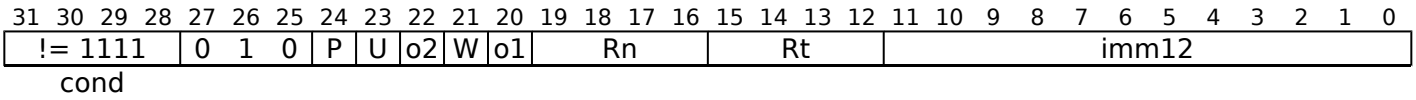
These instructions are under Data-processing immediate.



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)

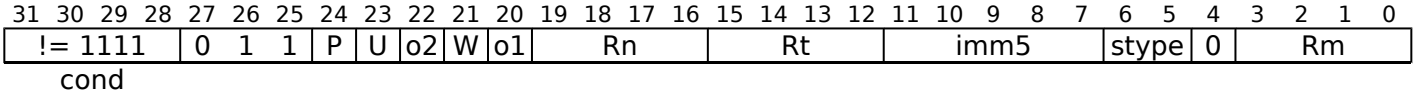
Load/Store Word, Unsigned Byte (immediate, literal)



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)

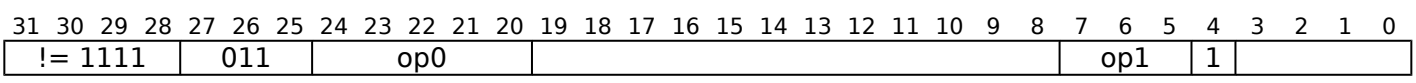


The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields				Instruction Details
P	o2	W	o1	
0	0	0	0	STR (register) — post-indexed
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

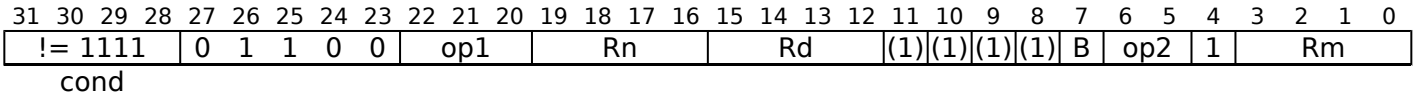
These instructions are under the top-level.



Decode fields op0	op1	Instruction details
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.



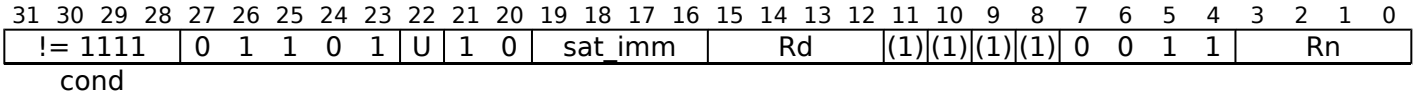
The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields op1	B	op2	Instruction Details
000			UNALLOCATED
001	0	00	SADD16
001	0	01	SASX
001	0	10	SSAX
001	0	11	SSUB16
001	1	00	SADD8
001	1	01	UNALLOCATED

Decode fields			Instruction Details
op1	B	op2	
001	1	10	UNALLOCATED
001	1	11	SSUB8
010	0	00	QADD16
010	0	01	QASX
010	0	10	QSAX
010	0	11	QSUB16
010	1	00	QADD8
010	1	01	UNALLOCATED
010	1	10	UNALLOCATED
010	1	11	QSUB8
011	0	00	SHADD16
011	0	01	SHASX
011	0	10	SHSAX
011	0	11	SHSUB16
011	1	00	SHADD8
011	1	01	UNALLOCATED
011	1	10	UNALLOCATED
011	1	11	SHSUB8
100			UNALLOCATED
101	0	00	UADD16
101	0	01	UASX
101	0	10	USAX
101	0	11	USUB16
101	1	00	UADD8
101	1	01	UNALLOCATED
101	1	10	UNALLOCATED
101	1	11	USUB8
110	0	00	UQADD16
110	0	01	UQASX
110	0	10	UQSAX
110	0	11	UQSUB16
110	1	00	UQADD8
110	1	01	UNALLOCATED
110	1	10	UNALLOCATED
110	1	11	UQSUB8
111	0	00	UHADD16
111	0	01	UHASX
111	0	10	UHSAX
111	0	11	UHSUB16
111	1	00	UHADD8
111	1	01	UNALLOCATED
111	1	10	UNALLOCATED
111	1	11	UHSUB8

Saturate 16-bit

These instructions are under Media instructions.

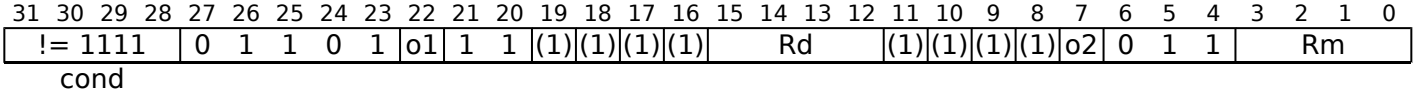


The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields U	Instruction Details
0	SSAT16
1	USAT16

Reverse Bit/Byte

These instructions are under Media instructions.

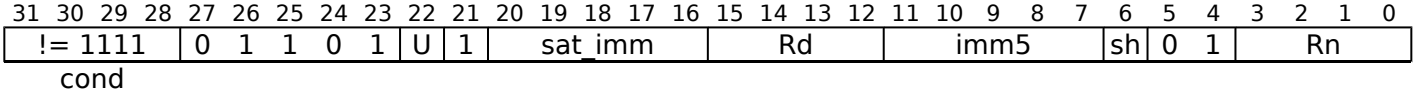


The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields o1	o2	Instruction Details
0	0	REV
0	1	REV16
1	0	RBIT
1	1	REVSH

Saturate 32-bit

These instructions are under Media instructions.

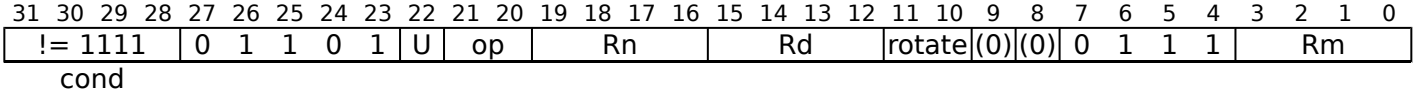


The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields U	Instruction Details
0	SSAT
1	USAT

Extend and Add

These instructions are under Media instructions.



The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields U	op	Rn	Instruction Details
0	00	!= 1111	SXTAB16

Decode fields			Instruction Details
U	op	Rn	
0	00	1111	SXTB16
0	10	!= 1111	SXTAB
0	10	1111	SXTB
0	11	!= 1111	SXTAH
0	11	1111	SXTH
1	00	!= 1111	UXTAB16
1	00	1111	UXTB16
1	10	!= 1111	UXTAB
1	10	1111	UXTB
1	11	!= 1111	UXTAH
1	11	1111	UXTH

Signed multiply, Divide

These instructions are under Media instructions.

31 30 29 28	27 26 25 24 23	22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5	4	3 2 1 0
!= 1111	0 1 1 1 0	op1	Rd	Ra	Rm	op2	1	Rn
cond								

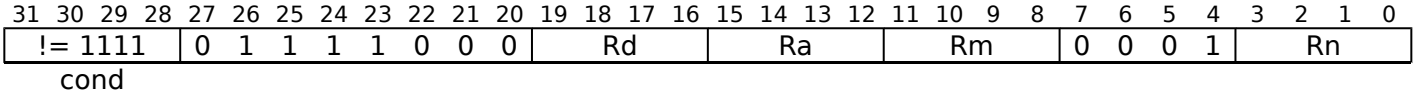
The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields			Instruction Details
op1	Ra	op2	
000	!= 1111	000	SMLAD, SMLADX — SMLAD
000	!= 1111	001	SMLAD, SMLADX — SMLADX
000	!= 1111	010	SMLSD, SMLSDX — SMLSD
000	!= 1111	011	SMLSD, SMLSDX — SMLSDX
000		1xx	UNALLOCATED
000	1111	000	SMUAD, SMUADX — SMUAD
000	1111	001	SMUAD, SMUADX — SMUADX
000	1111	010	SMUSD, SMUSDX — SMUSD
000	1111	011	SMUSD, SMUSDX — SMUSDX
001		000	SDIV
001		!= 000	UNALLOCATED
010			UNALLOCATED
011		000	UDIV
011		!= 000	UNALLOCATED
100		000	SMLALD, SMLALDX — SMLALD
100		001	SMLALD, SMLALDX — SMLALDX
100		010	SMLSLD, SMLSLDX — SMLSLD
100		011	SMLSLD, SMLSLDX — SMLSLDX
100		1xx	UNALLOCATED
101	!= 1111	000	SMMLA, SMMLAR — SMMLA
101	!= 1111	001	SMMLA, SMMLAR — SMMLAR
101		01x	UNALLOCATED
101		10x	UNALLOCATED
101		110	SMMLS, SMMLSR — SMMLS
101		111	SMMLS, SMMLSR — SMMLSR

Decode fields op1	Ra	op2	Instruction Details
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.

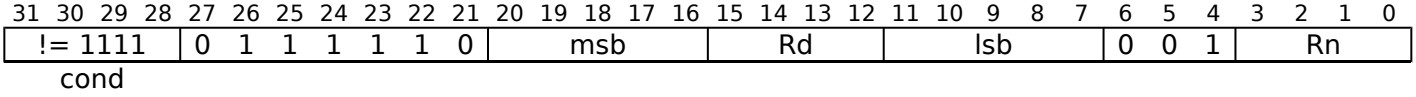


The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields Ra	Instruction Details
!= 1111	USADA8
1111	USAD8

Bitfield Insert

These instructions are under Media instructions.

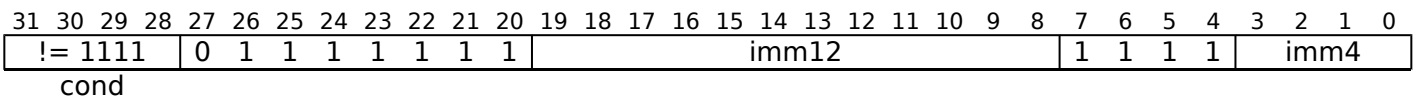


The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields Rn	Instruction Details
!= 1111	BFI
1111	BFC

Permanently UNDEFINED

These instructions are under Media instructions.

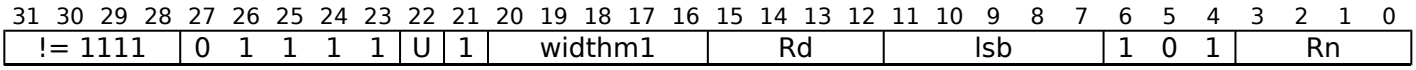


The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields cond	Instruction Details
0xxx	UNALLOCATED
10xx	UNALLOCATED
110x	UNALLOCATED
1110	UDF

Bitfield Extract

These instructions are under Media instructions.

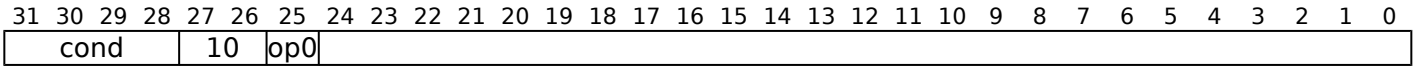


The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields U	Instruction Details
0	SBFX
1	UBFX

Branch, branch with link, and block data transfer

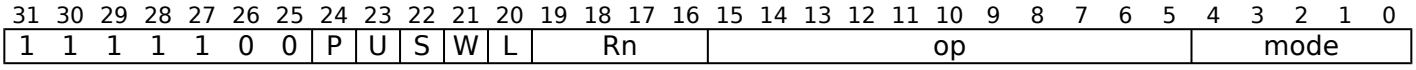
These instructions are under the top-level.



Decode fields cond	Decode fields op0	Instruction details
1111	0	Exception Save/Restore
!= 1111	0	Load/Store Multiple
	1	Branch (immediate)

Exception Save/Restore

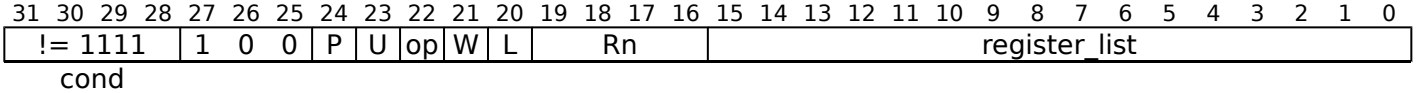
These instructions are under Branch, branch with link, and block data transfer.



Decode fields P	U	S	L	Instruction Details
		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.



The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields					Instruction Details
P	U	op	L	register_list	
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	0xxxxxxxxxxxxxxx	LDM (User registers)
1	1	0	0		STMIB, STMFA
1	1	0	1		LDMIB, LDMED
		1	1	1xxxxxxxxxxxxxxx	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	0x		UNALLOCATED
	10	0x		UNALLOCATED
	11			Supervisor call
1111	!= 11	1x		Unconditional Advanced SIMD and floating-point instructions
!= 1111	0x	1x		Advanced SIMD and System register load/store and 64-bit move
!= 1111	10	1x	1	Advanced SIMD and System register 32-bit move
!= 1111	10	10	0	Floating-point data-processing
!= 1111	10	11	0	UNALLOCATED

Supervisor call

These instructions are under System register access, Advanced SIMD, floating-point, and Supervisor call.

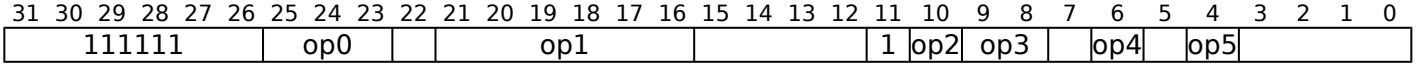
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1111																											

Decode fields	Instruction details

cond	
1111	UNALLOCATED
!= 1111	SVC

Unconditional Advanced SIMD and floating-point instructions

These instructions are under System register access, Advanced SIMD, floating-point, and Supervisor call.

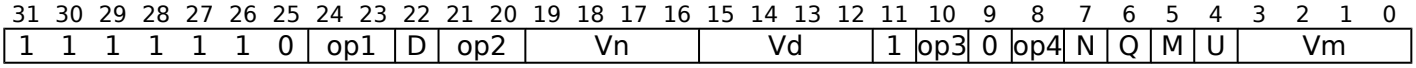


The following constraints also apply to this encoding: op0<2:1> != 11

Decode fields						Instruction details
op0	op1	op2	op3	op4	op5	
0xx			0x			Advanced SIMD three registers of the same length extension
100		0	!= 00	0	0	Floating-point conditional select
101	00xxxx	0	!= 00		0	Floating-point minNum/maxNum
101	110000	0	!= 00	1	0	Floating-point extraction and insertion
101	111xxx	0	!= 00	1	0	Floating-point directed convert to integer
10x		0	00			Advanced SIMD and floating-point multiply with accumulate
10x		1	0x			Advanced SIMD and floating-point dot product

Advanced SIMD three registers of the same length extension

These instructions are under Unconditional Advanced SIMD and floating-point instructions.

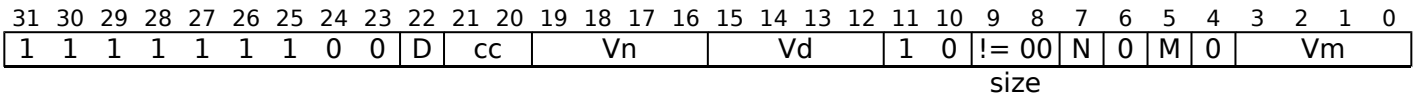


Decode fields						Instruction Details	Feature
op1	op2	op3	op4	Q	U		
x1	0x	0	0	0	0	VCADD — 64-bit SIMD vector	FEAT_FCMA
x1	0x	0	0	0	1	UNALLOCATED	-
x1	0x	0	0	1	0	VCADD — 128-bit SIMD vector	FEAT_FCMA
x1	0x	0	0	1	1	UNALLOCATED	-
00	0x	0	0			UNALLOCATED	-
00	0x	0	1			UNALLOCATED	-
00	00	1	0	0	0	UNALLOCATED	-
00	00	1	0	0	1	UNALLOCATED	-
00	00	1	0	1	0	VMMLA	FEAT_AA32BF16
00	00	1	0	1	1	UNALLOCATED	-
00	00	1	1	0	0	VDOT (vector) — 64-bit SIMD vector	FEAT_AA32BF16
00	00	1	1	0	1	UNALLOCATED	-
00	00	1	1	1	0	VDOT (vector) — 128-bit SIMD vector	FEAT_AA32BF16
00	00	1	1	1	1	UNALLOCATED	-
00	01	1	0			UNALLOCATED	-

Decode fields						Instruction Details	Feature
op1	op2	op3	op4	Q	U		
00	01	1	1			UNALLOCATED	-
00	10	0	0		1	VFMAL (vector)	FEAT_FHM
00	10	0	1			UNALLOCATED	-
00	10	1	0	0		UNALLOCATED	-
00	10	1	0	1	0	VSMMLA	FEAT_AA32I8MM
00	10	1	0	1	1	VUMMLA	FEAT_AA32I8MM
00	10	1	1	0	0	VSDOT (vector) — 64-bit SIMD vector	FEAT_DotProd
00	10	1	1	0	1	VUDOT (vector) — 64-bit SIMD vector	FEAT_DotProd
00	10	1	1	1	0	VSDOT (vector) — 128-bit SIMD vector	FEAT_DotProd
00	10	1	1	1	1	VUDOT (vector) — 128-bit SIMD vector	FEAT_DotProd
00	11	0	0		1	VFMAB, VFMAT (BFloat16, vector)	FEAT_AA32BF16
00	11	0	1			UNALLOCATED	-
00	11	1	0			UNALLOCATED	-
00	11	1	1			UNALLOCATED	-
01	10	0	0		1	VFMSL (vector)	FEAT_FHM
01	10	0	1			UNALLOCATED	-
01	10	1	0	0		UNALLOCATED	-
01	10	1	0	1	0	VUSMMLA	FEAT_AA32I8MM
01	10	1	0	1	1	UNALLOCATED	-
01	10	1	1	0	0	VUSDOT (vector) — 64-bit SIMD vector	FEAT_AA32I8MM
01	10	1	1		1	UNALLOCATED	-
01	10	1	1	1	0	VUSDOT (vector) — 128-bit SIMD vector	FEAT_AA32I8MM
01	11	0	1			UNALLOCATED	-
01	11	1	0			UNALLOCATED	-
01	11	1	1			UNALLOCATED	-
	1x	0	0		0	VCMLA	FEAT_FCMA
10	11	0	1			UNALLOCATED	-
10	11	1	0			UNALLOCATED	-
10	11	1	1			UNALLOCATED	-
11	11	0	1			UNALLOCATED	-
11	11	1	0			UNALLOCATED	-
11	11	1	1			UNALLOCATED	-

Floating-point conditional select

These instructions are under Unconditional Advanced SIMD and floating-point instructions.

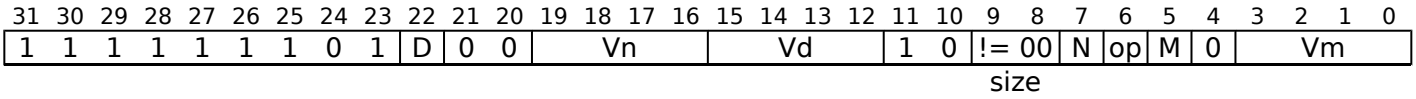


The following constraints also apply to this encoding: size != 00 && size != 00

Decode fields	Instruction Details	Feature
size		
01	VSELEQ, VSELGE, VSELGT, VSELVS — half-precision scalar	FEAT_FP16
10	VSELEQ, VSELGE, VSELGT, VSELVS — single-precision scalar	-
11	VSELEQ, VSELGE, VSELGT, VSELVS — double-precision scalar	-

Floating-point minNum/maxNum

These instructions are under Unconditional Advanced SIMD and floating-point instructions.

