Instruction Set Assembly Guide for Armv7 and earlier Arm® architectures

Version 2.0

Reference Guide

arm

100076_0200_00_en

Instruction Set Assembly Guide for Armv7 and earlier Arm® architectures

Reference Guide

Release Information

Document History

Issue	Date	Confidentiality	Change
0100-00	25 October 2018	Non-Confidential	First Release
0200-00	09 October 2019	Non-Confidential	Second Release. The title and scope of the document has changed.

Non-Confidential Proprietary Notice

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

LES-PRE-20349

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.

Unrestricted Access is an Arm internal classification.

Product Status

The information in this document is Final, that is for a developed product.

Web Address

www.arm.com

Contents

Instruction Set Assembly Guide for Armv7 and earlier Arm® architectures Reference Guide

List of Figures
Instruction Set Assembly Guide for Armv7 and earlier Arm® architectures Reference Guide

List of Tables
Instruction Set Assembly Guide for Armv7 and earlier Arm® architectures Reference Guide

Preface

This preface introduces the *Instruction Set Assembly Guide for Armv7 and earlier Arm® architectures Reference Guide*.

It contains the following:

About this book

Arm® Instruction Set Assembly Guide for Armv7 and earlier Arm architectures. This document contains an overview of the Arm architecture and information on A32 and T32 instruction sets. For assembler-specific features, such as additional pseudo-instructions, see the documentation for your assembler.

Using this book

This book is organized into the following chapters:

Part A Instruction Set Overview

Chapter A1 Overview of AArch32 state
Gives an overview of the AArch32 state.

Part B Advanced SIMD and Floating-point Programming

Chapter B1 Advanced SIMD Programming
Describes Advanced SIMD assembly language programming.

Chapter B2 Floating-point Programming
Describes floating-point assembly language programming.

Part C A32/T32 Instruction Set Reference

Chapter C1 Condition Codes
Describes condition codes and conditional execution of A32 and T32 code.

Chapter C2 A32 and T32 Instructions
Describes the A32 and T32 instructions supported in AArch32 state.

Chapter C3 Advanced SIMD Instructions (32-bit)
Describes Advanced SIMD assembly language instructions.

Chapter C4 Floating-point Instructions (32-bit)
Describes floating-point assembly language instructions.

Chapter C5 A32/T32 Cryptographic Algorithms
Lists the cryptographic algorithms that A32 and T32 SIMD instructions support.

Glossary

The Arm® Glossary is a list of terms used in Arm documentation, together with definitions for those terms. The Arm Glossary does not contain terms that are industry standard unless the Arm meaning differs from the generally accepted meaning.

See the *Arm® Glossary* for more information.

Typographic conventions

italic
Introduces special terminology, denotes cross-references, and citations.

bold
Highlights interface elements, such as menu names. Denotes signal names. Also used for terms in descriptive lists, where appropriate.

`monospace`
Denotes text that you can enter at the keyboard, such as commands, file and program names, and source code.

`monospace`
: Denotes a permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.

`monospace italic`
: Denotes arguments to monospace text where the argument is to be replaced by a specific value.

`monospace bold`
: Denotes language keywords when used outside example code.

`<and>`
: Encloses replaceable terms for assembler syntax where they appear in code or code fragments. For example:

```
MRC p15, 0, <Rd>, <CRn>, <CRm>, <Opcode_2>
```

SMALL CAPITALS
: Used in body text for a few terms that have specific technical meanings, that are defined in the *Arm® Glossary*. For example, IMPLEMENTATION DEFINED, IMPLEMENTATION SPECIFIC, UNKNOWN, and UNPREDICTABLE.

Feedback

Feedback on this product

If you have any comments or suggestions about this product, contact your supplier and give:

- The product name.
- The product revision or version.
- An explanation with as much information as you can provide. Include symptoms and diagnostic procedures if appropriate.

Feedback on content

If you have comments on content then send an e-mail to *errata@arm.com*. Give:

- The title *Instruction Set Assembly Guide for Armv7 and earlier Arm architectures Reference Guide*.
- The number 100076_0200_00_en.
- If applicable, the page number(s) to which your comments refer.
- A concise explanation of your comments.

Arm also welcomes general suggestions for additions and improvements.

Note

Arm tests the PDF only in Adobe Acrobat and Acrobat Reader, and cannot guarantee the quality of the represented document when used with any other PDF reader.

Other information

- *Arm® Developer*.
- *Arm® Information Center*.
- *Arm® Technical Support Knowledge Articles*.
- *Technical Support*.
- *Arm® Glossary*.

Part A
Instruction Set Overview

Chapter A1
Overview of AArch32 state

Gives an overview of the AArch32 state.

It contains the following sections:

A1.1 Terminology

This document uses the following terms to refer to instruction sets.

Instruction sets for Armv7 and earlier architectures were called the ARM and Thumb instruction sets.

This document describes the instruction sets for Armv7 and earlier architectures, but uses terminology that is introduced with Armv8:

A32
The A32 instruction set was previously called the ARM instruction set. It is a fixed-length instruction set that uses 32-bit instruction encodings.

T32
The T32 instruction set was previously called the Thumb instruction set. It is a variable-length instruction set that uses both 16-bit and 32-bit instruction.

AArch32
The AArch32 Execution state supports the A32 and T32 instruction sets.

The Arm 32-bit Execution state uses 32-bit general purpose registers, and a 32-bit program counter (PC), stack pointer (SP), and link register (LR). In implementations of the Arm architecture beforeArmv8, execution is always in AArch32 state.

Note

Some examples and descriptions in this document might apply only to the `armasm` legacy assembler.

A1.2 Changing between A32 and T32 instruction set states

A processor that is executing A32 instructions is operating in *A32 instruction set state*. A processor that is executing T32 instructions is operating in *T32 instruction set state*. For brevity, this document refers to them as the *A32 state* and *T32 state* respectively.

A processor in A32 state cannot execute T32 instructions, and a processor in T32 state cannot execute A32 instructions. You must ensure that the processor never receives instructions of the wrong instruction set for the current state.

The initial state after reset depends on the processor being used and its configuration.

To direct `armasm` to generate A32 or T32 instruction encodings, you must set the assembler mode using an `ARM` or `THUMB` directive. Assembly code using `CODE32` and `CODE16` directives can still be assembled, but Arm recommends you use the `ARM` and `THUMB` directives for new code.

These directives do not change the instruction set state of the processor. To do this, you must use an appropriate instruction, for example `BX` or `BLX` to change between A32 and T32 states when performing a branch.

Related references

C2.20 BLX, BLXNS on page C2-140
C2.21 BX, BXNS on page C2-142

A1.3 Processor modes, and privileged and unprivileged software execution

The Arm architecture supports different levels of execution privilege. The privilege level depends on the processor mode.

Note

Armv6-M, Armv7-M, Armv8-M Baseline, and Armv8-M Mainline do not support the same modes as other Arm architectures and profiles. Some of the processor modes listed here do not apply to these architectures.

Table A1-1 AArch32 processor modes

Processor mode	Mode number
User	0b10000
FIQ	0b10001
IRQ	0b10010
Supervisor	0b10011
Monitor	0b10110
Abort	0b10111
Hyp	0b11010
Undefined	0b11011
System	0b11111

User mode is an unprivileged mode, and has restricted access to system resources. All other modes have full access to system resources in the current security state, can change mode freely, and execute software as privileged.

Applications that require task protection usually execute in User mode. Some embedded applications might run entirely in any mode other than User mode. An application that requires full access to system resources usually executes in System mode.

Modes other than User mode are entered to service exceptions, or to access privileged resources.

Code can run in either a Secure state or in a Non-secure state. Hypervisor (Hyp) mode has privileged execution in Non-secure state.

Related concepts

A1.4 Processor modes in Armv6-M, Armv7-M, and Armv8-M on page A1-29

Related information

Arm Architecture Reference Manual

A1.4 Processor modes in Armv6-M, Armv7-M, and Armv8-M

The processor modes available in Armv6-M, Armv7-M, Armv8-M Baseline, and Armv8-M Mainline are Thread mode and Handler mode.

Thread mode is the normal mode that programs run in. Thread mode can be privileged or unprivileged software execution. Handler mode is the mode that exceptions are handled in. It is always privileged software execution.

Related concepts

A1.3 Processor modes, and privileged and unprivileged software execution on page A1-28

Related information

Arm Architecture Reference Manual

A1.5 Registers in AArch32 state

Arm processors provide general-purpose and special-purpose registers. Some additional registers are available in privileged execution modes.