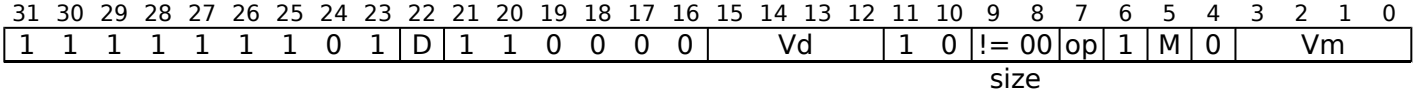


The following constraints also apply to this encoding: size != 00 && size != 00

Decode fields size	op	Instruction Details	Feature
01	0	VMAXNM — half-precision scalar	FEAT_FP16
01	1	VMINNM — half-precision scalar	FEAT_FP16
10	0	VMAXNM — single-precision scalar	-
10	1	VMINNM — single-precision scalar	-
11	0	VMAXNM — double-precision scalar	-
11	1	VMINNM — double-precision scalar	-

Floating-point extraction and insertion

These instructions are under Unconditional Advanced SIMD and floating-point instructions.

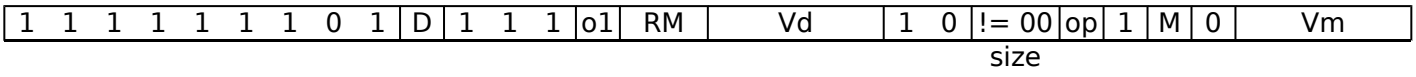


The following constraints also apply to this encoding: size != 00 && size != 00

Decode fields size	op	Instruction Details	Feature
01		UNALLOCATED	-
10	0	VMOVX	FEAT_FP16
10	1	VINS	FEAT_FP16
11		UNALLOCATED	-

Floating-point directed convert to integer

These instructions are under Unconditional Advanced SIMD and floating-point instructions.



The following constraints also apply to this encoding: size != 00 && size != 00

o1	Decode fields RM	size	op	Instruction Details	Feature
0		!= 00	1	UNALLOCATED	-
0	00	01	0	VRINTA (floating-point) — half-precision scalar	FEAT_FP16
0	00	10	0	VRINTA (floating-point) — single-precision scalar	-
0	00	11	0	VRINTA (floating-point) — double-precision scalar	-
0	01	01	0	VRINTN (floating-point) — half-precision scalar	FEAT_FP16
0	01	10	0	VRINTN (floating-point) — single-precision scalar	-
0	01	11	0	VRINTN (floating-point) — double-precision scalar	-

Decode fields				Instruction Details	Feature
o1	RM	size	op		
0	10	01	0	VRINTP (floating-point) — half-precision scalar	FEAT_FP16
0	10	10	0	VRINTP (floating-point) — single-precision scalar	-
0	10	11	0	VRINTP (floating-point) — double-precision scalar	-
0	11	01	0	VRINTM (floating-point) — half-precision scalar	FEAT_FP16
0	11	10	0	VRINTM (floating-point) — single-precision scalar	-
0	11	11	0	VRINTM (floating-point) — double-precision scalar	-
1	00	01		VCVTA (floating-point) — half-precision scalar	FEAT_FP16
1	00	10		VCVTA (floating-point) — single-precision scalar	-
1	00	11		VCVTA (floating-point) — double-precision scalar	-
1	01	01		VCVTN (floating-point) — half-precision scalar	FEAT_FP16
1	01	10		VCVTN (floating-point) — single-precision scalar	-
1	01	11		VCVTN (floating-point) — double-precision scalar	-
1	10	01		VCVTP (floating-point) — half-precision scalar	FEAT_FP16
1	10	10		VCVTP (floating-point) — single-precision scalar	-
1	10	11		VCVTP (floating-point) — double-precision scalar	-
1	11	01		VCVTM (floating-point) — half-precision scalar	FEAT_FP16
1	11	10		VCVTM (floating-point) — single-precision scalar	-
1	11	11		VCVTM (floating-point) — double-precision scalar	-

Advanced SIMD and floating-point multiply with accumulate

These instructions are under Unconditional Advanced SIMD and floating-point instructions.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0	op1	D	op2		Vn				Vd				1	0	0	0	N	Q	M	U	Vm			

Decode fields				Instruction Details	Feature
op1	op2	Q	U		
0			0	VCMLA (by element) — 128-bit SIMD vector of half-precision floating-point	FEAT_FCMA
0	00		1	VFMAL (by scalar)	FEAT_FHM
0	01		1	VFMSL (by scalar)	FEAT_FHM
0	10		1	UNALLOCATED	-
0	11		1	VFMAB, VFMAT (BFloat16, by scalar)	FEAT_AA32BF16
1		0	0	VCMLA (by element) — 64-bit SIMD vector of single-precision floating-point	FEAT_FCMA
1			1	UNALLOCATED	-
1		1	0	VCMLA (by element) — 128-bit SIMD vector of single-precision floating-point	FEAT_FCMA

Advanced SIMD and floating-point dot product

These instructions are under Unconditional Advanced SIMD and floating-point instructions.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0	op1	D	op2		Vn				Vd				1	1	0	op4	N	Q	M	U	Vm			

Decode fields					Instruction Details	Feature
op1	op2	op4	Q	U		
0	00	0			UNALLOCATED	-
0	00	1	0	0	VDOT (by element) — 64-bit SIMD vector	FEAT_AA32BF16

Decode fields op1	op2	op4	Q	U	Instruction Details	Feature
0	00	1		1	UNALLOCATED	-
0	00	1	1	0	VDOT (by element) — 128-bit SIMD vector	FEAT_AA32BF16
0	01	0			UNALLOCATED	-
0	10	0			UNALLOCATED	-
0	10	1	0	0	VSDOT (by element) — 64-bit SIMD vector	FEAT_DotProd
0	10	1	0	1	VUDOT (by element) — 64-bit SIMD vector	FEAT_DotProd
0	10	1	1	0	VSDOT (by element) — 128-bit SIMD vector	FEAT_DotProd
0	10	1	1	1	VUDOT (by element) — 128-bit SIMD vector	FEAT_DotProd
0	11				UNALLOCATED	-
1		0			UNALLOCATED	-
1	00	1	0	0	VUSDOT (by element) — 64-bit SIMD vector	FEAT_AA32I8MM
1	00	1	0	1	VSUDOT (by element) — 64-bit SIMD vector	FEAT_AA32I8MM
1	00	1	1	0	VUSDOT (by element) — 128-bit SIMD vector	FEAT_AA32I8MM
1	00	1	1	1	VSUDOT (by element) — 128-bit SIMD vector	FEAT_AA32I8MM
1	01	1			UNALLOCATED	-
1	1x	1			UNALLOCATED	-

Advanced SIMD and System register load/store and 64-bit move

These instructions are under System register access, Advanced SIMD, floating-point, and Supervisor call.

31 30 29 28	27 26 25	24 23 22 21	20 19 18 17 16 15 14 13 12	11	10 9	8 7 6 5 4 3 2 1 0
!= 1111	110	op0		1	op1	

Decode fields op0	op1	Instruction details
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												

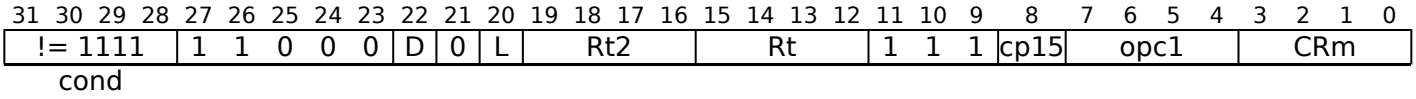
The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields D	op	size	opc2	o3	Instruction Details
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 D	op	size	opc2	o3	Instruction Details
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.

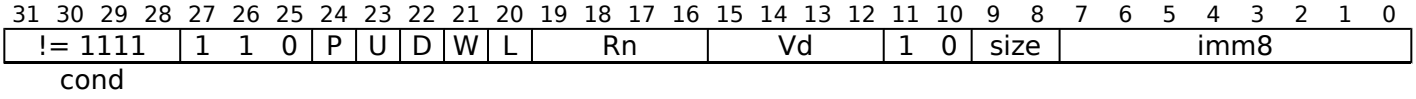


The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields D	L	Instruction Details
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.



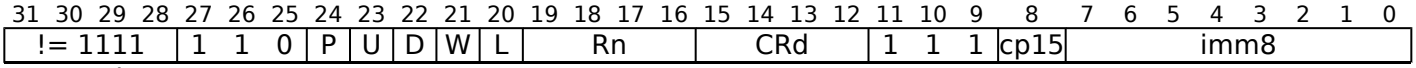
The following constraints also apply to this encoding: cond != 1111 && P:U:D:W != 00x0 && cond != 1111

Decode fields P	U	W	L	Rn	size	imm8	Instruction Details	Feature
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		01		VSTR — half-precision scalar	FEAT_FP16
1		0	0		10		VSTR — single-precision scalar	-
1		0	0		11		VSTR — double-precision scalar	-
1		0	1	!= 1111	01		VLDR (immediate) — half-precision scalar	FEAT_FP16
1		0	1	!= 1111	10		VLDR (immediate) — single-precision scalar	-

Decode fields							Instruction Details	Feature
P	U	W	L	Rn	size	imm8		
1		0	1	!= 1111	11		VLDR (immediate) — double-precision scalar	-
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	01		VLDR (literal) — half-precision scalar	FEAT_FP16
1		0	1	1111	10		VLDR (literal) — single-precision scalar	-
1		0	1	1111	11		VLDR (literal) — double-precision scalar	-
1	1	1					UNALLOCATED	-

System register load/store

These instructions are under Advanced SIMD and System register load/store and 64-bit move.

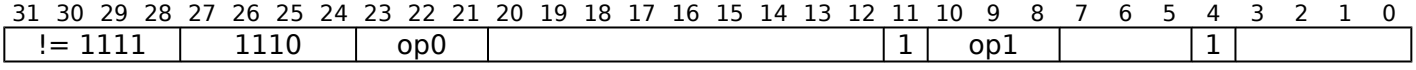


The following constraints also apply to this encoding: cond != 1111 && P:U:D:W != 00x0 && cond != 1111

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 SIMD and System register 32-bit move

These instructions are under System register access, Advanced SIMD, floating-point, and Supervisor call.



Decode fields		Instruction details	Architecture version
op0	op1		
000	000	UNALLOCATED	-
000	001	VMOV (between general-purpose register and half-precision)	FEAT_FP16

000	010	VMOV (between general-purpose register and single-precision)	-
001	010	UNALLOCATED	-
01x	010	UNALLOCATED	-
10x	010	UNALLOCATED	-
110	010	UNALLOCATED	-
111	010	Floating-point move special register	-
	011	Advanced SIMD 8/16/32-bit element move/duplicate	-
	10x	UNALLOCATED	-
	11x	System register 32-bit move	-

Floating-point move special register

These instructions are under Advanced SIMD and System register 32-bit move.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				1	1	1	0	1	1	1	L	reg				Rt				1	0	1	0	(0)	(0)	(0)	1	(0)	(0)	(0)	(0)
cond																															

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields L	Instruction Details
0	VMSR
1	VMRS

Advanced SIMD 8/16/32-bit element move/duplicate

These instructions are under Advanced SIMD and System register 32-bit move.

31	30	29	28	27	26	25	24	23	22	21	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 opc1	L	opc2	Instruction Details
0xx	0		VMOV (general-purpose register to scalar)
	1		VMOV (scalar to general-purpose register)
1xx	0	0x	VDUP (general-purpose register)
1xx	0	1x	UNALLOCATED

System register 32-bit move

These instructions are under Advanced SIMD and System register 32-bit move.

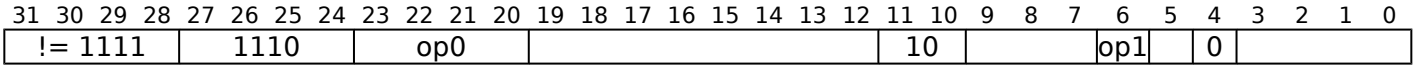
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				1	1	1	0	opc1			L	CRn				Rt				1	1	1	cp15	opc2			1	CRm			
cond																															

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields L	Instruction Details
0	MCR
1	MRC

Floating-point data-processing

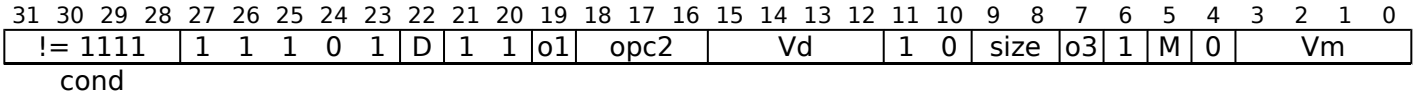
These instructions are under System register access, Advanced SIMD, floating-point, and Supervisor call.



Decode fields op0	op1	Instruction details
1x11	1	Floating-point data-processing (two registers)
1x11	0	Floating-point move immediate
!= 1x11		Floating-point data-processing (three registers)

Floating-point data-processing (two registers)

These instructions are under Floating-point data-processing.



The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields o1	opc2	size	o3	Instruction Details	Feature
		00		UNALLOCATED	-
0	000	01	0	UNALLOCATED	-
0	000	01	1	VABS — half-precision scalar	FEAT_FP16
0	000	10	0	VMOV (register) — single-precision scalar	-
0	000	10	1	VABS — single-precision scalar	-
0	000	11	0	VMOV (register) — double-precision scalar	-
0	000	11	1	VABS — double-precision scalar	-
0	001	01	0	VNEG — half-precision scalar	FEAT_FP16
0	001	01	1	VSQRT — half-precision scalar	FEAT_FP16
0	001	10	0	VNEG — single-precision scalar	-
0	001	10	1	VSQRT — single-precision scalar	-
0	001	11	0	VNEG — double-precision scalar	-
0	001	11	1	VSQRT — double-precision scalar	-
0	010	01		UNALLOCATED	-
0	010	10	0	VCVTB — half-precision to single-precision	-
0	010	10	1	VCVTT — half-precision to single-precision	-
0	010	11	0	VCVTB — half-precision to double-precision	-
0	010	11	1	VCVTT — half-precision to double-precision	-
0	011	01	0	VCVTB (BFloat16)	FEAT_AA32BF16
0	011	01	1	VCVTT (BFloat16)	FEAT_AA32BF16
0	011	10	0	VCVTB — single-precision to half-precision	-
0	011	10	1	VCVTT — single-precision to half-precision	-

Decode fields				Instruction Details	Feature
o1	opc2	size	o3		
0	011	11	0	VCVTB — double-precision to half-precision	-
0	011	11	1	VCVTT — double-precision to half-precision	-
0	100	01	0	VCMP	FEAT_FP16
0	100	01	1	VCMPE	FEAT_FP16
0	100	10	0	VCMP	-
0	100	10	1	VCMPE	-
0	100	11	0	VCMP	-
0	100	11	1	VCMPE	-
0	101	01	0	VCMP	FEAT_FP16
0	101	01	1	VCMPE	FEAT_FP16
0	101	10	0	VCMP	-
0	101	10	1	VCMPE	-
0	101	11	0	VCMP	-
0	101	11	1	VCMPE	-
0	110	01	0	VRINTR — half-precision scalar	FEAT_FP16
0	110	01	1	VRINTZ (floating-point) — half-precision scalar	FEAT_FP16
0	110	10	0	VRINTR — single-precision scalar	-
0	110	10	1	VRINTZ (floating-point) — single-precision scalar	-
0	110	11	0	VRINTR — double-precision scalar	-
0	110	11	1	VRINTZ (floating-point) — double-precision scalar	-
0	111	01	0	VRINTX (floating-point) — half-precision scalar	FEAT_FP16
0	111	01	1	UNALLOCATED	-
0	111	10	0	VRINTX (floating-point) — single-precision scalar	-
0	111	10	1	VCVT (between double-precision and single-precision) — single-precision to double-precision	-
0	111	11	0	VRINTX (floating-point) — double-precision scalar	-
0	111	11	1	VCVT (between double-precision and single-precision) — double-precision to single-precision	-
1	000	01		VCVT (integer to floating-point, floating-point) — half-precision scalar	FEAT_FP16
1	000	10		VCVT (integer to floating-point, floating-point) — single-precision scalar	-
1	000	11		VCVT (integer to floating-point, floating-point) — double-precision scalar	-
1	001	01		UNALLOCATED	-
1	001	10		UNALLOCATED	-
1	001	11	0	UNALLOCATED	-
1	001	11	1	VJCVT	FEAT_JSCVT
1	01x	01		VCVT (between floating-point and fixed-point, floating-point)	FEAT_FP16
1	01x	10		VCVT (between floating-point and fixed-point, floating-point)	-
1	01x	11		VCVT (between floating-point and fixed-point, floating-point)	-
1	100	01	0	VCVTR	FEAT_FP16
1	100	01	1	VCVT (floating-point to integer, floating-point)	FEAT_FP16
1	100	10	0	VCVTR	-
1	100	10	1	VCVT (floating-point to integer, floating-point)	-
1	100	11	0	VCVTR	-
1	100	11	1	VCVT (floating-point to integer, floating-point)	-
1	101	01	0	VCVTR	FEAT_FP16
1	101	01	1	VCVT (floating-point to integer, floating-point)	FEAT_FP16

Decode fields o1	opc2	size	o3	Instruction Details	Feature
1	101	10	0	VCVTR	-
1	101	10	1	VCVT (floating-point to integer, floating-point)	-
1	101	11	0	VCVTR	-
1	101	11	1	VCVT (floating-point to integer, floating-point)	-
1	11x	01		VCVT (between floating-point and fixed-point, floating-point)	FEAT_FP16
1	11x	10		VCVT (between floating-point and fixed-point, floating-point)	-
1	11x	11		VCVT (between floating-point and fixed-point, floating-point)	-

Floating-point move immediate

These instructions are under Floating-point data-processing.

31 30 29 28	27	26	25	24	23	22	21	20	19 18 17 16	15 14 13 12	11	10	9 8	7	6	5	4	3 2 1 0
!= 1111	1	1	1	0	1	D	1	1	imm4H	Vd	1	0	size	(0)	0	(0)	0	imm4L
cond																		

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields size	Instruction Details	Feature
00	UNALLOCATED	-
01	VMOV (immediate) — half-precision scalar	FEAT_FP16
10	VMOV (immediate) — single-precision scalar	-
11	VMOV (immediate) — double-precision scalar	-

Floating-point data-processing (three registers)

These instructions are under Floating-point data-processing.