In all Arm processors in AArch32 state, the following registers are available and accessible in any processor mode:

- 15 general-purpose registers R0-R12, the *Stack Pointer* (SP), and *Link Register* (LR).
- 1 *Program Counter* (PC).
- 1 *Application Program Status Register* (APSR).

Note

- SP and LR can be used as general-purpose registers, although Arm deprecates using SP other than as a stack pointer.

Additional registers are available in privileged software execution. Arm processors have a total of 43 registers. The registers are arranged in partially overlapping banks. There is a different register bank for each processor mode. The banked registers give rapid context switching for dealing with processor exceptions and privileged operations.

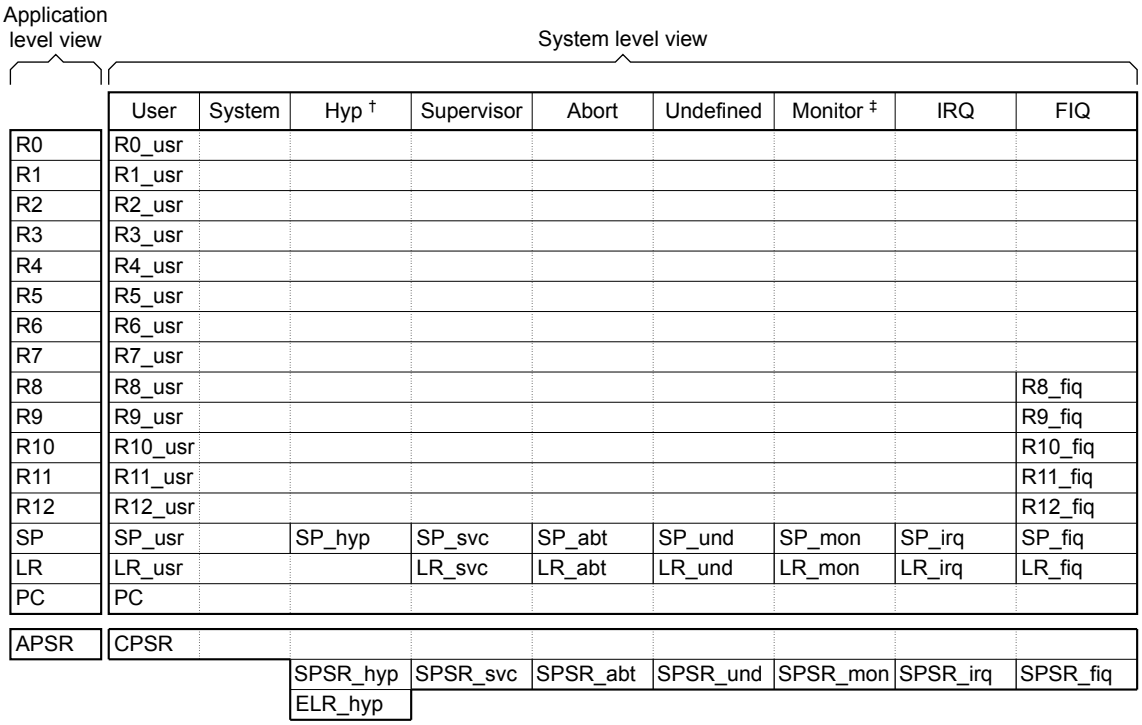
The additional registers in Arm processors are:

- 2 supervisor mode registers for banked SP and LR.
- 2 abort mode registers for banked SP and LR.
- 2 undefined mode registers for banked SP and LR.
- 2 interrupt mode registers for banked SP and LR.
- 7 FIQ mode registers for banked R8-R12, SP and LR.
- 2 monitor mode registers for banked SP and LR.
- 1 Hyp mode register for banked SP.
- 7 *Saved Program Status Register* (SPSRs), one for each exception mode.
- 1 Hyp mode register for ELR_Hyp to store the preferred return address from Hyp mode.

Note

In privileged software execution, CPSR is an alias for APSR and gives access to additional bits.

The following figure shows how the registers are banked in the Arm architecture.



‡ Exists only in Secure state.

† Exists only in Non-secure state.

Cells with no entry indicate that the User mode register is used.

Figure A1-1 Organization of general-purpose registers and Program Status Registers

In Armv6-M, Armv7-M, Armv8-M Baseline, and Armv8-M Mainline based processors, SP is an alias for the two banked stack pointer registers:

- Main stack pointer register, that is only available in privileged software execution.
- Process stack pointer register.