31 30 29 28	27	26	25	24	23	22	21	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																		

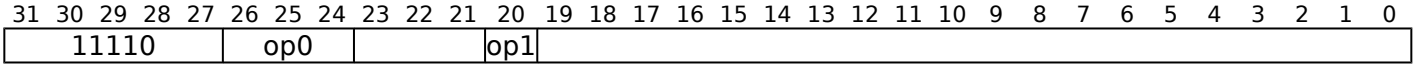
The following constraints also apply to this encoding: cond != 1111 && o0:D:o1 != 1x11 && cond != 1111

Decode fields o0:o1	size	o2	Instruction Details	Feature
!= 111	00		UNALLOCATED	-
000	01	0	VMLA (floating-point) — half-precision scalar	FEAT_FP16
000	01	1	VMLS (floating-point) — half-precision scalar	FEAT_FP16
000	10	0	VMLA (floating-point) — single-precision scalar	-
000	10	1	VMLS (floating-point) — single-precision scalar	-
000	11	0	VMLA (floating-point) — double-precision scalar	-
000	11	1	VMLS (floating-point) — double-precision scalar	-
001	01	0	VNMLS — half-precision scalar	FEAT_FP16
001	01	1	VNMLA — half-precision scalar	FEAT_FP16
001	10	0	VNMLS — single-precision scalar	-
001	10	1	VNMLA — single-precision scalar	-
001	11	0	VNMLS — double-precision scalar	-
001	11	1	VNMLA — double-precision scalar	-
010	01	0	VMUL (floating-point) — half-precision scalar	FEAT_FP16

Decode fields o0:o1	size	o2	Instruction Details	Feature
010	01	1	VNMUL — half-precision scalar	FEAT_FP16
010	10	0	VMUL (floating-point) — single-precision scalar	-
010	10	1	VNMUL — single-precision scalar	-
010	11	0	VMUL (floating-point) — double-precision scalar	-
010	11	1	VNMUL — double-precision scalar	-
011	01	0	VADD (floating-point) — half-precision scalar	FEAT_FP16
011	01	1	VSUB (floating-point) — half-precision scalar	FEAT_FP16
011	10	0	VADD (floating-point) — single-precision scalar	-
011	10	1	VSUB (floating-point) — single-precision scalar	-
011	11	0	VADD (floating-point) — double-precision scalar	-
011	11	1	VSUB (floating-point) — double-precision scalar	-
100	01	0	VDIV — half-precision scalar	FEAT_FP16
100	10	0	VDIV — single-precision scalar	-
100	11	0	VDIV — double-precision scalar	-
101	01	0	VFNMS — half-precision scalar	FEAT_FP16
101	01	1	VFNMA — half-precision scalar	FEAT_FP16
101	10	0	VFNMS — single-precision scalar	-
101	10	1	VFNMA — single-precision scalar	-
101	11	0	VFNMS — double-precision scalar	-
101	11	1	VFNMA — double-precision scalar	-
110	01	0	VFMA — half-precision scalar	FEAT_FP16
110	01	1	VFMS — half-precision scalar	FEAT_FP16
110	10	0	VFMA — single-precision scalar	-
110	10	1	VFMS — single-precision scalar	-
110	11	0	VFMA — double-precision scalar	-
110	11	1	VFMS — double-precision scalar	-

Unconditional instructions

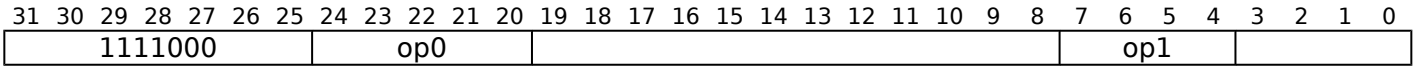
These instructions are under the top-level.



Decode fields op0	op1	Instruction details
00x		Miscellaneous
01x		Advanced SIMD data-processing
1xx	1	Memory hints and barriers
100	0	Advanced SIMD element or structure load/store
101	0	UNALLOCATED
11x	0	UNALLOCATED

Miscellaneous

These instructions are under Unconditional instructions.



Decode fields op0	op1	Instruction details	Architecture version
0xxxx		UNALLOCATED	-
10000	xx0x	Change Process State	-
10001	1000	UNALLOCATED	-
10001	x100	UNALLOCATED	-
10001	xx01	UNALLOCATED	-
10001	0000	SETPAN	FEAT_PAN
1000x	0111	UNALLOCATED	-
10010	0111	CONSTRAINED UNPREDICTABLE	-
10011	0111	UNALLOCATED	-
1001x	xx0x	UNALLOCATED	-
100xx	0011	UNALLOCATED	-
100xx	0x10	UNALLOCATED	-
100xx	1x1x	UNALLOCATED	-
101xx		UNALLOCATED	-
11xxx		UNALLOCATED	-

The behavior of the CONSTRAINED UNPREDICTABLE encodings in this table is described in *CONSTRAINED UNPREDICTABLE behavior for A32 and T32 instruction encodings*

Change Process State

These instructions are under Miscellaneous.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	0	1	0	0	0	0	imod		M	op	(0)		(0)		(0)		(0)		E	A	I	F	0	mode		

Decode fields imod	M	op	I	F	mode	Instruction Details
		1	0	0	0xxxx	SETEND
00	1	0				CPS, CPSID, CPSIE — change mode
10		0				CPS, CPSID, CPSIE — interrupt enable and change mode
		1	0	0	1xxxx	UNALLOCATED
		1	0	1		UNALLOCATED
		1	1			UNALLOCATED
11		0				CPS, CPSID, CPSIE — interrupt disable and change mode

Advanced SIMD data-processing

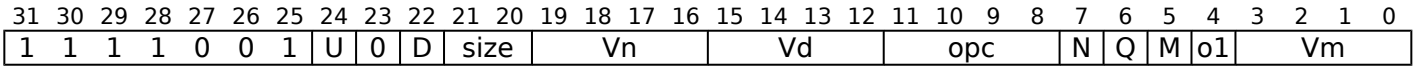
These instructions are under Unconditional instructions.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1111001								op0																			op1					

Decode fields op0	op1	Instruction details
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.

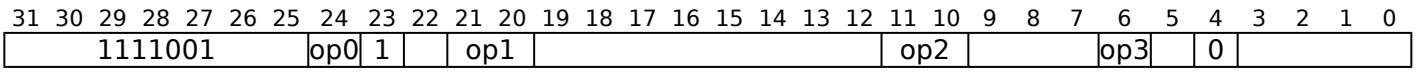


Decode fields					Instruction Details	Feature
U	size	opc	Q	o1		
0	0x	1100		1	VFMA	-
0	0x	1101		0	VADD (floating-point)	-
0	0x	1101		1	VMLA (floating-point)	-
0	0x	1110		0	VCEQ (register) — A2	-
0	0x	1111		0	VMAX (floating-point)	-
0	0x	1111		1	VRECPS	-
		0000		0	VHADD	-
0	00	0001		1	VAND (register)	-
		0000		1	VQADD	-
		0001		0	VRHADD	-
0	00	1100		0	SHA1C	-
		0010		0	VHSUB	-
0	01	0001		1	VBIC (register)	-
		0010		1	VQSUB	-
		0011		0	VCGT (register) — A1	-
		0011		1	VCGE (register) — A1	-
0	01	1100		0	SHA1P	-
0	1x	1100		1	VFMS	-
0	1x	1101		0	VSUB (floating-point)	-
0	1x	1101		1	VMLS (floating-point)	-
0	1x	1110		0	UNALLOCATED	-
0	1x	1111		0	VMIN (floating-point)	-
0	1x	1111		1	VRSQRTS	-
		0100		0	VSHL (register)	-
0		1000		0	VADD (integer)	-
0	10	0001		1	VORR (register)	-
0		1000		1	VTST	-
		0100		1	VQSHL (register)	-
0		1001		0	VMLA (integer)	-
		0101		0	VRSHL	-
		0101		1	VQRSHL	-
0		1011		0	VQDMULH	-
0	10	1100		0	SHA1M	-
0		1011		1	VPADD (integer)	-
		0110		0	VMAX (integer)	-
0	11	0001		1	VORN (register)	-
		0110		1	VMIN (integer)	-
		0111		0	VABD (integer)	-
		0111		1	VABA	-
0	11	1100		0	SHA1SU0	-
1	0x	1101		0	VPADD (floating-point)	-
1	0x	1101		1	VMUL (floating-point)	-

Decode fields					Instruction Details	Feature
U	size	opc	Q	o1		
1	0x	1110		0	VCGE (register) — A2	-
1	0x	1110		1	VACGE	-
1	0x	1111	0	0	VPMAX (floating-point)	-
1	0x	1111		1	VMAXNM	-
1	00	0001		1	VEOR	-
		1001		1	VMUL (integer and polynomial)	-
1	00	1100		0	SHA256H	-
		1010	0	0	VPMAX (integer)	-
1	01	0001		1	VBSL	-
		1010	0	1	VPMIN (integer)	-
		1010	1		UNALLOCATED	-
1	01	1100		0	SHA256H2	-
1	1x	1101		0	VABD (floating-point)	-
1	1x	1110		0	VCGT (register) — A2	-
1	1x	1110		1	VACGT	-
1	1x	1111	0	0	VPMIN (floating-point)	-
1	1x	1111		1	VMINNM	-
1		1000		0	VSUB (integer)	-
1	10	0001		1	VBIT	-
1		1000		1	VCEQ (register) — A1	-
1		1001		0	VMLS (integer)	-
1		1011		0	VQRDMULH	-
1	10	1100		0	SHA256SU1	-
1		1011		1	VQRDMLAH	FEAT_RDM
1	11	0001		1	VBIF	-
1		1100		1	VQRDMLSH	FEAT_RDM
1		1111	1	0	UNALLOCATED	-

Advanced SIMD two registers, or three registers of different lengths

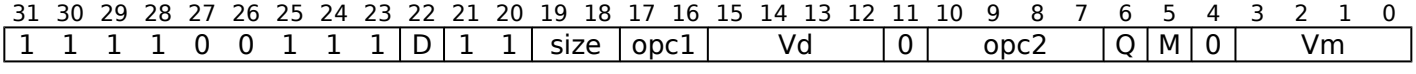
These instructions are under Advanced SIMD data-processing.



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.



Decode fields				Instruction Details	Feature
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)	-
	01	x001		VCGE (immediate #0)	-
	01	x010		VCEQ (immediate #0)	-
	01	x011		VCLE (immediate #0)	-
	01	x100		VCLT (immediate #0)	-
	01	x110		VABS	-
	01	x111		VNEG	-
	01	0101	1	SHA1H	-
01	10	1100	1	VCVT (from single-precision to BFloat16, Advanced SIMD)	FEAT_AA32BF16
	10	0001		VTRN	-
	10	0010		VUZP	-
	10	0011		VZIP	-
	10	0100	0	VMOVN	-
	10	0100	1	VQMOVN, VQMOVUN — VQMOVUN	-
	10	0101		VQMOVN, VQMOVUN — VQMOVN	-
	10	0110	0	VSHLL	-
	10	0111	0	SHA1SU1	-
	10	0111	1	SHA256SU0	-
	10	1000		VRINTN (Advanced SIMD)	-
	10	1001		VRINTX (Advanced SIMD)	-
	10	1010		VRINTA (Advanced SIMD)	-
	10	1011		VRINTZ (Advanced SIMD)	-
10	10	1100	1	UNALLOCATED	-
	10	1100	0	VCVT (between half-precision and single-precision, Advanced SIMD) — single-precision to half-precision	-
	10	1101		VRINTM (Advanced SIMD)	-
	10	1110	0	VCVT (between half-precision and single-precision, Advanced SIMD) — half-precision to single-precision	-
	10	1110	1	UNALLOCATED	-
	10	1111		VRINTP (Advanced SIMD)	-

Decode fields				Instruction Details	Feature
size	opc1	opc2	Q		
	11	000x		VCVTA (Advanced SIMD)	-
	11	001x		VCVTN (Advanced SIMD)	-
	11	010x		VCVTP (Advanced SIMD)	-
	11	011x		VCVTM (Advanced SIMD)	-
	11	10x0		VRECPE	-
	11	10x1		VRSQRTE	-
11	10	1100	1	UNALLOCATED	-
	11	11xx		VCVT (between floating-point and integer, Advanced SIMD)	-

Advanced SIMD duplicate (scalar)

These instructions are under Advanced SIMD two registers, or three registers of different lengths.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	imm4				Vd				1	1	opc			Q	M	0	Vm			

Decode fields	
opc	Instruction Details
000	VDUP (scalar)
001	UNALLOCATED
01x	UNALLOCATED
1xx	UNALLOCATED

Advanced SIMD three registers of different lengths

These instructions are under Advanced SIMD two registers, or three registers of different lengths.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	1	D	!= 11		Vn				Vd				opc				N	0	M	0	Vm			
										size																					

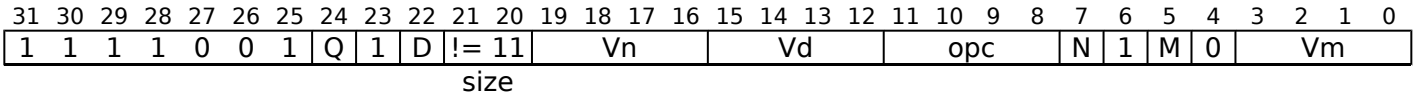
The following constraints also apply to this encoding: size != 11 && size != 11

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)

Decode fields U	opc	Instruction Details
1	1001	UNALLOCATED
1	1011	UNALLOCATED
1	1101	UNALLOCATED
	1111	UNALLOCATED

Advanced SIMD two registers and a scalar

These instructions are under Advanced SIMD two registers, or three registers of different lengths.

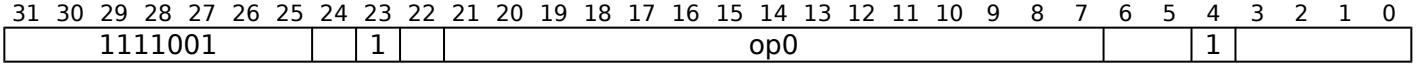


The following constraints also apply to this encoding: size != 11 && size != 11

Decode fields Q	opc	Instruction Details	Feature
	000x	VMLA (by scalar)	-
0	0011	VQDMLAL	-
	0010	VMLAL (by scalar)	-
0	0111	VQDMLSL	-
	010x	VMLS (by scalar)	-
0	1011	VQDMULL	-
	0110	VMLSL (by scalar)	-
	100x	VMUL (by scalar)	-
1	0011	UNALLOCATED	-
	1010	VMULL (by scalar)	-
1	0111	UNALLOCATED	-
	1100	VQDMULH	-
	1101	VQRDMULH	-
1	1011	UNALLOCATED	-
	1110	VQRDMLAH	FEAT_RDM
	1111	VQRDMLSH	FEAT_RDM

Advanced SIMD shifts and immediate generation

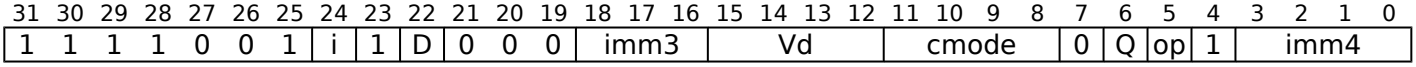
These instructions are under Advanced SIMD data-processing.



Decode fields op0	Instruction details
000xxxxxxxxxx0	Advanced SIMD one register and modified immediate
!= 000xxxxxxxxxx0	Advanced SIMD two registers and shift amount

Advanced SIMD one register and modified immediate

These instructions are under Advanced SIMD shifts and immediate generation.



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 != 000xxxxxxxxxx0

U	Decode fields imm3H:L	imm3L	opc	Q	Instruction Details
	!= 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

Memory hints and barriers

These instructions are under Unconditional instructions.

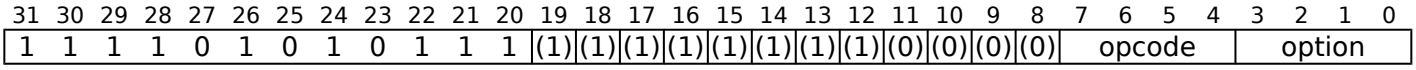
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
111101						op0					1																op1				

Decode fields op0	op1	Instruction details
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.

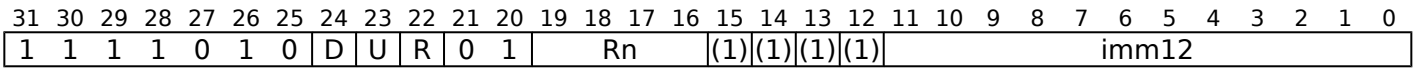


Decode fields opcode	option	Instruction Details
0000		CONSTRAINED UNPREDICTABLE
0001		CLREX
001x		CONSTRAINED UNPREDICTABLE
0100	!= 0x00	DSB
0100	0000	SSBB
0100	0100	PSSBB
0101		DMB
0110		ISB
0111		SB
1xxx		CONSTRAINED UNPREDICTABLE

The behavior of the CONSTRAINED UNPREDICTABLE encodings in this table is described in *CONSTRAINED UNPREDICTABLE behavior for A32 and T32 instruction encodings*

Preload (immediate)

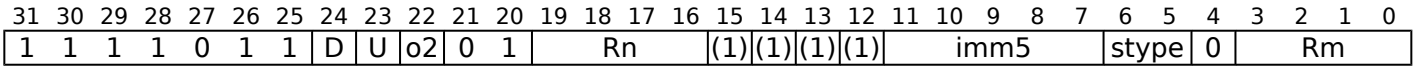
These instructions are under Memory hints and barriers.



Decode fields D	R	Rn	Instruction Details
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)

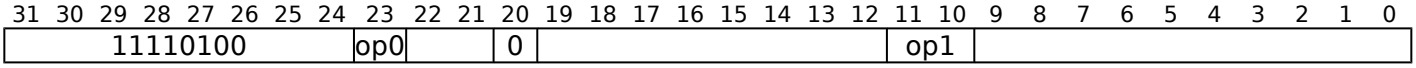
These instructions are under Memory hints and barriers.



D	Decode fields o2	imm5:stype	Instruction Details
0	0		Reserved hint, behaves as NOP
0	1	!= 0000011	PLI (register) — shift or rotate by value
0	1	0000011	PLI (register) — rotate right with extend
1	0	!= 0000011	PLD, PLDW (register) — preload write, optional shift or rotate
1	0	0000011	PLD, PLDW (register) — preload write, rotate right with extend
1	1	!= 0000011	PLD, PLDW (register) — preload read, optional shift or rotate
1	1	0000011	PLD, PLDW (register) — preload read, rotate right with extend

Advanced SIMD element or structure load/store

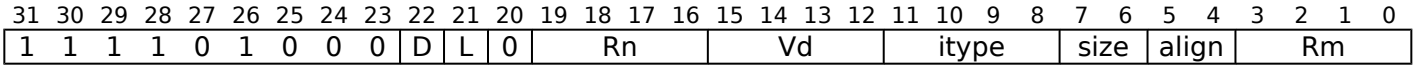
These instructions are under Unconditional instructions.



op0	Decode fields op1	Instruction details
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.



L	Decode fields itype	Rm	Instruction Details
0	000x	!= 11x1	VST4 (multiple 4-element structures)
0	000x	1101	VST4 (multiple 4-element structures)
0	000x	1111	VST4 (multiple 4-element structures)
0	0010	!= 11x1	VST1 (multiple single elements)
0	0010	1101	VST1 (multiple single elements)
0	0010	1111	VST1 (multiple single elements)
0	0011	!= 11x1	VST2 (multiple 2-element structures)
0	0011	1101	VST2 (multiple 2-element structures)
0	0011	1111	VST2 (multiple 2-element structures)
0	010x	!= 11x1	VST3 (multiple 3-element structures)
0	010x	1101	VST3 (multiple 3-element structures)
0	010x	1111	VST3 (multiple 3-element structures)
0	0110	!= 11x1	VST1 (multiple single elements)
0	0110	1101	VST1 (multiple single elements)
0	0110	1111	VST1 (multiple single elements)

Decode fields			Instruction Details
L	itype	Rm	
0	0111	!= 11x1	VST1 (multiple single elements)
0	0111	1101	VST1 (multiple single elements)
0	0111	1111	VST1 (multiple single elements)
0	100x	!= 11x1	VST2 (multiple 2-element structures)
0	100x	1101	VST2 (multiple 2-element structures)
0	100x	1111	VST2 (multiple 2-element structures)
0	1010	!= 11x1	VST1 (multiple single elements)
0	1010	1101	VST1 (multiple single elements)
0	1010	1111	VST1 (multiple single elements)
1	000x	!= 11x1	VLD4 (multiple 4-element structures)
1	000x	1101	VLD4 (multiple 4-element structures)
1	000x	1111	VLD4 (multiple 4-element structures)
1	0010	!= 11x1	VLD1 (multiple single elements)
1	0010	1101	VLD1 (multiple single elements)
1	0010	1111	VLD1 (multiple single elements)
1	0011	!= 11x1	VLD2 (multiple 2-element structures)
1	0011	1101	VLD2 (multiple 2-element structures)
1	0011	1111	VLD2 (multiple 2-element structures)
1	010x	!= 11x1	VLD3 (multiple 3-element structures)
1	010x	1101	VLD3 (multiple 3-element structures)
1	010x	1111	VLD3 (multiple 3-element structures)
	1011		UNALLOCATED
1	0110	!= 11x1	VLD1 (multiple single elements)
1	0110	1101	VLD1 (multiple single elements)
1	0110	1111	VLD1 (multiple single elements)
1	0111	!= 11x1	VLD1 (multiple single elements)
1	0111	1101	VLD1 (multiple single elements)
1	0111	1111	VLD1 (multiple single elements)
	11xx		UNALLOCATED
1	100x	!= 11x1	VLD2 (multiple 2-element structures)
1	100x	1101	VLD2 (multiple 2-element structures)
1	100x	1111	VLD2 (multiple 2-element structures)
1	1010	!= 11x1	VLD1 (multiple single elements)
1	1010	1101	VLD1 (multiple single elements)
1	1010	1111	VLD1 (multiple single elements)

Advanced SIMD load single structure to all lanes

These instructions are under Advanced SIMD element or structure load/store.

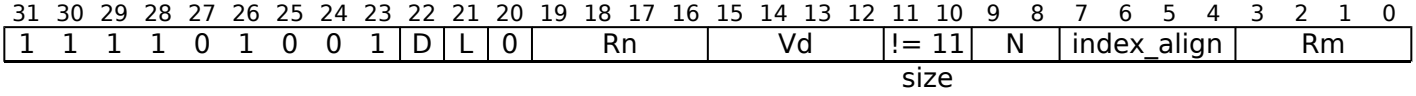
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	1	D	L	0	Rn				Vd				1	1	N		size		T	a	Rm			

Decode fields				Instruction Details
L	N	a	Rm	
0				UNALLOCATED
1	00		!= 11x1	VLD1 (single element to all lanes)
1	00		1101	VLD1 (single element to all lanes)
1	00		1111	VLD1 (single element to all lanes)

Decode fields				Instruction Details
L	N	a	Rm	
1	01		!= 11x1	VLD2 (single 2-element structure to all lanes)
1	01		1101	VLD2 (single 2-element structure to all lanes)
1	01		1111	VLD2 (single 2-element structure to all lanes)
1	10	0	!= 11x1	VLD3 (single 3-element structure to all lanes)
1	10	0	1101	VLD3 (single 3-element structure to all lanes)
1	10	0	1111	VLD3 (single 3-element structure to all lanes)
1	10	1		UNALLOCATED
1	11		!= 11x1	VLD4 (single 4-element structure to all lanes)
1	11		1101	VLD4 (single 4-element structure to all lanes)
1	11		1111	VLD4 (single 4-element structure to all lanes)

Advanced SIMD load/store single structure to one lane

These instructions are under Advanced SIMD element or structure load/store.



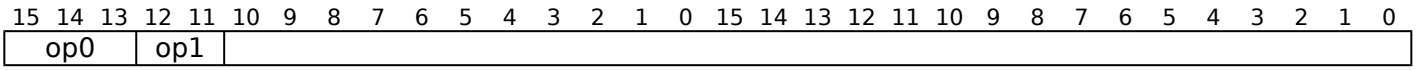
The following constraints also apply to this encoding: size != 11 && size != 11

Decode fields				Instruction Details
L	size	N	Rm	
0	00	00	!= 11x1	VST1 (single element from one lane)
0	00	00	1101	VST1 (single element from one lane)
0	00	00	1111	VST1 (single element from one lane)
0	00	01	!= 11x1	VST2 (single 2-element structure from one lane)
0	00	01	1101	VST2 (single 2-element structure from one lane)
0	00	01	1111	VST2 (single 2-element structure from one lane)
0	00	10	!= 11x1	VST3 (single 3-element structure from one lane)
0	00	10	1101	VST3 (single 3-element structure from one lane)
0	00	10	1111	VST3 (single 3-element structure from one lane)
0	00	11	!= 11x1	VST4 (single 4-element structure from one lane)
0	00	11	1101	VST4 (single 4-element structure from one lane)
0	00	11	1111	VST4 (single 4-element structure from one lane)
0	01	00	!= 11x1	VST1 (single element from one lane)
0	01	00	1101	VST1 (single element from one lane)
0	01	00	1111	VST1 (single element from one lane)
0	01	01	!= 11x1	VST2 (single 2-element structure from one lane)
0	01	01	1101	VST2 (single 2-element structure from one lane)
0	01	01	1111	VST2 (single 2-element structure from one lane)
0	01	10	!= 11x1	VST3 (single 3-element structure from one lane)
0	01	10	1101	VST3 (single 3-element structure from one lane)
0	01	10	1111	VST3 (single 3-element structure from one lane)
0	01	11	!= 11x1	VST4 (single 4-element structure from one lane)
0	01	11	1101	VST4 (single 4-element structure from one lane)
0	01	11	1111	VST4 (single 4-element structure from one lane)
0	10	00	!= 11x1	VST1 (single element from one lane)
0	10	00	1101	VST1 (single element from one lane)

Decode fields				Instruction Details
L	size	N	Rm	
0	10	00	1111	VST1 (single element from one lane)
0	10	01	!= 11x1	VST2 (single 2-element structure from one lane)
0	10	01	1101	VST2 (single 2-element structure from one lane)
0	10	01	1111	VST2 (single 2-element structure from one lane)
0	10	10	!= 11x1	VST3 (single 3-element structure from one lane)
0	10	10	1101	VST3 (single 3-element structure from one lane)
0	10	10	1111	VST3 (single 3-element structure from one lane)
0	10	11	!= 11x1	VST4 (single 4-element structure from one lane)
0	10	11	1101	VST4 (single 4-element structure from one lane)
0	10	11	1111	VST4 (single 4-element structure from one lane)
1	00	00	!= 11x1	VLD1 (single element to one lane)
1	00	00	1101	VLD1 (single element to one lane)
1	00	00	1111	VLD1 (single element to one lane)
1	00	01	!= 11x1	VLD2 (single 2-element structure to one lane)
1	00	01	1101	VLD2 (single 2-element structure to one lane)
1	00	01	1111	VLD2 (single 2-element structure to one lane)
1	00	10	!= 11x1	VLD3 (single 3-element structure to one lane)
1	00	10	1101	VLD3 (single 3-element structure to one lane)
1	00	10	1111	VLD3 (single 3-element structure to one lane)
1	00	11	!= 11x1	VLD4 (single 4-element structure to one lane)
1	00	11	1101	VLD4 (single 4-element structure to one lane)
1	00	11	1111	VLD4 (single 4-element structure to one lane)
1	01	00	!= 11x1	VLD1 (single element to one lane)
1	01	00	1101	VLD1 (single element to one lane)
1	01	00	1111	VLD1 (single element to one lane)
1	01	01	!= 11x1	VLD2 (single 2-element structure to one lane)
1	01	01	1101	VLD2 (single 2-element structure to one lane)
1	01	01	1111	VLD2 (single 2-element structure to one lane)
1	01	10	!= 11x1	VLD3 (single 3-element structure to one lane)
1	01	10	1101	VLD3 (single 3-element structure to one lane)
1	01	10	1111	VLD3 (single 3-element structure to one lane)
1	01	11	!= 11x1	VLD4 (single 4-element structure to one lane)
1	01	11	1101	VLD4 (single 4-element structure to one lane)
1	01	11	1111	VLD4 (single 4-element structure to one lane)
1	10	00	!= 11x1	VLD1 (single element to one lane)
1	10	00	1101	VLD1 (single element to one lane)
1	10	00	1111	VLD1 (single element to one lane)
1	10	01	!= 11x1	VLD2 (single 2-element structure to one lane)
1	10	01	1101	VLD2 (single 2-element structure to one lane)
1	10	01	1111	VLD2 (single 2-element structure to one lane)
1	10	10	!= 11x1	VLD3 (single 3-element structure to one lane)
1	10	10	1101	VLD3 (single 3-element structure to one lane)
1	10	10	1111	VLD3 (single 3-element structure to one lane)
1	10	11	!= 11x1	VLD4 (single 4-element structure to one lane)
1	10	11	1101	VLD4 (single 4-element structure to one lane)
1	10	11	1111	VLD4 (single 4-element structure to one lane)

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

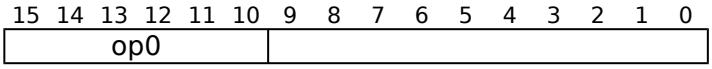
Top-level encodings for T32



Decode fields op0	op1	Instruction details
!= 111		16-bit
111	00	B — T2
111	!= 00	32-bit

16-bit

These instructions are under the top-level.

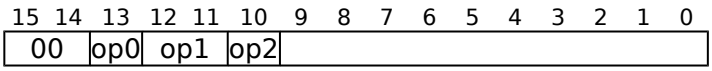


The following constraints also apply to this encoding: op0<5:3> != 111

Decode fields op0	Instruction details
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

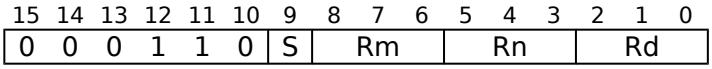
These instructions are under 16-bit.



Decode fields op0	op1	op2	Instruction details
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)

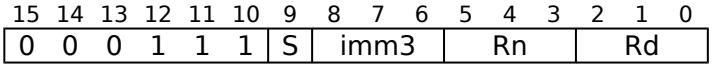
These instructions are under Shift (immediate), add, subtract, move, and compare.



Decode fields S	Instruction Details
0	ADD, ADDS (register)
1	SUB, SUBS (register)

Add, subtract (two low registers and immediate)

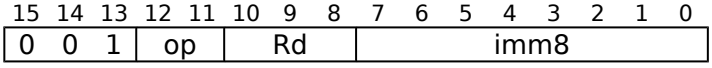
These instructions are under Shift (immediate), add, subtract, move, and compare.



Decode fields S	Instruction Details
0	ADD, ADDS (immediate)
1	SUB, SUBS (immediate)

Add, subtract, compare, move (one low register and immediate)

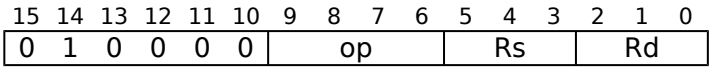
These instructions are under Shift (immediate), add, subtract, move, and compare.



Decode fields op	Instruction Details
00	MOV, MOVS (immediate)
01	CMP (immediate)
10	ADD, ADDS (immediate)
11	SUB, SUBS (immediate)

Data-processing (two low registers)

These instructions are under 16-bit.



Decode fields op	Instruction Details
0000	AND, ANDS (register)
0001	EOR, EORS (register)
0010	MOV, MOVS (register-shifted register) — logical shift left
0011	MOV, MOVS (register-shifted register) — logical shift right
0100	MOV, MOVS (register-shifted register) — arithmetic shift right
0101	ADC, ADCS (register)
0110	SBC, SBCS (register)
0111	MOV, MOVS (register-shifted register) — rotate right
1000	TST (register)
1001	RSB, RSBS (immediate)
1010	CMP (register)
1011	CMN (register)

Decode fields op	Instruction Details
1100	ORR, ORRS (register)
1101	MUL, MULS
1110	BIC, BICS (register)
1111	MVN, MVNS (register)

Special data instructions and branch and exchange

These instructions are under 16-bit.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	0	1	0	0	0	1	1	0

Decode fields op0	Instruction details
11	Branch and exchange
!= 11	Add, subtract, compare, move (two high registers)

Branch and exchange

These instructions are under Special data instructions and branch and exchange.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	1	1	L	Rm				(0)	(0)	(0)

Decode fields L	Instruction Details
0	BX
1	BLX (register)

Add, subtract, compare, move (two high registers)

These instructions are under Special data instructions and branch and exchange.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	!= 11		D	Rs				Rd		
						op									

The following constraints also apply to this encoding: op != 11 && op != 11

op	Decode fields D:Rd	Rs	Instruction Details
00	!= 1101	!= 1101	ADD, ADDS (register)
00		1101	ADD, ADDS (SP plus register) — T1
00	1101	!= 1101	ADD, ADDS (SP plus register) — T2
01			CMP (register)
10			MOV, MOVS (register)

Load/store (register offset)

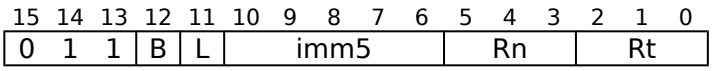
These instructions are under 16-bit.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	L	B	H	Rm			Rn			Rt		

Decode fields L	B	H	Instruction Details
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)

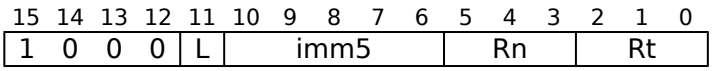
These instructions are under 16-bit.



Decode fields B	L	Instruction Details
0	0	STR (immediate)
0	1	LDR (immediate)
1	0	STRB (immediate)
1	1	LDRB (immediate)

Load/store halfword (immediate offset)

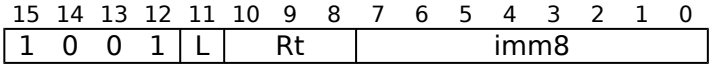
These instructions are under 16-bit.



Decode fields L	Instruction Details
0	STRH (immediate)
1	LDRH (immediate)

Load/store (SP-relative)

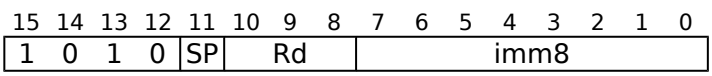
These instructions are under 16-bit.



Decode fields L	Instruction Details
0	STR (immediate)
1	LDR (immediate)

Add PC/SP (immediate)

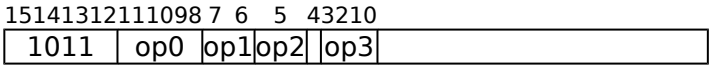
These instructions are under 16-bit.



Decode fields SP	Instruction Details
0	ADR
1	ADD, ADDS (SP plus immediate)

Miscellaneous 16-bit instructions

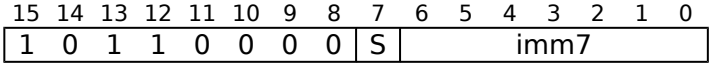
These instructions are under 16-bit.



op0	Decode fields op1	op2	op3	Instruction details	Architecture version
0000				Adjust SP (immediate)	-
0010				Extend	-
0110	00	0		SETPAN	FEAT_PAN
0110	00	1		UNALLOCATED	-
0110	01			Change Processor State	-
0110	1x			UNALLOCATED	-
0111				UNALLOCATED	-
1000				UNALLOCATED	-
1010	10			HLT	-
1010	!= 10			Reverse bytes	-
1110				BKPT	-
1111			0000	Hints	-
1111			!= 0000	IT	-
x0x1				CBNZ, CBZ	-
x10x				Push and Pop	-

Adjust SP (immediate)

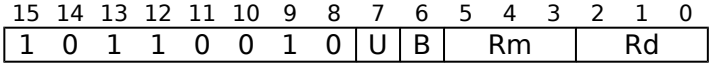
These instructions are under Miscellaneous 16-bit instructions.



Decode fields S	Instruction Details
0	ADD, ADDS (SP plus immediate)
1	SUB, SUBS (SP minus immediate)

Extend

These instructions are under Miscellaneous 16-bit instructions.

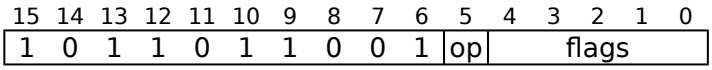


Decode fields U	B	Instruction Details
0	0	SXTH
0	1	SXTB
1	0	UXTH

Decode fields U	B	Instruction Details
1	1	UXTB

Change Processor State

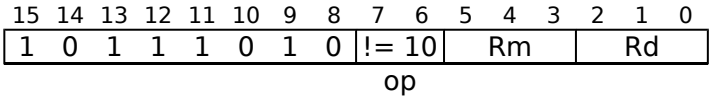
These instructions are under Miscellaneous 16-bit instructions.



Decode fields op	flags	Instruction Details
0		SETEND
1	0xxxx	CPS, CPSID, CPSIE — interrupt enable
1	1xxxx	CPS, CPSID, CPSIE — interrupt disable

Reverse bytes

These instructions are under Miscellaneous 16-bit instructions.

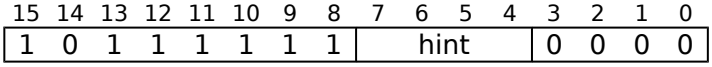


The following constraints also apply to this encoding: op != 10 && op != 10

Decode fields op	Instruction Details
00	REV
01	REV16
11	REVSH

Hints

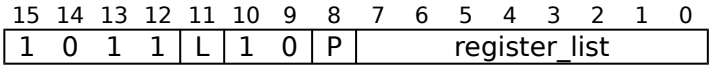
These instructions are under Miscellaneous 16-bit instructions.



Decode fields hint	Instruction Details
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

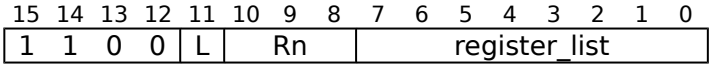
These instructions are under Miscellaneous 16-bit instructions.



Decode fields L	Instruction Details
0	PUSH
1	POP

Load/store multiple

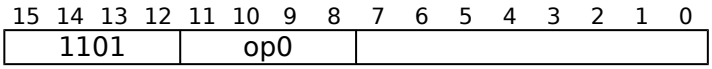
These instructions are under 16-bit.



Decode fields L	Instruction Details
0	STM, STMIA, STMEA
1	LDM, LDMIA, LDMFD

Conditional branch, and Supervisor Call

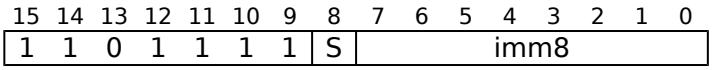
These instructions are under 16-bit.



Decode fields op0	Instruction details
111x	Exception generation
!= 111x	B — T1

Exception generation

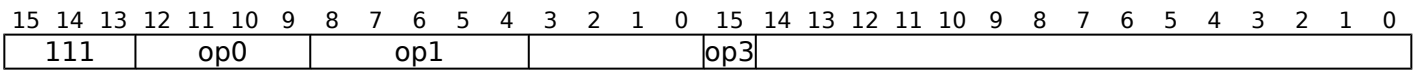
These instructions are under Conditional branch, and Supervisor Call.



Decode fields S	Instruction Details
0	UDF
1	SVC

32-bit

These instructions are under the top-level.