Related concepts

A1.6 General-purpose registers in AArch32 state on page A1-32

A1.10 Program Counter in AArch32 state on page A1-36

A1.12 Application Program Status Register on page A1-38

A1.14 Saved Program Status Registers in AArch32 state on page A1-40

A1.13 Current Program Status Register in AArch32 state on page A1-39

A1.3 Processor modes, and privileged and unprivileged software execution on page A1-28

Related information

Arm Architecture Reference Manual

A1.6 General-purpose registers in AArch32 state

There are restrictions on the use of SP and LR as general-purpose registers.

With the exception of Armv6-M, Armv7-M, Armv8-M Baseline, and Armv8-M Mainline based processors, there are 33 general-purpose 32-bit registers, including the banked SP and LR registers. Fifteen general-purpose registers are visible at any one time, depending on the current processor mode. These are R0-R12, SP, and LR. The PC (R15) is not considered a general-purpose register.

SP (or R13) is the *stack pointer*. The C and C++ compilers always use SP as the stack pointer. Arm deprecates most uses of SP as a general purpose register. In T32 state, SP is strictly defined as the stack pointer. The instruction descriptions in *Chapter C2 A32 and T32 Instructions* on page C2-101 describe when SP and PC can be used.

In User mode, LR (or R14) is used as a *link register* to store the return address when a subroutine call is made. It can also be used as a general-purpose register if the return address is stored on the stack.

In the exception handling modes, LR holds the return address for the exception, or a subroutine return address if subroutine calls are executed within an exception. LR can be used as a general-purpose register if the return address is stored on the stack.

Related concepts

A1.10 Program Counter in AArch32 state on page A1-36

A1.7 Register accesses in AArch32 state on page A1-33

Related references

A1.8 Predeclared core register names in AArch32 state on page A1-34

C2.62 MRS (PSR to general-purpose register) on page C2-204

C2.65 MSR (general-purpose register to PSR) on page C2-208

A1.7 Register accesses in AArch32 state

16-bit T32 instructions can access only a limited set of registers. There are also some restrictions on the use of special-purpose registers by A32 and 32-bit T32 instructions.

Most 16-bit T32 instructions can only access R0 to R7. Only a small number of T32 instructions can access R8-R12, SP, LR, and PC. Registers R0 to R7 are called Lo registers. Registers R8-R12, SP, LR, and PC are called Hi registers.

All 32-bit T32 instructions can access R0 to R12, and LR. However, apart from a few designated stack manipulation instructions, most T32 instructions cannot use SP. Except for a few specific instructions where PC is useful, most T32 instructions cannot use PC.

In A32 state, all instructions can access R0 to R12, SP, and LR, and most instructions can also access PC (R15). However, the use of the SP in an A32 instruction, in any way that is not possible in the corresponding T32 instruction, is deprecated. Explicit use of the PC in an A32 instruction is not usually useful, and except for specific instances that are useful, such use is deprecated. Implicit use of the PC, for example in branch instructions or load (literal) instructions, is never deprecated.

The `MRS` instructions can move the contents of a status register to a general-purpose register, where they can be manipulated by normal data processing operations. You can use the `MSR` instruction to move the contents of a general-purpose register to a status register.

Related concepts

A1.6 General-purpose registers in AArch32 state on page A1-32

A1.10 Program Counter in AArch32 state on page A1-36

A1.12 Application Program Status Register on page A1-38

A1.13 Current Program Status Register in AArch32 state on page A1-39

A1.14 Saved Program Status Registers in AArch32 state on page A1-40

Related references

A1.8 Predeclared core register names in AArch32 state on page A1-34

C2.62 MRS (PSR to general-purpose register) on page C2-204

C2.65 MSR (general-purpose register to PSR) on page C2-208

A1.8 Predeclared core register names in AArch32 state

Many of the core register names have synonyms.

The following table shows the predeclared core registers:

Table A1-2 Predeclared core registers in AArch32 state

Register names	Meaning
`r0-r15` and `R0-R15`	General purpose registers.
`a1-a4`	Argument, result or scratch registers. These are synonyms for R0 to R3.
`v1-v8`	Variable registers. These are synonyms for R4 to R11.
`SB`	Static base register. This is a synonym for R9.
`IP`	Intra-procedure call scratch register. This is a synonym for R12.
`SP`	Stack pointer. This is a synonym for R13.
`LR`	Link register. This is a synonym for R14.
`PC`	Program counter. This is a synonym for R15.

With the exception of `a1-a4` and `v1-v8`, you can write the register names either in all upper case or all lower case.

Related concepts

A1.6 General-purpose registers in AArch32 state on page A1-32

A1.9 Predeclared extension register names in AArch32 state

You can write the names of Advanced SIMD and floating-point registers either in upper case or lower case.

The following table shows the predeclared extension register names:

Table A1-3 Predeclared extension registers in AArch32 state

Register names	Meaning
`Q0-Q15`	Advanced SIMD quadword registers
`D0-D31`	Advanced SIMD doubleword registers, floating-point double-precision registers
`S0-S31`	Floating-point single-precision registers

You can write the register names either in upper case or lower case.

A1.10 Program Counter in AArch32 state

You can use the Program Counter explicitly, for example in some T32 data processing instructions, and implicitly, for example in branch instructions.

The *Program Counter* (PC) is accessed as PC (or R15). It is incremented by the size of the instruction executed, which is always four bytes in A32 state. Branch instructions load the destination address into the PC. You can also load the PC directly using data operation instructions. For example, to branch to the address in a general purpose register, use:

```
MOV PC,R0
```

During execution, the PC does not contain the address of the currently executing instruction. The address of the currently executing instruction is typically PC-8 for A32, or PC-4 for T32.

Note

Arm recommends you use the `BX` instruction to jump to an address or to return from a function, rather than writing to the PC directly.

Related references

C2.14 B on page C2-132
C2.21 BX, BXNS on page C2-142
C2.23 CBZ and CBNZ on page C2-145
C2.154 TBB and TBH on page C2-333

A1.11 The Q flag in AArch32 state

The Q flag indicates overflow or saturation. It is one of the program status flags held in the APSR.

The Q flag is set to 1 when saturation occurs in saturating arithmetic instructions, or when overflow occurs in certain multiply instructions.

The Q flag is a *sticky* flag. Although the saturating and certain multiply instructions can set the flag, they cannot clear it. You can execute a series of such instructions, and then test the flag to find out whether saturation or overflow occurred at any point in the series, without having to check the flag after each instruction.

To clear the Q flag, use an `MSR` instruction to read-modify-write the APSR:

```
MRS r5, APSR
BIC r5, r5, #(1<<27)
MSR APSR_nzcvq, r5
```

The state of the Q flag cannot be tested directly by the condition codes. To read the state of the Q flag, use an `MRS` instruction.

```
MRS r6, APSR
TST r6, #(1<<27); Z is clear if Q flag was set
```

Related references

C2.62 MRS (PSR to general-purpose register) on page C2-204
C2.65 MSR (general-purpose register to PSR) on page C2-208
C2.75 QADD on page C2-223
C2.125 SMULxy on page C2-285
C2.127 SMULWy on page C2-287

A1.12 Application Program Status Register

The *Application Program Status Register* (APSR) holds the program status flags that are accessible in any processor mode.

It holds copies of the N, Z, C, and V *condition flags*. The processor uses them to determine whether or not to execute conditional instructions.

The APSR also holds:

- The Q (saturation) flag.
- The APSR also holds the GE (Greater than or Equal) flags. The GE flags can be set by the parallel add and subtract instructions. They are used by the `SEL` instruction to perform byte-based selection from two registers.

These flags are accessible in all modes, using the `MSR` and `MRS` instructions.

Related concepts

C1.1 Conditional instructions on page C1-84

Related references

C1.5 Updates to the condition flags in A32/T32 code on page C1-88

C2.62 MRS (PSR to general-purpose register) on page C2-204

C2.65 MSR (general-purpose register to PSR) on page C2-208

C2.100 SEL on page C2-257

A1.13 Current Program Status Register in AArch32 state

The *Current Program Status Register* (CPSR) holds the same program status flags as the APSR, and some additional information.

It holds:

- The APSR flags.
- The processor mode.
- The interrupt disable flags.
- The instruction set state (A32 or T32).
- The endianness state.
- The execution state bits for the IT block.

The execution state bits control conditional execution in the IT block.

Only the APSR flags are accessible in all modes. Arm deprecates using an `MSR` instruction to change the endianness bit (E) of the CPSR, in any mode. Each exception level can have its own endianness, but mixed endianness within an exception level is deprecated.

The `SETEND` instruction is deprecated.

The execution state bits for the IT block (IT[1:0]) and the T32 bit (T) can be accessed by `MRS` only in Debug state.