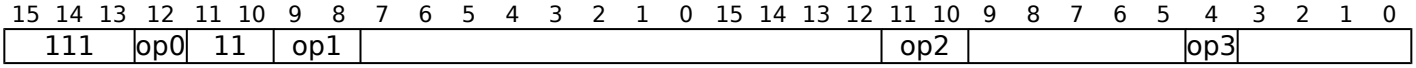
The following constraints also apply to this encoding: op0<3:2> != 00

Decode fields op0	op1	op3	Instruction details
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

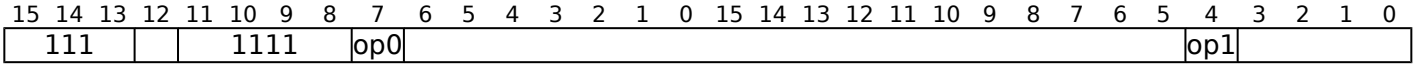
These instructions are under 32-bit.



op0	Decode fields op1	op2	op3	Instruction details
	0x	0x		UNALLOCATED
	10	0x		UNALLOCATED
	11			Advanced SIMD data-processing
0	0x	1x		Advanced SIMD and System register load/store and 64-bit move
0	10	1x	1	Advanced SIMD and System register 32-bit move
0	10	10	0	Floating-point data-processing
0	10	11	0	UNALLOCATED
1	!= 11	1x		Additional Advanced SIMD and floating-point instructions

Advanced SIMD data-processing

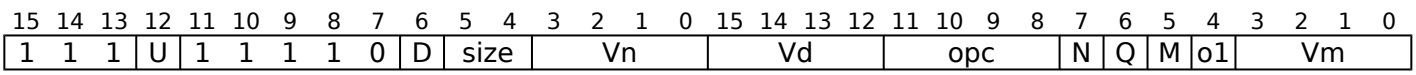
These instructions are under System register access, Advanced SIMD, and floating-point.



op0	Decode fields op1	Instruction details
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.

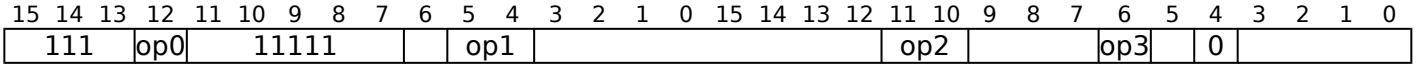


Decode fields					Instruction Details	Feature
U	size	opc	Q	o1		
0	0x	1100		1	VFMA	-
0	0x	1101		0	VADD (floating-point)	-
0	0x	1101		1	VMLA (floating-point)	-
0	0x	1110		0	VCEQ (register) — T2	-
0	0x	1111		0	VMAX (floating-point)	-
0	0x	1111		1	VRECPS	-
		0000		0	VHADD	-
0	00	0001		1	VAND (register)	-
		0000		1	VQADD	-
		0001		0	VRHADD	-
0	00	1100		0	SHA1C	-
		0010		0	VHSUB	-
0	01	0001		1	VBIC (register)	-
		0010		1	VQSUB	-
		0011		0	VCGT (register) — T1	-
		0011		1	VCGE (register) — T1	-
0	01	1100		0	SHA1P	-
0	1x	1100		1	VFMS	-
0	1x	1101		0	VSUB (floating-point)	-
0	1x	1101		1	VMLS (floating-point)	-
0	1x	1110		0	UNALLOCATED	-
0	1x	1111		0	VMIN (floating-point)	-
0	1x	1111		1	VRSQRTS	-
		0100		0	VSHL (register)	-
0		1000		0	VADD (integer)	-
0	10	0001		1	VORR (register)	-
0		1000		1	VTST	-
		0100		1	VQSHL (register)	-
0		1001		0	VMLA (integer)	-
		0101		0	VRSHL	-
		0101		1	VQRSHL	-
0		1011		0	VQDMULH	-
0	10	1100		0	SHA1M	-
0		1011		1	VPADD (integer)	-
		0110		0	VMAX (integer)	-
0	11	0001		1	VORN (register)	-
		0110		1	VMIN (integer)	-
		0111		0	VABD (integer)	-
		0111		1	VABA	-
0	11	1100		0	SHA1SU0	-
1	0x	1101		0	VPADD (floating-point)	-
1	0x	1101		1	VMUL (floating-point)	-
1	0x	1110		0	VCGE (register) — T2	-
1	0x	1110		1	VACGE	-
1	0x	1111	0	0	VPMAX (floating-point)	-
1	0x	1111		1	VMAXNM	-
1	00	0001		1	VEOR	-

Decode fields					Instruction Details	Feature
U	size	opc	Q	o1		
		1001		1	VMUL (integer and polynomial)	-
1	00	1100		0	SHA256H	-
		1010	0	0	VPMAX (integer)	-
1	01	0001		1	VBSL	-
		1010	0	1	VPMIN (integer)	-
		1010	1		UNALLOCATED	-
1	01	1100		0	SHA256H2	-
1	1x	1101		0	VABD (floating-point)	-
1	1x	1110		0	VCGT (register) — T2	-
1	1x	1110		1	VACGT	-
1	1x	1111	0	0	VPMIN (floating-point)	-
1	1x	1111		1	VMINNM	-
1		1000		0	VSUB (integer)	-
1	10	0001		1	VBIT	-
1		1000		1	VCEQ (register) — T1	-
1		1001		0	VMLS (integer)	-
1		1011		0	VQRDMULH	-
1	10	1100		0	SHA256SU1	-
1		1011		1	VQRDMLAH	FEAT_RDM
1	11	0001		1	VBIF	-
1		1100		1	VQRDMLSH	FEAT_RDM
1		1111	1	0	UNALLOCATED	-

Advanced SIMD two registers, or three registers of different lengths

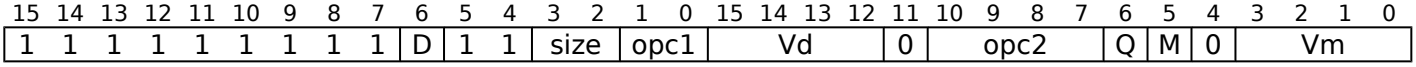
These instructions are under Advanced SIMD data-processing.



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.



Decode fields				Instruction Details	Feature
size	opc1	opc2	Q		
	00	0000		VREV64	-
	00	0001		VREV32	-

Decode fields				Instruction Details	Feature
size	opc1	opc2	Q		
	00	0010		VREV16	-
	00	0011		UNALLOCATED	-
	00	010x		VPADDL	-
	00	0110	0	AESE	-
	00	0110	1	AESD	-
	00	0111	0	AESMC	-
	00	0111	1	AESIMC	-
	00	1000		VCLS	-
00	10	0000		VSWP	-
	00	1001		VCLZ	-
	00	1010		VCNT	-
	00	1011		VMVN (register)	-
00	10	1100	1	UNALLOCATED	-
	00	110x		VPADAL	-
	00	1110		VQABS	-
	00	1111		VQNEG	-
	01	x000		VCGT (immediate #0)	-
	01	x001		VCGE (immediate #0)	-
	01	x010		VCEQ (immediate #0)	-
	01	x011		VCLE (immediate #0)	-
	01	x100		VCLT (immediate #0)	-
	01	x110		VABS	-
	01	x111		VNEG	-
	01	0101	1	SHA1H	-
01	10	1100	1	VCVT (from single-precision to BFloat16, Advanced SIMD)	FEAT_AA32BF16
	10	0001		VTRN	-
	10	0010		VUZP	-
	10	0011		VZIP	-
	10	0100	0	VMOVN	-
	10	0100	1	VQMOVN, VQMOVUN — VQMOVUN	-
	10	0101		VQMOVN, VQMOVUN — VQMOVN	-
	10	0110	0	VSHLL	-
	10	0111	0	SHA1SU1	-
	10	0111	1	SHA256SU0	-
	10	1000		VRINTN (Advanced SIMD)	-
	10	1001		VRINTX (Advanced SIMD)	-
	10	1010		VRINTA (Advanced SIMD)	-
	10	1011		VRINTZ (Advanced SIMD)	-
10	10	1100	1	UNALLOCATED	-
	10	1100	0	VCVT (between half-precision and single-precision, Advanced SIMD) — single-precision to half-precision	-
	10	1101		VRINTM (Advanced SIMD)	-
	10	1110	0	VCVT (between half-precision and single-precision, Advanced SIMD) — half-precision to single-precision	-
	10	1110	1	UNALLOCATED	-
	10	1111		VRINTP (Advanced SIMD)	-
	11	000x		VCVTA (Advanced SIMD)	-
	11	001x		VCVTN (Advanced SIMD)	-

Decode fields size	opc1	opc2	Q	Instruction Details	Feature
	11	010x		VCVTP (Advanced SIMD)	-
	11	011x		VCVTM (Advanced SIMD)	-
	11	10x0		VRECPE	-
	11	10x1		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																					

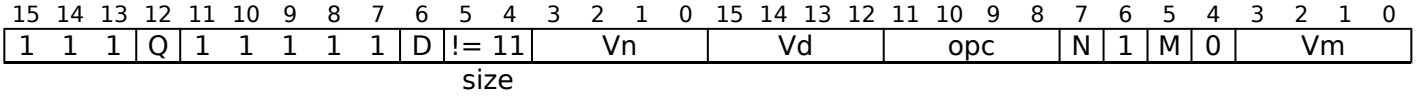
The following constraints also apply to this encoding: size != 11 && size != 11

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
1	1011	UNALLOCATED

Decode fields U	opc	Instruction Details
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.

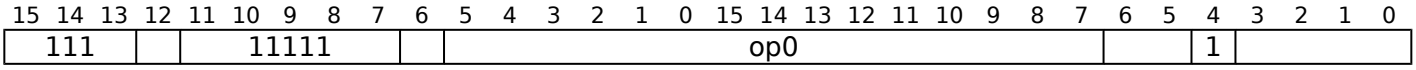


The following constraints also apply to this encoding: size != 11 && size != 11

Decode fields Q	opc	Instruction Details	Feature
	000x	VMLA (by scalar)	-
0	0011	VQDMLAL	-
	0010	VMLAL (by scalar)	-
0	0111	VQDMLSL	-
	010x	VMLS (by scalar)	-
0	1011	VQDMULL	-
	0110	VMLSL (by scalar)	-
	100x	VMUL (by scalar)	-
1	0011	UNALLOCATED	-
	1010	VMULL (by scalar)	-
1	0111	UNALLOCATED	-
	1100	VQDMULH	-
	1101	VQRDMULH	-
1	1011	UNALLOCATED	-
	1110	VQRDMLAH	FEAT_RDM
	1111	VQRDMLSH	FEAT_RDM

Advanced SIMD shifts and immediate generation

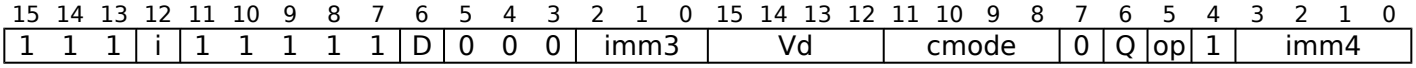
These instructions are under Advanced SIMD data-processing.



Decode fields op0	Instruction details
000xxxxxxxxxx0	Advanced SIMD one register and modified immediate
!= 000xxxxxxxxxx0	Advanced SIMD two registers and shift amount

Advanced SIMD one register and modified immediate

These instructions are under Advanced SIMD shifts and immediate generation.



Decode fields cmode	op	Instruction Details
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 != 000xxxxxxxxxx0

U	Decode fields imm3H:L	imm3L	opc	Q	Instruction Details
	!= 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

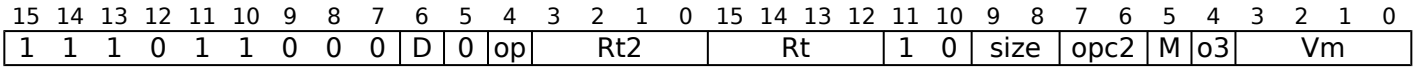
These instructions are under System register access, Advanced SIMD, and floating-point.

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 op0	op1	Instruction details
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

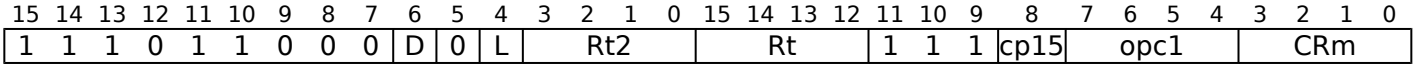
These instructions are under Advanced SIMD and System register load/store and 64-bit move.



Decode fields D	op	size	opc2	o3	Instruction Details
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

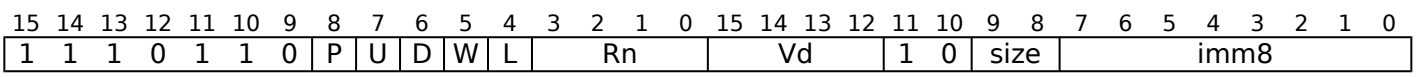
These instructions are under Advanced SIMD and System register load/store and 64-bit move.



Decode fields D	L	Instruction Details
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.

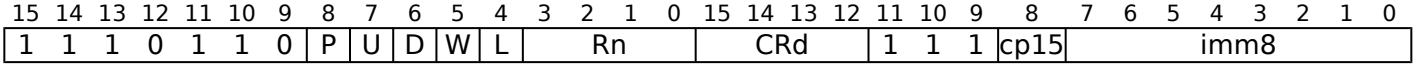


The following constraints also apply to this encoding: P:U:D:W != 00x0

Decode fields							Instruction Details	Feature
P	U	W	L	Rn	size	imm8		
0	0	1					UNALLOCATED	-
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		01		VSTR — half-precision scalar	FEAT_FP16
1		0	0		10		VSTR — single-precision scalar	-
1		0	0		11		VSTR — double-precision scalar	-
1		0	1	!= 1111	01		VLDR (immediate) — half-precision scalar	FEAT_FP16
1		0	1	!= 1111	10		VLDR (immediate) — single-precision scalar	-
1		0	1	!= 1111	11		VLDR (immediate) — double-precision scalar	-
1	0	1			0x		UNALLOCATED	-
1	0	1	0		10		VSTM, VSTMDB, VSTMIA	-
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	01		VLDR (literal) — half-precision scalar	FEAT_FP16
1		0	1	1111	10		VLDR (literal) — single-precision scalar	-
1		0	1	1111	11		VLDR (literal) — double-precision scalar	-
1	1	1					UNALLOCATED	-

System register Load/Store

These instructions are under Advanced SIMD and System register load/store and 64-bit move.



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

Decode fields						Instruction Details
P:U:W	D	L	Rn	CRd	cp15	
1x0	0	0		0101	0	STC — offset
1x0	0	1	!= 1111	0101	0	LDC (immediate) — offset
1x1	0	0		0101	0	STC — pre-indexed
1x1	0	1	!= 1111	0101	0	LDC (immediate) — pre-indexed

Advanced SIMD and System register 32-bit move

These instructions are under System register access, Advanced SIMD, and floating-point.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
11101110								op0												1	op1						1					

Decode fields		Instruction details	Architecture version
op0	op1		
000	000	UNALLOCATED	-
000	001	VMOV (between general-purpose register and half-precision)	FEAT_FP16
000	010	VMOV (between general-purpose register and single-precision)	-
001	010	UNALLOCATED	-
01x	010	UNALLOCATED	-
10x	010	UNALLOCATED	-
110	010	UNALLOCATED	-
111	010	Floating-point move special register	-
	011	Advanced SIMD 8/16/32-bit element move/duplicate	-
	10x	UNALLOCATED	-
	11x	System register 32-bit move	-

Floating-point move special register

These instructions are under Advanced SIMD and System register 32-bit move.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	1	1	1	L	reg				Rt				1	0	1	0	(0)	(0)	(0)	1	(0)	(0)	(0)	(0)

Decode fields	Instruction Details
L	
0	VMSR
1	VMRS

Advanced SIMD 8/16/32-bit element move/duplicate

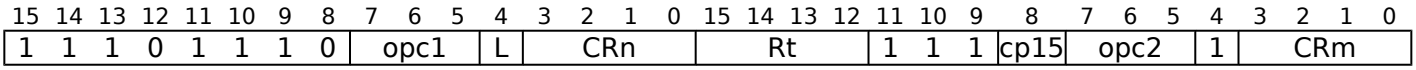
These instructions are under Advanced SIMD and System register 32-bit move.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	opc1			L	Vn				Rt				1	0	1	1	N	opc2		1	(0)	(0)	(0)	(0)

Decode fields			Instruction Details
opc1	L	opc2	
0xx	0		VMOV (general-purpose register to scalar)
	1		VMOV (scalar to general-purpose register)
1xx	0	0x	VDUP (general-purpose register)
1xx	0	1x	UNALLOCATED

System register 32-bit move

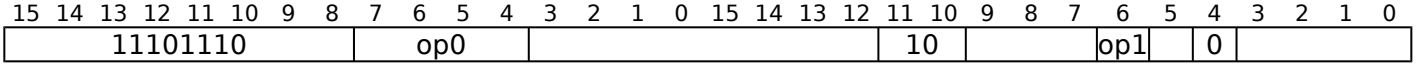
These instructions are under Advanced SIMD and System register 32-bit move.



Decode fields L	Instruction Details
0	MCR
1	MRC

Floating-point data-processing

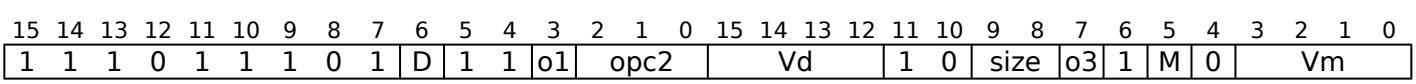
These instructions are under System register access, Advanced SIMD, and floating-point.