Related concepts

A1.14 Saved Program Status Registers in AArch32 state on page A1-40

Related references

C2.41 IT on page C2-169
C2.62 MRS (PSR to general-purpose register) on page C2-204
C2.65 MSR (general-purpose register to PSR) on page C2-208
C2.101 SETEND on page C2-259
C1.5 Updates to the condition flags in A32/T32 code on page C1-88

A1.14 Saved Program Status Registers in AArch32 state

The *Saved Program Status Register* (SPSR) stores the current value of the CPSR when an exception is taken so that it can be restored after handling the exception.

Each exception handling mode can access its own SPSR. User mode and System mode do not have an SPSR because they are not exception handling modes.

The execution state bits, including the endianness state and current instruction set state can be accessed from the SPSR in any exception mode, using the `MSR` and `MRS` instructions. You cannot access the SPSR using `MSR` or `MRS` in User or System mode.

Related concepts

A1.13 Current Program Status Register in AArch32 state on page A1-39

A1.15 A32 and T32 instruction set overview

A32 and T32 instructions can be grouped by functional area.

All A32 instructions are 32 bits long. Instructions are stored word-aligned, so the least significant two bits of instruction addresses are always zero in A32 state.

T32 instructions are either 16 or 32 bits long. Instructions are stored half-word aligned. Some instructions use the least significant bit of the address to determine whether the code being branched to is T32 or A32.

Before the introduction of 32-bit T32 instructions, the T32 instruction set was limited to a restricted subset of the functionality of the A32 instruction set. Almost all T32 instructions were 16-bit. Together, the 32-bit and 16-bit T32 instructions provide functionality that is almost identical to that of the A32 instruction set.

The following table describes some of the functional groupings of the available instructions.

Table A1-4 A32 instruction groups

Instruction group	Description
Branch and control	These instructions do the following: • Branch to subroutines. • Branch backwards to form loops. • Branch forward in conditional structures. • Make the following instruction conditional without branching. • Change the processor between A32 state and T32 state.
Data processing	These instructions operate on the general-purpose registers. They can perform operations such as addition, subtraction, or bitwise logic on the contents of two registers and place the result in a third register. They can also operate on the value in a single register, or on a value in a register and an immediate value supplied within the instruction. Long multiply instructions give a 64-bit result in two registers.
Register load and store	These instructions load or store the value of a single register from or to memory. They can load or store a 32-bit word, a 16-bit halfword, or an 8-bit unsigned byte. Byte and halfword loads can either be sign extended or zero extended to fill the 32-bit register. A few instructions are also defined that can load or store 64-bit doubleword values into two 32-bit registers.
Multiple register load and store	These instructions load or store any subset of the general-purpose registers from or to memory.
Status register access	These instructions move the contents of a status register to or from a general-purpose register.

A1.16 Access to the inline barrel shifter in AArch32 state

The AArch32 arithmetic logic unit has a 32-bit barrel shifter that is capable of shift and rotate operations.

The second operand to many A32 and T32 data-processing and single register data-transfer instructions can be shifted, before the data-processing or data-transfer is executed, as part of the instruction. This supports, but is not limited to:

- Scaled addressing.
- Multiplication by an immediate value.
- Constructing immediate values.

32-bit T32 instructions give almost the same access to the barrel shifter as A32 instructions.

16-bit T32 instructions only allow access to the barrel shifter using separate instructions.

Part B
Advanced SIMD and Floating-point Programming

Chapter B1
Advanced SIMD Programming

Describes Advanced SIMD assembly language programming.

It contains the following sections:

B1.1 Architecture support for Advanced SIMD

Advanced SIMD is an optional extension to the Armv7 architecture.

All Advanced SIMD instructions are available on systems that support Advanced SIMD. Some of these instructions are also available on systems that implement the floating-point extension without Advanced SIMD. These are called shared instructions.

The Advanced SIMD register bank consists of thirty-two 64-bit registers, and smaller registers are packed into larger ones.

Note

Advanced SIMD and floating-point instructions share the same extension register bank.

Related information

Floating-point support

B1.2 Extension register bank mapping for Advanced SIMD in AArch32 state

The Advanced SIMD extension register bank is a collection of registers that can be accessed as either 64-bit or 128-bit registers.

Advanced SIMD and floating-point instructions use the same extension register bank, and is distinct from the Arm core register bank.

The following figure shows the views of the extension register bank, and the overlap between the different size registers. For example, the 128-bit register `Q0` is an alias for two consecutive 64-bit registers `D0` and D1. The 128-bit register `Q8` is an alias for 2 consecutive 64-bit registers `D16` and `D17`.

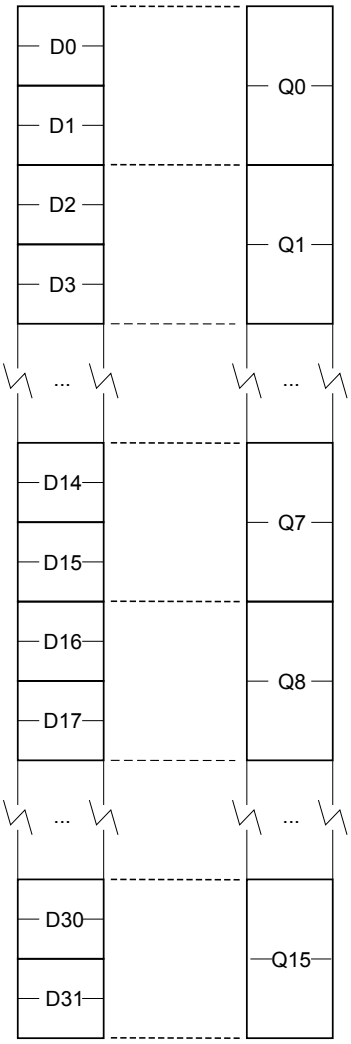


Figure B1-1 Extension register bank for Advanced SIMD in AArch32 state

Note

If your processor supports both Advanced SIMD and floating-point, all the Advanced SIMD registers overlap with the floating-point registers.

The aliased views enable half-precision, single-precision, and double-precision values, and Advanced SIMD vectors to coexist in different non-overlapped registers at the same time.

You can also use the same overlapped registers to store half-precision, single-precision, and double-precision values, and Advanced SIMD vectors at different times.

Do not attempt to use overlapped 64-bit and 128-bit registers at the same time because it creates meaningless results.

The mapping between the registers is as follows:

- `D<2n>` maps to the least significant half of `Q<n>`
- `D<2n+1>` maps to the most significant half of `Q<n>`.

For example, you can access the least significant half of the elements of a vector in `Q6` by referring to `D12`, and the most significant half of the elements by referring to `D13`.

Related concepts

B2.3 Views of the floating-point extension register bank in AArch32 state on page B2-69
B1.3 Views of the Advanced SIMD register bank in AArch32 state on page B1-49

B1.3 Views of the Advanced SIMD register bank in AArch32 state

Advanced SIMD can have different views of the extension register bank in AArch32 state.

It can view the extension register bank as:

- Sixteen 128-bit registers, `Q0-Q15`.
- Thirty-two 64-bit registers, `D0-D31`.
- A combination of registers from these views.

Advanced SIMD views each register as containing a *vector* of 1, 2, 4, 8, or 16 elements, all of the same size and type. Individual elements can also be accessed as *scalars*.

In Advanced SIMD, the 64-bit registers are called doubleword registers and the 128-bit registers are called quadword registers.

Related concepts

B1.2 Extension register bank mapping for Advanced SIMD in AArch32 state on page B1-47
B2.3 Views of the floating-point extension register bank in AArch32 state on page B2-69

B1.4 Load values to Advanced SIMD registers

To load a register with a floating-point immediate value, use `VMOV` instruction. This instruction has scalar and vector forms.

The Advanced SIMD instructions `VMOV` and `VMVN` can also load integer immediates.

Related references

C3.57 VLDR pseudo-instruction on page C3-452
C4.22 VMOV (floating-point) on page C4-569
C3.68 VMOV (immediate) on page C3-463

B1.5 Conditional execution of A32/T32 Advanced SIMD instructions

Most Advanced SIMD instructions always execute unconditionally.

You cannot use any of the following Advanced SIMD instructions in an IT block:

- `VCVT{A, N, P, M}`.
- `VMAXNM`.
- `VMINNM`.
- `VRINT{N, X, A, Z, M, P}`.
- All instructions in the Crypto extension.

In addition, specifying any other Advanced SIMD instruction in an IT block is deprecated.

Arm deprecates conditionally executing any Advanced SIMD instruction unless it is a shared Advanced SIMD and floating-point instruction.