Decode fields op0	op1	Instruction details
1x11	1	Floating-point data-processing (two registers)
1x11	0	Floating-point move immediate
!= 1x11		Floating-point data-processing (three registers)

Floating-point data-processing (two registers)

These instructions are under Floating-point data-processing.



o1	Decode fields opc2	size	o3	Instruction Details	Feature
		00		UNALLOCATED	-
0	000	01	0	UNALLOCATED	-
0	000	01	1	VABS — half-precision scalar	FEAT_FP16
0	000	10	0	VMOV (register) — single-precision scalar	-
0	000	10	1	VABS — single-precision scalar	-
0	000	11	0	VMOV (register) — double-precision scalar	-
0	000	11	1	VABS — double-precision scalar	-
0	001	01	0	VNEG — half-precision scalar	FEAT_FP16
0	001	01	1	VSQRT — half-precision scalar	FEAT_FP16
0	001	10	0	VNEG — single-precision scalar	-
0	001	10	1	VSQRT — single-precision scalar	-
0	001	11	0	VNEG — double-precision scalar	-
0	001	11	1	VSQRT — double-precision scalar	-
0	010	01		UNALLOCATED	-
0	010	10	0	VCVTB — half-precision to single-precision	-
0	010	10	1	VCVTT — half-precision to single-precision	-
0	010	11	0	VCVTB — half-precision to double-precision	-
0	010	11	1	VCVTT — half-precision to double-precision	-
0	011	01	0	VCVTB (BFloat16)	FEAT_AA32BF16
0	011	01	1	VCVTT (BFloat16)	FEAT_AA32BF16

Decode fields				Instruction Details	Feature
o1	opc2	size	o3		
0	011	10	0	VCVTB — single-precision to half-precision	-
0	011	10	1	VCVTT — single-precision to half-precision	-
0	011	11	0	VCVTB — double-precision to half-precision	-
0	011	11	1	VCVTT — double-precision to half-precision	-
0	100	01	0	VCMP	FEAT_FP16
0	100	01	1	VCMPE	FEAT_FP16
0	100	10	0	VCMP	-
0	100	10	1	VCMPE	-
0	100	11	0	VCMP	-
0	100	11	1	VCMPE	-
0	101	01	0	VCMP	FEAT_FP16
0	101	01	1	VCMPE	FEAT_FP16
0	101	10	0	VCMP	-
0	101	10	1	VCMPE	-
0	101	11	0	VCMP	-
0	101	11	1	VCMPE	-
0	110	01	0	VRINTR — half-precision scalar	FEAT_FP16
0	110	01	1	VRINTZ (floating-point) — half-precision scalar	FEAT_FP16
0	110	10	0	VRINTR — single-precision scalar	-
0	110	10	1	VRINTZ (floating-point) — single-precision scalar	-
0	110	11	0	VRINTR — double-precision scalar	-
0	110	11	1	VRINTZ (floating-point) — double-precision scalar	-
0	111	01	0	VRINTX (floating-point) — half-precision scalar	FEAT_FP16
0	111	01	1	UNALLOCATED	-
0	111	10	0	VRINTX (floating-point) — single-precision scalar	-
0	111	10	1	VCVT (between double-precision and single-precision) — single-precision to double-precision	-
0	111	11	0	VRINTX (floating-point) — double-precision scalar	-
0	111	11	1	VCVT (between double-precision and single-precision) — double-precision to single-precision	-
1	000	01		VCVT (integer to floating-point, floating-point) — half-precision scalar	FEAT_FP16
1	000	10		VCVT (integer to floating-point, floating-point) — single-precision scalar	-
1	000	11		VCVT (integer to floating-point, floating-point) — double-precision scalar	-
1	001	01		UNALLOCATED	-
1	001	10		UNALLOCATED	-
1	001	11	0	UNALLOCATED	-
1	001	11	1	VJCVT	FEAT_JSCVT
1	01x	01		VCVT (between floating-point and fixed-point, floating-point)	FEAT_FP16
1	01x	10		VCVT (between floating-point and fixed-point, floating-point)	-
1	01x	11		VCVT (between floating-point and fixed-point, floating-point)	-
1	100	01	0	VCVTR	FEAT_FP16
1	100	01	1	VCVT (floating-point to integer, floating-point)	FEAT_FP16
1	100	10	0	VCVTR	-
1	100	10	1	VCVT (floating-point to integer, floating-point)	-
1	100	11	0	VCVTR	-
1	100	11	1	VCVT (floating-point to integer, floating-point)	-

Decode fields o1	opc2	size	o3	Instruction Details	Feature
1	101	01	0	VCVTR	FEAT_FP16
1	101	01	1	VCVT (floating-point to integer, floating-point)	FEAT_FP16
1	101	10	0	VCVTR	-
1	101	10	1	VCVT (floating-point to integer, floating-point)	-
1	101	11	0	VCVTR	-
1	101	11	1	VCVT (floating-point to integer, floating-point)	-
1	11x	01		VCVT (between floating-point and fixed-point, floating-point)	FEAT_FP16
1	11x	10		VCVT (between floating-point and fixed-point, floating-point)	-
1	11x	11		VCVT (between floating-point and fixed-point, floating-point)	-

Floating-point move immediate

These instructions are under Floating-point data-processing.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	1	D	1	1	imm4H				Vd				1	0	size		(0)	0	(0)	0	imm4L			

Decode fields size	Instruction Details	Feature
00	UNALLOCATED	-
01	VMOV (immediate) — half-precision scalar	FEAT_FP16
10	VMOV (immediate) — single-precision scalar	-
11	VMOV (immediate) — double-precision scalar	-

Floating-point data-processing (three registers)

These instructions are under Floating-point data-processing.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	o0	D	o1		Vn				Vd				1	0	size		N	o2	M	0	Vm			

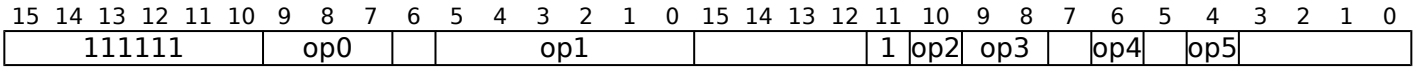
The following constraints also apply to this encoding: o0:D:o1 != 1x11

Decode fields o0:o1	size	o2	Instruction Details	Feature
!= 111	00		UNALLOCATED	-
000	01	0	VMLA (floating-point) — half-precision scalar	FEAT_FP16
000	01	1	VMLS (floating-point) — half-precision scalar	FEAT_FP16
000	10	0	VMLA (floating-point) — single-precision scalar	-
000	10	1	VMLS (floating-point) — single-precision scalar	-
000	11	0	VMLA (floating-point) — double-precision scalar	-
000	11	1	VMLS (floating-point) — double-precision scalar	-
001	01	0	VNMLS — half-precision scalar	FEAT_FP16
001	01	1	VNMLA — half-precision scalar	FEAT_FP16
001	10	0	VNMLS — single-precision scalar	-
001	10	1	VNMLA — single-precision scalar	-
001	11	0	VNMLS — double-precision scalar	-
001	11	1	VNMLA — double-precision scalar	-
010	01	0	VMUL (floating-point) — half-precision scalar	FEAT_FP16
010	01	1	VNMUL — half-precision scalar	FEAT_FP16

Decode fields			Instruction Details	Feature
o0:o1	size	o2		
010	10	0	VMUL (floating-point) — single-precision scalar	-
010	10	1	VNMUL — single-precision scalar	-
010	11	0	VMUL (floating-point) — double-precision scalar	-
010	11	1	VNMUL — double-precision scalar	-
011	01	0	VADD (floating-point) — half-precision scalar	FEAT_FP16
011	01	1	VSUB (floating-point) — half-precision scalar	FEAT_FP16
011	10	0	VADD (floating-point) — single-precision scalar	-
011	10	1	VSUB (floating-point) — single-precision scalar	-
011	11	0	VADD (floating-point) — double-precision scalar	-
011	11	1	VSUB (floating-point) — double-precision scalar	-
100	01	0	VDIV — half-precision scalar	FEAT_FP16
100	10	0	VDIV — single-precision scalar	-
100	11	0	VDIV — double-precision scalar	-
101	01	0	VFNMS — half-precision scalar	FEAT_FP16
101	01	1	VFNMA — half-precision scalar	FEAT_FP16
101	10	0	VFNMS — single-precision scalar	-
101	10	1	VFNMA — single-precision scalar	-
101	11	0	VFNMS — double-precision scalar	-
101	11	1	VFNMA — double-precision scalar	-
110	01	0	VFMA — half-precision scalar	FEAT_FP16
110	01	1	VFMS — half-precision scalar	FEAT_FP16
110	10	0	VFMA — single-precision scalar	-
110	10	1	VFMS — single-precision scalar	-
110	11	0	VFMA — double-precision scalar	-
110	11	1	VFMS — double-precision scalar	-

Additional Advanced SIMD and floating-point instructions

These instructions are under System register access, Advanced SIMD, and floating-point.

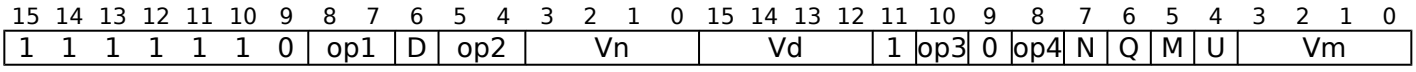


The following constraints also apply to this encoding: op0<2:1> != 11

Decode fields						Instruction details
op0	op1	op2	op3	op4	op5	
0xx			0x			Advanced SIMD three registers of the same length extension
100		0	!= 00	0	0	Floating-point conditional select
101	00xxxx	0	!= 00		0	Floating-point minNum/maxNum
101	110000	0	!= 00	1	0	Floating-point extraction and insertion
101	111xxx	0	!= 00	1	0	Floating-point directed convert to integer
10x		0	00			Advanced SIMD and floating-point multiply with accumulate
10x		1	0x			Advanced SIMD and floating-point dot product

Advanced SIMD three registers of the same length extension

These instructions are under Additional Advanced SIMD and floating-point instructions.



Decode fields						Instruction Details	Feature
op1	op2	op3	op4	Q	U		
x1	0x	0	0	0	0	VCADD — 64-bit SIMD vector	FEAT_FCMA
x1	0x	0	0	0	1	UNALLOCATED	-
x1	0x	0	0	1	0	VCADD — 128-bit SIMD vector	FEAT_FCMA
x1	0x	0	0	1	1	UNALLOCATED	-
00	0x	0	0			UNALLOCATED	-
00	0x	0	1			UNALLOCATED	-
00	00	1	0	0	0	UNALLOCATED	-
00	00	1	0	0	1	UNALLOCATED	-
00	00	1	0	1	0	VMMLA	FEAT_AA32BF16
00	00	1	0	1	1	UNALLOCATED	-
00	00	1	1	0	0	VDOT (vector) — 64-bit SIMD vector	FEAT_AA32BF16
00	00	1	1	0	1	UNALLOCATED	-
00	00	1	1	1	0	VDOT (vector) — 128-bit SIMD vector	FEAT_AA32BF16
00	00	1	1	1	1	UNALLOCATED	-
00	01	1	0			UNALLOCATED	-
00	01	1	1			UNALLOCATED	-
00	10	0	0		1	VFMAL (vector)	FEAT_FHM
00	10	0	1			UNALLOCATED	-
00	10	1	0	0		UNALLOCATED	-
00	10	1	0	1	0	VSMMLA	FEAT_AA32I8MM
00	10	1	0	1	1	VUMMLA	FEAT_AA32I8MM
00	10	1	1	0	0	VSDOT (vector) — 64-bit SIMD vector	FEAT_DotProd
00	10	1	1	0	1	VUDOT (vector) — 64-bit SIMD vector	FEAT_DotProd
00	10	1	1	1	0	VSDOT (vector) — 128-bit SIMD vector	FEAT_DotProd
00	10	1	1	1	1	VUDOT (vector) — 128-bit SIMD vector	FEAT_DotProd
00	11	0	0		1	VFMAB, VFMAT (BFloat16, vector)	FEAT_AA32BF16
00	11	0	1			UNALLOCATED	-
00	11	1	0			UNALLOCATED	-
00	11	1	1			UNALLOCATED	-
01	10	0	0		1	VFMSL (vector)	FEAT_FHM
01	10	0	1			UNALLOCATED	-
01	10	1	0	0		UNALLOCATED	-
01	10	1	0	1	0	VUSMMLA	FEAT_AA32I8MM
01	10	1	0	1	1	UNALLOCATED	-
01	10	1	1	0	0	VUSDOT (vector) — 64-bit SIMD vector	FEAT_AA32I8MM
01	10	1	1		1	UNALLOCATED	-
01	10	1	1	1	0	VUSDOT (vector) — 128-bit SIMD vector	FEAT_AA32I8MM
01	11	0	1			UNALLOCATED	-
01	11	1	0			UNALLOCATED	-
01	11	1	1			UNALLOCATED	-
	1x	0	0		0	VCMLA	FEAT_FCMA
10	11	0	1			UNALLOCATED	-

Decode fields						Instruction Details	Feature
op1	op2	op3	op4	Q	U		
10	11	1	0			UNALLOCATED	-
10	11	1	1			UNALLOCATED	-
11	11	0	1			UNALLOCATED	-
11	11	1	0			UNALLOCATED	-
11	11	1	1			UNALLOCATED	-

Floating-point conditional select

These instructions are under Additional Advanced SIMD and floating-point instructions.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0	0	D	cc		Vn				Vd				1	0	!= 00		N	0	M	0	Vm			
																						size									

The following constraints also apply to this encoding: size != 00 && size != 00

Decode fields	Instruction Details	Feature
size		
01	VSELEQ, VSELGE, VSELGT, VSELVS — half-precision scalar	FEAT_FP16
10	VSELEQ, VSELGE, VSELGT, VSELVS — single-precision scalar	-
11	VSELEQ, VSELGE, VSELGT, VSELVS — double-precision scalar	-

Floating-point minNum/maxNum

These instructions are under Additional Advanced SIMD and floating-point instructions.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0	1	D	0	0	Vn				Vd				1	0	!= 00		N	op	M	0	Vm			
																						size									

The following constraints also apply to this encoding: size != 00 && size != 00

Decode fields		Instruction Details	Feature
size	op		
01	0	VMAXNM — half-precision scalar	FEAT_FP16
01	1	VMINNM — half-precision scalar	FEAT_FP16
10	0	VMAXNM — single-precision scalar	-
10	1	VMINNM — single-precision scalar	-
11	0	VMAXNM — double-precision scalar	-
11	1	VMINNM — double-precision scalar	-

Floating-point extraction and insertion

These instructions are under Additional Advanced SIMD and floating-point instructions.

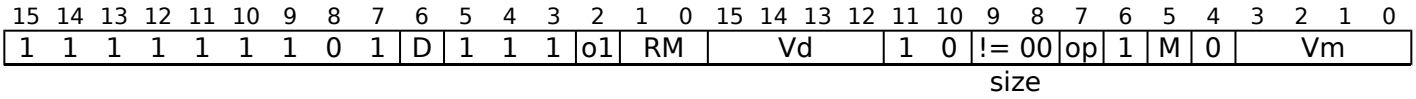
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0	1	D	1	1	0	0	0	0	Vd				1	0	!= 00		op	1	M	0	Vm			
																						size									

The following constraints also apply to this encoding: size != 00 && size != 00

Decode fields size	op	Instruction Details	Feature
01		UNALLOCATED	-
10	0	VMOVX	FEAT_FP16
10	1	VINS	FEAT_FP16
11		UNALLOCATED	-

Floating-point directed convert to integer

These instructions are under Additional Advanced SIMD and floating-point instructions.

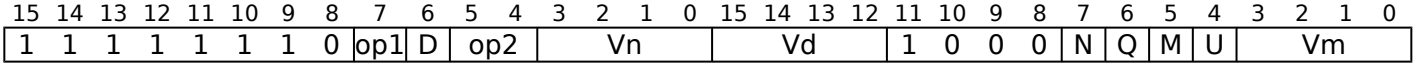


The following constraints also apply to this encoding: size != 00 && size != 00

o1	Decode fields RM	size	op	Instruction Details	Feature
0		!= 00	1	UNALLOCATED	-
0	00	01	0	VRINTA (floating-point) — half-precision scalar	FEAT_FP16
0	00	10	0	VRINTA (floating-point) — single-precision scalar	-
0	00	11	0	VRINTA (floating-point) — double-precision scalar	-
0	01	01	0	VRINTN (floating-point) — half-precision scalar	FEAT_FP16
0	01	10	0	VRINTN (floating-point) — single-precision scalar	-
0	01	11	0	VRINTN (floating-point) — double-precision scalar	-
0	10	01	0	VRINTP (floating-point) — half-precision scalar	FEAT_FP16
0	10	10	0	VRINTP (floating-point) — single-precision scalar	-
0	10	11	0	VRINTP (floating-point) — double-precision scalar	-
0	11	01	0	VRINTM (floating-point) — half-precision scalar	FEAT_FP16
0	11	10	0	VRINTM (floating-point) — single-precision scalar	-
0	11	11	0	VRINTM (floating-point) — double-precision scalar	-
1	00	01		VCVTA (floating-point) — half-precision scalar	FEAT_FP16
1	00	10		VCVTA (floating-point) — single-precision scalar	-
1	00	11		VCVTA (floating-point) — double-precision scalar	-
1	01	01		VCVTN (floating-point) — half-precision scalar	FEAT_FP16
1	01	10		VCVTN (floating-point) — single-precision scalar	-
1	01	11		VCVTN (floating-point) — double-precision scalar	-
1	10	01		VCVTP (floating-point) — half-precision scalar	FEAT_FP16
1	10	10		VCVTP (floating-point) — single-precision scalar	-
1	10	11		VCVTP (floating-point) — double-precision scalar	-
1	11	01		VCVTM (floating-point) — half-precision scalar	FEAT_FP16
1	11	10		VCVTM (floating-point) — single-precision scalar	-
1	11	11		VCVTM (floating-point) — double-precision scalar	-

Advanced SIMD and floating-point multiply with accumulate

These instructions are under Additional Advanced SIMD and floating-point instructions.



Decode fields op1	op2	Q	U	Instruction Details	Feature
0			0	VCMLA (by element) — 128-bit SIMD vector of half-precision floating-point	FEAT_FCMA
0	00		1	VFMAL (by scalar)	FEAT_FHM
0	01		1	VFMSL (by scalar)	FEAT_FHM
0	10		1	UNALLOCATED	-
0	11		1	VFMAB, VFMAT (BFloat16, by scalar)	FEAT_AA32BF16
1		0	0	VCMLA (by element) — 64-bit SIMD vector of single-precision floating-point	FEAT_FCMA
1			1	UNALLOCATED	-
1		1	0	VCMLA (by element) — 128-bit SIMD vector of single-precision floating-point	FEAT_FCMA

Advanced SIMD and floating-point dot product

These instructions are under 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 op1	op2	op4	Q	U	Instruction Details	Feature
0	00	0			UNALLOCATED	-
0	00	1	0	0	VDOT (by element) — 64-bit SIMD vector	FEAT_AA32BF16
0	00	1		1	UNALLOCATED	-
0	00	1	1	0	VDOT (by element) — 128-bit SIMD vector	FEAT_AA32BF16
0	01	0			UNALLOCATED	-
0	10	0			UNALLOCATED	-
0	10	1	0	0	VSDOT (by element) — 64-bit SIMD vector	FEAT_DotProd
0	10	1	0	1	VUDOT (by element) — 64-bit SIMD vector	FEAT_DotProd
0	10	1	1	0	VSDOT (by element) — 128-bit SIMD vector	FEAT_DotProd
0	10	1	1	1	VUDOT (by element) — 128-bit SIMD vector	FEAT_DotProd
0	11				UNALLOCATED	-
1		0			UNALLOCATED	-
1	00	1	0	0	VUSDOT (by element) — 64-bit SIMD vector	FEAT_AA32I8MM
1	00	1	0	1	VSUDOT (by element) — 64-bit SIMD vector	FEAT_AA32I8MM
1	00	1	1	0	VUSDOT (by element) — 128-bit SIMD vector	FEAT_AA32I8MM
1	00	1	1	1	VSUDOT (by element) — 128-bit SIMD vector	FEAT_AA32I8MM
1	01	1			UNALLOCATED	-
1	1x	1			UNALLOCATED	-

Load/store multiple

These instructions are under 32-bit.

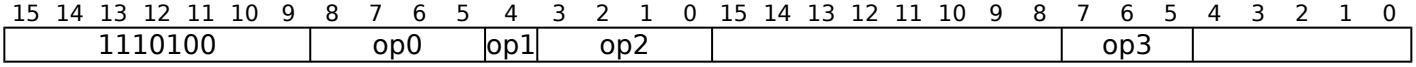
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	opc		0	W	L	Rn				P	M	register_list													

Decode fields opc	L	Instruction Details
00	0	SRS, SRSDA, SRSDB, SRSIA, SRSIB — T1
00	1	RFE, RFEDA, RFEDB, RFEIA, RFEIB — T1

Decode fields opc	L	Instruction Details
01	0	STM, STMIA, STMEA
01	1	LDM, LDMIA, LDMFD
10	0	STMDB, STMFD
10	1	LDMDB, LDMEA
11	0	SRS, SRSDA, SRSDB, SRSIA, SRSIB — T2
11	1	RFE, RFEDA, RFEDB, RFEIA, RFEIB — T2

Load/store dual, load/store exclusive, load-acquire/store-release, and table branch

These instructions are under 32-bit.

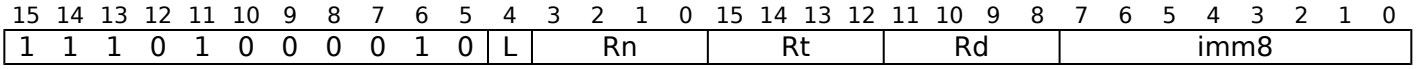


The following constraints also apply to this encoding: op0<1> == 1

op0	Decode fields op1	op2	op3	Instruction details
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

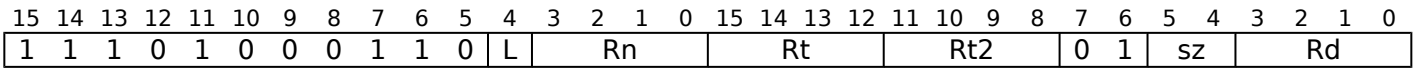
These instructions are under Load/store dual, load/store exclusive, load-acquire/store-release, and table branch.



Decode fields L	Instruction Details
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.

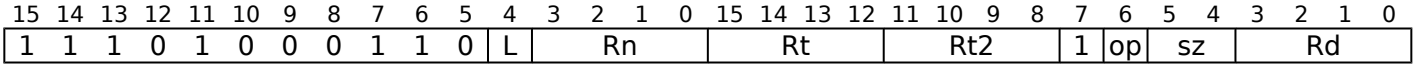


Decode fields L	sz	Instruction Details
0	00	STREXB
0	01	STREXH

Decode fields L	sz	Instruction Details
0	10	UNALLOCATED
0	11	STREXD
1	00	LDREXB
1	01	LDREXH
1	10	UNALLOCATED
1	11	LDREXD

Load-acquire / Store-release

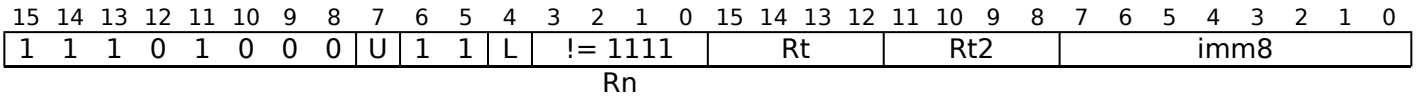
These instructions are under Load/store dual, load/store exclusive, load-acquire/store-release, and table branch.



Decode fields L	op	sz	Instruction Details
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.

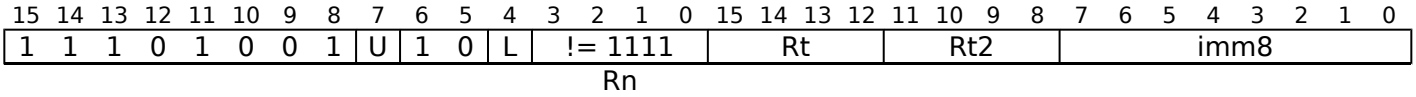


The following constraints also apply to this encoding: Rn != 1111 && Rn != 1111

Decode fields L	Instruction Details
0	STRD (immediate)
1	LDRD (immediate)

Load/store dual (immediate)

These instructions are under Load/store dual, load/store exclusive, load-acquire/store-release, and table branch.

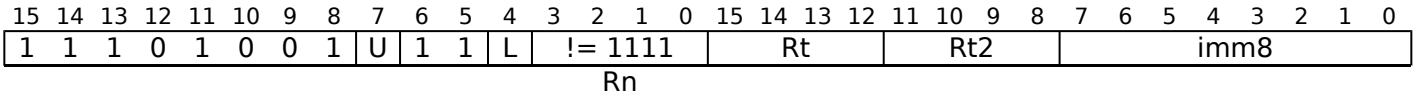


The following constraints also apply to this encoding: Rn != 1111 && Rn != 1111

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.

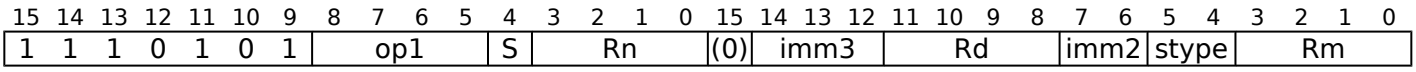


The following constraints also apply to this encoding: Rn != 1111 && Rn != 1111

Decode fields L	Instruction Details
0	STRD (immediate)
1	LDRD (immediate)

Data-processing (shifted register)

These instructions are under 32-bit.



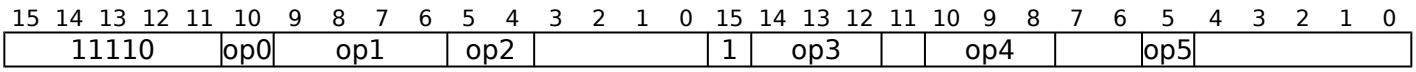
Decode fields op1	S	Rn	imm3:imm2:stype	Rd	Instruction Details
0000	0		!= 0000011		AND, ANDS (register) — AND, shift or rotate by value
0000	0		0000011		AND, ANDS (register) — AND, rotate right with extend
0000	1		!= 0000011	!= 1111	AND, ANDS (register) — ANDS, shift or rotate by value
0000	1		!= 0000011	1111	TST (register) — shift or rotate by value
0000	1		0000011	!= 1111	AND, ANDS (register) — ANDS, rotate right with extend
0000	1		0000011	1111	TST (register) — rotate right with extend
0001			!= 0000011		BIC, BICS (register) — shift or rotate by value
0001			0000011		BIC, BICS (register) — rotate right with extend
0010	0	!= 1111	!= 0000011		ORR, ORRS (register) — ORR, shift or rotate by value
0010	0	!= 1111	0000011		ORR, ORRS (register) — ORR, rotate right with extend
0010	0	1111	!= 0000011		MOV, MOVS (register) — MOV, shift or rotate by value
0010	0	1111	0000011		MOV, MOVS (register) — MOV, rotate right with extend

Decode fields					Instruction Details
op1	S	Rn	imm3:imm2:stype	Rd	
0010	1	!= 1111	!= 0000011		ORR, ORRS (register) — ORRS, shift or rotate by value
0010	1	!= 1111	0000011		ORR, ORRS (register) — ORRS, rotate right with extend
0010	1	1111	!= 0000011		MOV, MOVS (register) — MOVS, shift or rotate by value
0010	1	1111	0000011		MOV, MOVS (register) — MOVS, rotate right with extend
0011	0	!= 1111	!= 0000011		ORN, ORNS (register) — ORN, shift or rotate by value
0011	0	!= 1111	0000011		ORN, ORNS (register) — ORN, rotate right with extend
0011	0	1111	!= 0000011		MVN, MVNS (register) — MVN, shift or rotate by value
0011	0	1111	0000011		MVN, MVNS (register) — MVN, rotate right with extend
0011	1	!= 1111	!= 0000011		ORN, ORNS (register) — ORNS, shift or rotate by value
0011	1	!= 1111	0000011		ORN, ORNS (register) — ORNS, rotate right with extend
0011	1	1111	!= 0000011		MVN, MVNS (register) — MVNS, shift or rotate by value
0011	1	1111	0000011		MVN, MVNS (register) — MVNS, rotate right with extend
0100	0		!= 0000011		EOR, EORS (register) — EOR, shift or rotate by value
0100	0		0000011		EOR, EORS (register) — EOR, rotate right with extend
0100	1		!= 0000011	!= 1111	EOR, EORS (register) — EORS, shift or rotate by value
0100	1		!= 0000011	1111	TEQ (register) — shift or rotate by value
0100	1		0000011	!= 1111	EOR, EORS (register) — EORS, rotate right with extend
0100	1		0000011	1111	TEQ (register) — rotate right with extend
0101					UNALLOCATED
0110	0		xxxxx00		PKHBT, PKHTB — PKHBT
0110	0		xxxxx01		UNALLOCATED
0110	0		xxxxx10		PKHBT, PKHTB — PKHTB
0110	0		xxxxx11		UNALLOCATED
0111					UNALLOCATED
1000	0	!= 1101	!= 0000011		ADD, ADDS (register) — ADD, shift or rotate by value
1000	0	!= 1101	0000011		ADD, ADDS (register) — ADD, rotate right with extend
1000	0	1101	!= 0000011		ADD, ADDS (SP plus register) — ADD, shift or rotate by value
1000	0	1101	0000011		ADD, ADDS (SP plus register) — ADD, rotate right with extend
1000	1		!= 0000011	1111	CMN (register) — shift or rotate by value
1000	1	!= 1101	!= 0000011	!= 1111	ADD, ADDS (register) — ADDS, shift or rotate by value

Decode fields					Instruction Details
op1	S	Rn	imm3:imm2:stype	Rd	
1000	1	!= 1101	0000011	!= 1111	ADD, ADDS (register) — ADDS, rotate right with extend
1000	1		0000011	1111	CMN (register) — rotate right with extend
1000	1	1101	!= 0000011	!= 1111	ADD, ADDS (SP plus register) — ADDS, shift or rotate by value
1000	1	1101	0000011	!= 1111	ADD, ADDS (SP plus register) — ADDS, rotate right with extend
1001					UNALLOCATED
1010			!= 0000011		ADC, ADCS (register) — shift or rotate by value
1010			0000011		ADC, ADCS (register) — rotate right with extend
1011			!= 0000011		SBC, SBCS (register) — shift or rotate by value
1011			0000011		SBC, SBCS (register) — rotate right with extend
1100					UNALLOCATED
1101	0	!= 1101	!= 0000011		SUB, SUBS (register) — SUB, shift or rotate by value
1101	0	!= 1101	0000011		SUB, SUBS (register) — SUB, rotate right with extend
1101	0	1101	!= 0000011		SUB, SUBS (SP minus register) — SUB, shift or rotate by value
1101	0	1101	0000011		SUB, SUBS (SP minus register) — SUB, rotate right with extend
1101	1		!= 0000011	1111	CMP (register) — shift or rotate by value
1101	1	!= 1101	!= 0000011	!= 1111	SUB, SUBS (register) — SUBS, shift or rotate by value
1101	1	!= 1101	0000011	!= 1111	SUB, SUBS (register) — SUBS, rotate right with extend
1101	1		0000011	1111	CMP (register) — rotate right with extend
1101	1	1101	!= 0000011	!= 1111	SUB, SUBS (SP minus register) — SUBS, shift or rotate by value
1101	1	1101	0000011	!= 1111	SUB, SUBS (SP minus register) — SUBS, rotate right with extend
1110			!= 0000011		RSB, RSBS (register) — shift or rotate by value
1110			0000011		RSB, RSBS (register) — rotate right with extend
1111					UNALLOCATED

Branches and miscellaneous control

These instructions are under 32-bit.



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	Feature
0000	0000	NOP	-
0000	0001	YIELD	-
0000	0010	WFE	-
0000	0011	WFI	-
0000	0100	SEV	-
0000	0101	SEVL	-
0000	011x	Reserved hint, behaves as NOP	-
0000	1xxx	Reserved hint, behaves as NOP	-
0001	0000	ESB	FEAT_RAS
0001	0001	Reserved hint, behaves as NOP	-
0001	0010	TSB CSYNC	FEAT_TRF
0001	0011	Reserved hint, behaves as NOP	-
0001	0100	CSDB	-
0001	0101	Reserved hint, behaves as NOP	-
0001	011x	Reserved hint, behaves as NOP	-
0001	1xxx	Reserved hint, behaves as NOP	-
001x		Reserved hint, behaves as NOP	-
01xx		Reserved hint, behaves as NOP	-
10xx		Reserved hint, behaves as NOP	-
110x		Reserved hint, behaves as NOP	-
1110		Reserved hint, behaves as NOP	-
1111		DBG	-

Change processor state

These instructions are under Branches and miscellaneous control.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	0	(1)	(1)	(1)	(1)	1	0	(0)	0	(0)	imod		M	A	I	F		mode			

The following constraints also apply to this encoding: imod:M != 000

Decode fields imod	M	Instruction Details
00	1	CPS, CPSID, CPSIE — change mode
01		UNALLOCATED
10		CPS, CPSID, CPSIE — interrupt enable and change mode
11		CPS, CPSID, CPSIE — interrupt disable and change mode

Miscellaneous system

These instructions are under Branches and miscellaneous control.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	1	(1)	(1)	(1)	(1)	1	0	(0)	0	(1)	(1)	(1)	(1)	opc				option			

Decode fields opc	option	Instruction Details
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 Rn:imm8	Instruction Details
!= 111000000000	SUB, SUBS (immediate)
111000000000	ERET

DCPS

These instructions are under Branches and miscellaneous control.

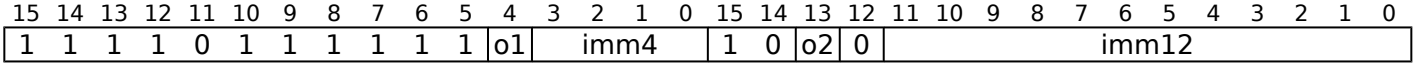
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	1	1	1	0	0	0	imm4				1	0	0	0	imm10										opt	

imm4	Decode fields imm10	opt	Instruction Details
!= 1111			UNALLOCATED
1111	!= 0000000000		UNALLOCATED
1111	0000000000	00	UNALLOCATED
1111	0000000000	01	DCPS1
1111	0000000000	10	DCPS2

Decode fields			Instruction Details
imm4	imm10	opt	
1111	0000000000	11	DCPS3

Exception generation

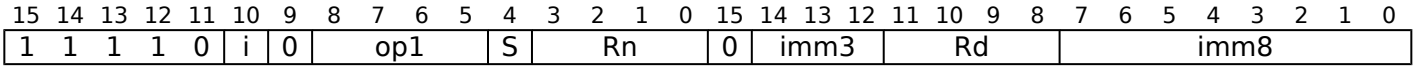
These instructions are under Branches and miscellaneous control.



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.



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 op1	S	Rn	Rd	Instruction Details
1100				UNALLOCATED
1101	0	!= 1101		SUB, SUBS (immediate) — SUB
1101	0	1101		SUB, SUBS (SP minus immediate) — SUB
1101	1	!= 1101	!= 1111	SUB, SUBS (immediate) — SUBS
1101	1	1101	!= 1111	SUB, SUBS (SP minus immediate) — SUBS
1101	1		1111	CMP (immediate)
1110				RSB, RSBS (immediate)
1111				UNALLOCATED

Data-processing (plain binary immediate)

These instructions are under 32-bit.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
11110						1	op0		op1		0					0																

Decode fields op0	op1	Instruction details
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 o1	o2	Rn	Instruction Details
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)

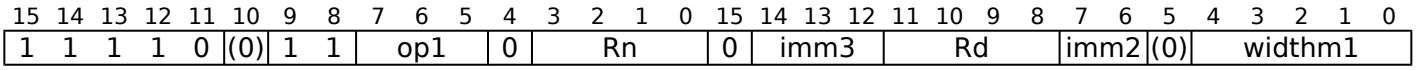
These instructions are under Data-processing (plain binary immediate).

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 o1	Instruction Details
0	MOV, MOVS (immediate)
1	MOVT

Saturate, Bitfield

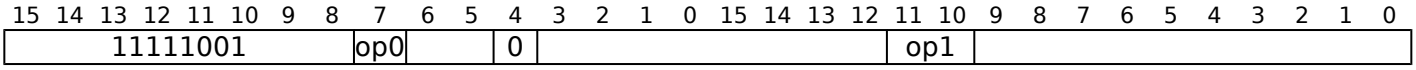
These instructions are under Data-processing (plain binary immediate).



op1	Decode fields Rn	imm3:imm2	Instruction Details
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

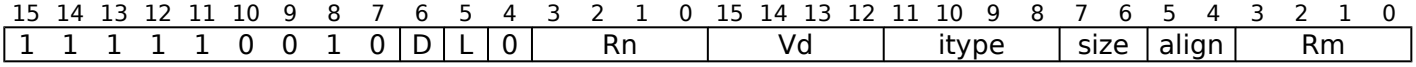
These instructions are under 32-bit.



op0	Decode fields op1	Instruction details
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.



L	Decode fields itype	Rm	Instruction Details
0	000x	!= 11x1	VST4 (multiple 4-element structures)
0	000x	1101	VST4 (multiple 4-element structures)
0	000x	1111	VST4 (multiple 4-element structures)
0	0010	!= 11x1	VST1 (multiple single elements)
0	0010	1101	VST1 (multiple single elements)
0	0010	1111	VST1 (multiple single elements)
0	0011	!= 11x1	VST2 (multiple 2-element structures)
0	0011	1101	VST2 (multiple 2-element structures)
0	0011	1111	VST2 (multiple 2-element structures)
0	010x	!= 11x1	VST3 (multiple 3-element structures)
0	010x	1101	VST3 (multiple 3-element structures)

Decode fields			Instruction Details
L	itype	Rm	
0	010x	1111	VST3 (multiple 3-element structures)
0	0110	!= 11x1	VST1 (multiple single elements)
0	0110	1101	VST1 (multiple single elements)
0	0110	1111	VST1 (multiple single elements)
0	0111	!= 11x1	VST1 (multiple single elements)
0	0111	1101	VST1 (multiple single elements)
0	0111	1111	VST1 (multiple single elements)
0	100x	!= 11x1	VST2 (multiple 2-element structures)
0	100x	1101	VST2 (multiple 2-element structures)
0	100x	1111	VST2 (multiple 2-element structures)
0	1010	!= 11x1	VST1 (multiple single elements)
0	1010	1101	VST1 (multiple single elements)
0	1010	1111	VST1 (multiple single elements)
1	000x	!= 11x1	VLD4 (multiple 4-element structures)
1	000x	1101	VLD4 (multiple 4-element structures)
1	000x	1111	VLD4 (multiple 4-element structures)
1	0010	!= 11x1	VLD1 (multiple single elements)
1	0010	1101	VLD1 (multiple single elements)
1	0010	1111	VLD1 (multiple single elements)
1	0011	!= 11x1	VLD2 (multiple 2-element structures)
1	0011	1101	VLD2 (multiple 2-element structures)
1	0011	1111	VLD2 (multiple 2-element structures)
1	010x	!= 11x1	VLD3 (multiple 3-element structures)
1	010x	1101	VLD3 (multiple 3-element structures)
1	010x	1111	VLD3 (multiple 3-element structures)
	1011		UNALLOCATED
1	0110	!= 11x1	VLD1 (multiple single elements)
1	0110	1101	VLD1 (multiple single elements)
1	0110	1111	VLD1 (multiple single elements)
1	0111	!= 11x1	VLD1 (multiple single elements)
1	0111	1101	VLD1 (multiple single elements)
1	0111	1111	VLD1 (multiple single elements)
	11xx		UNALLOCATED
1	100x	!= 11x1	VLD2 (multiple 2-element structures)
1	100x	1101	VLD2 (multiple 2-element structures)
1	100x	1111	VLD2 (multiple 2-element structures)
1	1010	!= 11x1	VLD1 (multiple single elements)
1	1010	1101	VLD1 (multiple single elements)
1	1010	1111	VLD1 (multiple single elements)

Advanced SIMD load single structure to all lanes

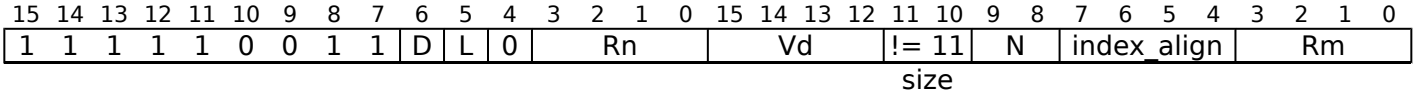
These instructions are under Advanced SIMD element or structure load/store.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	1	D	L	0	Rn				Vd				1	1	N		size		T	a	Rm			

Decode fields				Instruction Details
L	N	a	Rm	
0				UNALLOCATED
1	00		!= 11x1	VLD1 (single element to all lanes)
1	00		1101	VLD1 (single element to all lanes)
1	00		1111	VLD1 (single element to all lanes)
1	01		!= 11x1	VLD2 (single 2-element structure to all lanes)
1	01		1101	VLD2 (single 2-element structure to all lanes)
1	01		1111	VLD2 (single 2-element structure to all lanes)
1	10	0	!= 11x1	VLD3 (single 3-element structure to all lanes)
1	10	0	1101	VLD3 (single 3-element structure to all lanes)
1	10	0	1111	VLD3 (single 3-element structure to all lanes)
1	10	1		UNALLOCATED
1	11		!= 11x1	VLD4 (single 4-element structure to all lanes)
1	11		1101	VLD4 (single 4-element structure to all lanes)
1	11		1111	VLD4 (single 4-element structure to all lanes)

Advanced SIMD load/store single structure to one lane

These instructions are under Advanced SIMD element or structure load/store.



The following constraints also apply to this encoding: size != 11 && size != 11

Decode fields				Instruction Details
L	size	N	Rm	
0	00	00	!= 11x1	VST1 (single element from one lane)
0	00	00	1101	VST1 (single element from one lane)
0	00	00	1111	VST1 (single element from one lane)
0	00	01	!= 11x1	VST2 (single 2-element structure from one lane)
0	00	01	1101	VST2 (single 2-element structure from one lane)
0	00	01	1111	VST2 (single 2-element structure from one lane)
0	00	10	!= 11x1	VST3 (single 3-element structure from one lane)
0	00	10	1101	VST3 (single 3-element structure from one lane)
0	00	10	1111	VST3 (single 3-element structure from one lane)
0	00	11	!= 11x1	VST4 (single 4-element structure from one lane)
0	00	11	1101	VST4 (single 4-element structure from one lane)
0	00	11	1111	VST4 (single 4-element structure from one lane)
0	01	00	!= 11x1	VST1 (single element from one lane)
0	01	00	1101	VST1 (single element from one lane)
0	01	00	1111	VST1 (single element from one lane)
0	01	01	!= 11x1	VST2 (single 2-element structure from one lane)
0	01	01	1101	VST2 (single 2-element structure from one lane)
0	01	01	1111	VST2 (single 2-element structure from one lane)
0	01	10	!= 11x1	VST3 (single 3-element structure from one lane)
0	01	10	1101	VST3 (single 3-element structure from one lane)
0	01	10	1111	VST3 (single 3-element structure from one lane)
0	01	11	!= 11x1	VST4 (single 4-element structure from one lane)

Decode fields				Instruction Details
L	size	N	Rm	
0	01	11	1101	VST4 (single 4-element structure from one lane)
0	01	11	1111	VST4 (single 4-element structure from one lane)
0	10	00	!= 11x1	VST1 (single element from one lane)
0	10	00	1101	VST1 (single element from one lane)
0	10	00	1111	VST1 (single element from one lane)
0	10	01	!= 11x1	VST2 (single 2-element structure from one lane)
0	10	01	1101	VST2 (single 2-element structure from one lane)
0	10	01	1111	VST2 (single 2-element structure from one lane)
0	10	10	!= 11x1	VST3 (single 3-element structure from one lane)
0	10	10	1101	VST3 (single 3-element structure from one lane)
0	10	10	1111	VST3 (single 3-element structure from one lane)
0	10	11	!= 11x1	VST4 (single 4-element structure from one lane)
0	10	11	1101	VST4 (single 4-element structure from one lane)
0	10	11	1111	VST4 (single 4-element structure from one lane)
1	00	00	!= 11x1	VLD1 (single element to one lane)
1	00	00	1101	VLD1 (single element to one lane)
1	00	00	1111	VLD1 (single element to one lane)
1	00	01	!= 11x1	VLD2 (single 2-element structure to one lane)
1	00	01	1101	VLD2 (single 2-element structure to one lane)
1	00	01	1111	VLD2 (single 2-element structure to one lane)
1	00	10	!= 11x1	VLD3 (single 3-element structure to one lane)
1	00	10	1101	VLD3 (single 3-element structure to one lane)
1	00	10	1111	VLD3 (single 3-element structure to one lane)
1	00	11	!= 11x1	VLD4 (single 4-element structure to one lane)
1	00	11	1101	VLD4 (single 4-element structure to one lane)
1	00	11	1111	VLD4 (single 4-element structure to one lane)
1	01	00	!= 11x1	VLD1 (single element to one lane)
1	01	00	1101	VLD1 (single element to one lane)
1	01	00	1111	VLD1 (single element to one lane)
1	01	01	!= 11x1	VLD2 (single 2-element structure to one lane)
1	01	01	1101	VLD2 (single 2-element structure to one lane)
1	01	01	1111	VLD2 (single 2-element structure to one lane)
1	01	10	!= 11x1	VLD3 (single 3-element structure to one lane)
1	01	10	1101	VLD3 (single 3-element structure to one lane)
1	01	10	1111	VLD3 (single 3-element structure to one lane)
1	01	11	!= 11x1	VLD4 (single 4-element structure to one lane)
1	01	11	1101	VLD4 (single 4-element structure to one lane)
1	01	11	1111	VLD4 (single 4-element structure to one lane)
1	10	00	!= 11x1	VLD1 (single element to one lane)
1	10	00	1101	VLD1 (single element to one lane)
1	10	00	1111	VLD1 (single element to one lane)
1	10	01	!= 11x1	VLD2 (single 2-element structure to one lane)
1	10	01	1101	VLD2 (single 2-element structure to one lane)
1	10	01	1111	VLD2 (single 2-element structure to one lane)
1	10	10	!= 11x1	VLD3 (single 3-element structure to one lane)
1	10	10	1101	VLD3 (single 3-element structure to one lane)
1	10	10	1111	VLD3 (single 3-element structure to one lane)

Decode fields				Instruction Details
L	size	N	Rm	
1	10	11	!= 11x1	VLD4 (single 4-element structure to one lane)
1	10	11	1101	VLD4 (single 4-element structure to one lane)
1	10	11	1111	VLD4 (single 4-element structure to one lane)

Load/store single

These instructions are under 32-bit.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1111100							op0				op1	op2								op3											

The following constraints also apply to this encoding: op0<1>:op1 != 10

Decode fields				Instruction details
op0	op1	op2	op3	
00		!= 1111	000000	Load/store, unsigned (register offset)
00		!= 1111	000001	UNALLOCATED
00		!= 1111	00001x	UNALLOCATED
00		!= 1111	0001xx	UNALLOCATED
00		!= 1111	001xxx	UNALLOCATED
00		!= 1111	01xxxx	UNALLOCATED
00		!= 1111	10x0xx	UNALLOCATED
00		!= 1111	10x1xx	Load/store, unsigned (immediate, post-indexed)
00		!= 1111	1100xx	Load/store, unsigned (negative immediate)
00		!= 1111	1110xx	Load/store, unsigned (unprivileged)
00		!= 1111	11x1xx	Load/store, unsigned (immediate, pre-indexed)
01		!= 1111		Load/store, unsigned (positive immediate)
0x		1111		Load, unsigned (literal)
10	1	!= 1111	000000	Load/store, signed (register offset)
10	1	!= 1111	000001	UNALLOCATED
10	1	!= 1111	00001x	UNALLOCATED
10	1	!= 1111	0001xx	UNALLOCATED
10	1	!= 1111	001xxx	UNALLOCATED
10	1	!= 1111	01xxxx	UNALLOCATED
10	1	!= 1111	10x0xx	UNALLOCATED
10	1	!= 1111	10x1xx	Load/store, signed (immediate, post-indexed)
10	1	!= 1111	1100xx	Load/store, signed (negative immediate)
10	1	!= 1111	1110xx	Load/store, signed (unprivileged)
10	1	!= 1111	11x1xx	Load/store, signed (immediate, pre-indexed)
11	1	!= 1111		Load/store, signed (positive immediate)
1x	1	1111		Load, signed (literal)

Load/store, unsigned (register offset)

These instructions are under Load/store single.

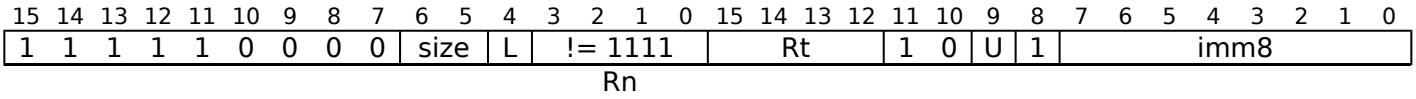
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	0	size		L	!= 1111				Rt				0	0	0	0	0	0	imm2		Rm			
Rn																															

The following constraints also apply to this encoding: Rn != 1111 && Rn != 1111

Decode fields size	L	Rt	Instruction Details
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.

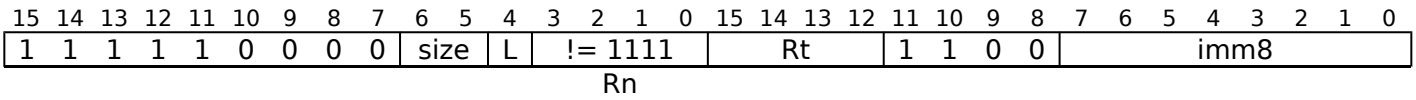


The following constraints also apply to this encoding: Rn != 1111 && Rn != 1111

Decode fields size	L	Instruction Details
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.

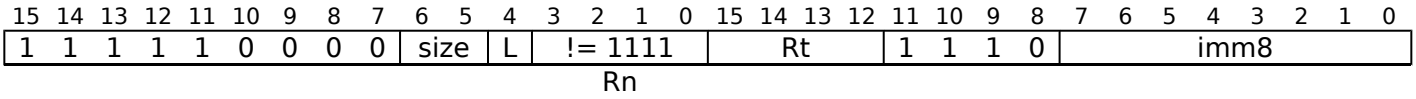


The following constraints also apply to this encoding: Rn != 1111 && Rn != 1111

Decode fields size	L	Rt	Instruction Details
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)
11			UNALLOCATED

Load/store, unsigned (unprivileged)

These instructions are under Load/store single.

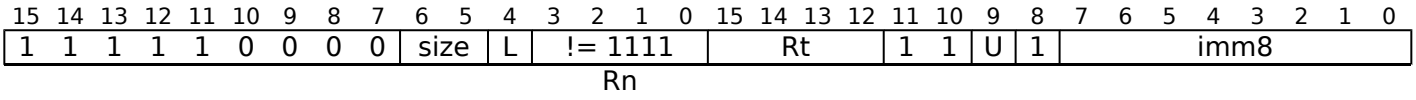


The following constraints also apply to this encoding: Rn != 1111 && Rn != 1111

Decode fields size	L	Instruction Details
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.

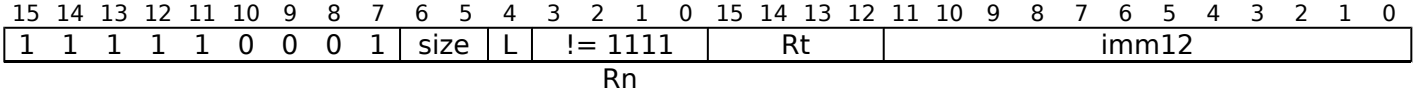


The following constraints also apply to this encoding: Rn != 1111 && Rn != 1111

Decode fields size	L	Instruction Details
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.



The following constraints also apply to this encoding: Rn != 1111 && Rn != 1111

Decode fields size	L	Rt	Instruction Details
00	0		STRB (immediate)
00	1	!= 1111	LDRB (immediate)
00	1	1111	PLD, PLDW (immediate) — preload read

Decode fields size	L	Rt	Instruction Details
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 size	L	Rt	Instruction Details
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 size	Rt	Instruction Details
00	!= 1111	LDRSB (register)
00	1111	PLI (register)
01	!= 1111	LDRSH (register)
01	1111	Reserved hint, behaves as NOP
1x		UNALLOCATED

Load/store, signed (immediate, post-indexed)

These instructions are under Load/store single.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	0	size		1	!= 1111				Rt				1	0	U	1	imm8							
Rn																															

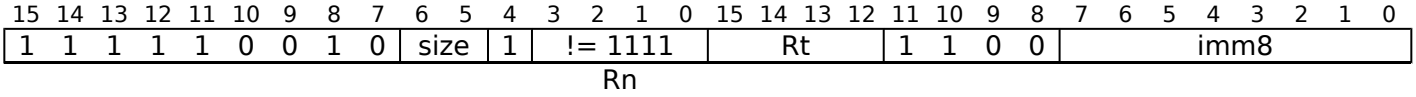
The following constraints also apply to this encoding: Rn != 1111 && Rn != 1111

Decode fields size	Instruction Details
00	LDRSB (immediate)

Decode fields size	Instruction Details
01	LDRSH (immediate)
1x	UNALLOCATED

Load/store, signed (negative immediate)

These instructions are under Load/store single.

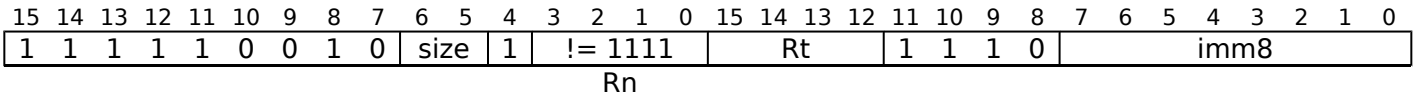


The following constraints also apply to this encoding: Rn != 1111 && Rn != 1111

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.

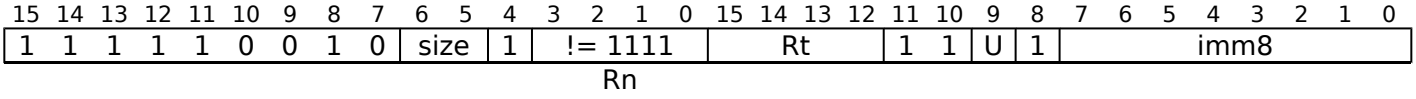


The following constraints also apply to this encoding: Rn != 1111 && Rn != 1111

Decode fields size	Instruction Details
00	LDRSBT
01	LDRSHT
1x	UNALLOCATED

Load/store, signed (immediate, pre-indexed)

These instructions are under Load/store single.

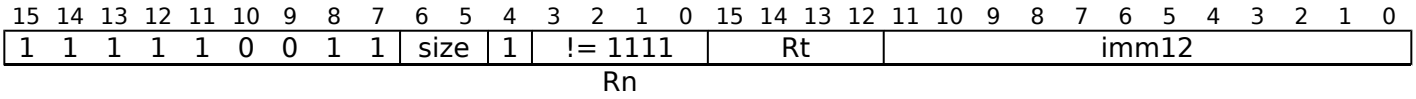


The following constraints also apply to this encoding: Rn != 1111 && Rn != 1111

Decode fields size	Instruction Details
00	LDRSB (immediate)
01	LDRSH (immediate)
1x	UNALLOCATED

Load/store, signed (positive immediate)

These instructions are under Load/store single.

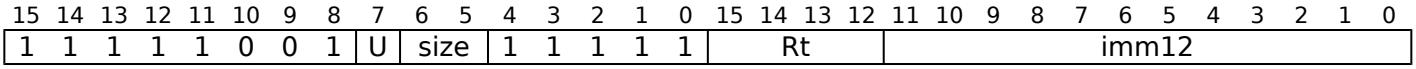


The following constraints also apply to this encoding: Rn != 1111 && Rn != 1111

Decode fields size	Rt	Instruction Details
00	!= 1111	LDRSB (immediate)
00	1111	PLI (immediate, literal)
01	!= 1111	LDRSH (immediate)
01	1111	Reserved hint, behaves as NOP

Load, signed (literal)

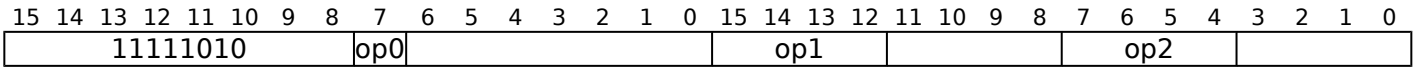
These instructions are under Load/store single.



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)

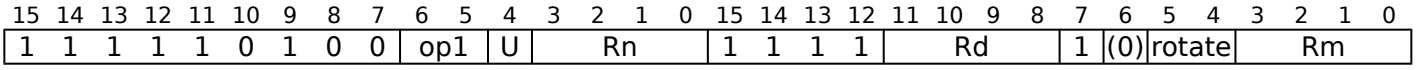
These instructions are under 32-bit.



Decode fields op0	op1	op2	Instruction details
0	1111	0000	MOV, MOVS (register-shifted register) — T2, Flag setting
0	1111	0001	UNALLOCATED
0	1111	001x	UNALLOCATED
0	1111	01xx	UNALLOCATED
0	1111	1xxx	Register extends
1	1111	0xxx	Parallel add-subtract
1	1111	10xx	Data-processing (two source registers)
1	1111	11xx	UNALLOCATED
	!= 1111		UNALLOCATED

Register extends

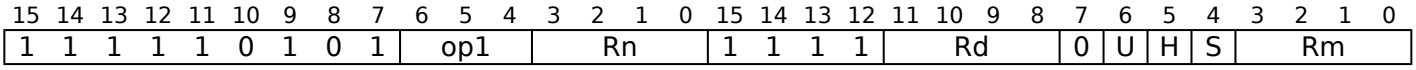
These instructions are under Data-processing (register).



Decode fields op1	U	Rn	Instruction Details
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).

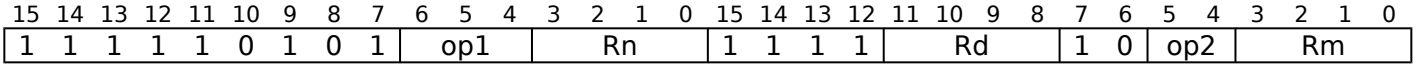


Decode fields op1	U	H	S	Instruction Details
000	0	0	0	SADD8
000	0	0	1	QADD8
000	0	1	0	SHADD8
000	0	1	1	UNALLOCATED
000	1	0	0	UADD8
000	1	0	1	UQADD8
000	1	1	0	UHADD8
000	1	1	1	UNALLOCATED
001	0	0	0	SADD16
001	0	0	1	QADD16
001	0	1	0	SHADD16
001	0	1	1	UNALLOCATED
001	1	0	0	UADD16
001	1	0	1	UQADD16
001	1	1	0	UHADD16
001	1	1	1	UNALLOCATED
010	0	0	0	SASX
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

Decode fields op1	U	H	S	Instruction Details
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).



Decode fields op1	op2	Instruction Details
000	00	QADD
000	01	QDADD
000	10	QSUB
000	11	QDSUB
001	00	REV
001	01	REV16
001	10	RBIT
001	11	REVSH
010	00	SEL
010	01	UNALLOCATED
010	1x	UNALLOCATED
011	00	CLZ
011	01	UNALLOCATED
011	1x	UNALLOCATED

Decode fields op1	op2	Instruction Details
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 op0	Instruction details
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 op1	Ra	op2	Instruction Details
000	!= 1111	00	MLA, MLAS
000		01	MLS
000		1x	UNALLOCATED
000	1111	00	MUL, MULS
001	!= 1111	00	SMLABB, SMLABT, SMLATB, SMLATT — SMLABB
001	!= 1111	01	SMLABB, SMLABT, SMLATB, SMLATT — SMLABT
001	!= 1111	10	SMLABB, SMLABT, SMLATB, SMLATT — SMLATB
001	!= 1111	11	SMLABB, SMLABT, SMLATB, SMLATT — SMLATT
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

Decode fields op1	Ra	op2	Instruction Details
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 op1	op2	Instruction Details
000	!= 0000	UNALLOCATED
000	0000	SMULL, SMULLS
001	!= 1111	UNALLOCATED
001	1111	SDIV
010	!= 0000	UNALLOCATED
010	0000	UMULL, UMULLS
011	!= 1111	UNALLOCATED
011	1111	UDIV
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

Decode fields		Instruction Details
op1	op2	
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	SMLSLD, SMLSLDX — SMLSLD
101	1101	SMLSLD, SMLSLDX — SMLSLDX
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_25, pseudocode v2021-06_xml ; Build timestamp: 2021-06-30T11:15

Copyright © 2010-2021 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.