Related concepts

C1.2 Conditional execution in A32 code on page C1-85

C1.3 Conditional execution in T32 code on page C1-86

Related references

C1.11 Comparison of condition code meanings in integer and floating-point code on page C1-94

C1.9 Condition code suffixes on page C1-92

B1.6 Floating-point exceptions for Advanced SIMD in A32/T32 instructions

The Advanced SIMD extension records floating-point exceptions in the FPSCR cumulative flags.

It records the following exceptions:

Invalid operation
The exception is caused if the result of an operation has no mathematical value or cannot be represented.

Division by zero
The exception is caused if a divide operation has a zero divisor and a dividend that is not zero, an infinity or a NaN.

Overflow
The exception is caused if the absolute value of the result of an operation, produced after rounding, is greater than the maximum positive normalized number for the destination precision.

Underflow
The exception is caused if the absolute value of the result of an operation, produced before rounding, is less than the minimum positive normalized number for the destination precision, and the rounded result is inexact.

Inexact
The exception is caused if the result of an operation is not equivalent to the value that would be produced if the operation were performed with unbounded precision and exponent range.

Input denormal
The exception is caused if a denormalized input operand is replaced in the computation by a zero.

The descriptions of the Advanced SIMD instructions that can cause floating-point exceptions include a subsection listing the exceptions. If there is no such subsection, that instruction cannot cause any floating-point exception.

Related concepts

B1.15 Flush-to-zero mode in Advanced SIMD on page B1-61

Related references

Chapter B1 Advanced SIMD Programming on page B1-45

Related information

Arm Architecture Reference Manual

B1.7 Advanced SIMD data types in A32/T32 instructions

Most Advanced SIMD instructions use a data type specifier to define the size and type of data that the instruction operates on.

Data type specifiers in Advanced SIMD instructions consist of a letter indicating the type of data, usually followed by a number indicating the width. They are separated from the instruction mnemonic by a point. The following table shows the data types available in Advanced SIMD instructions:

Table B1-1 Advanced SIMD data types

	8-bit	16-bit	32-bit	64-bit
Unsigned integer	U8	U16	U32	U64
Signed integer	S8	S16	S32	S64
Integer of unspecified type	I8	I16	I32	I64
Floating-point number	not available	F16	F32 (or F)	not available
Polynomial over {0,1}	P8	P16	not available	not available

The datatype of the second (or only) operand is specified in the instruction.

Note

Most instructions have a restricted range of permitted data types. See the instruction descriptions for details. However, the data type description is flexible:

- If the description specifies I, you can also use the S or U data types.
- If only the data size is specified, you can specify a type (I, S, U, P or F).
- If no data type is specified, you can specify a data type.

Related concepts

B1.7 Advanced SIMD data types in A32/T32 instructions on page B1-53
B1.8 Polynomial arithmetic over {0,1} on page B1-54

B1.8 Polynomial arithmetic over {0,1}

The coefficients 0 and 1 are manipulated using the rules of Boolean arithmetic.

The following rules apply:

- 0 + 0 = 1 + 1 = 0.
- 0 + 1 = 1 + 0 = 1.
- 0 * 0 = 0 * 1 = 1 * 0 = 0.
- 1 * 1 = 1.

That is, adding two polynomials over {0,1} is the same as a bitwise exclusive OR, and multiplying two polynomials over {0,1} is the same as integer multiplication except that partial products are exclusive-ORed instead of being added.

Related concepts

B1.7 Advanced SIMD data types in A32/T32 instructions on page B1-53

B1.9 Advanced SIMD vectors

An Advanced SIMD operand can be a vector or a scalar. An Advanced SIMD vector can be a 64-bit doubleword vector or a 128-bit quadword vector.

The size of the elements in an Advanced SIMD vector is specified by a datatype suffix appended to the mnemonic.

Doubleword vectors can contain:

- Eight 8-bit elements.
- Four 16-bit elements.
- Two 32-bit elements.
- One 64-bit element.

Quadword vectors can contain:

- Sixteen 8-bit elements.
- Eight 16-bit elements.
- Four 32-bit elements.
- Two 64-bit elements.

Related concepts

B1.12 Advanced SIMD scalars on page B1-58
B1.2 Extension register bank mapping for Advanced SIMD in AArch32 state on page B1-47
B1.13 Extended notation extension for Advanced SIMD on page B1-59
B1.7 Advanced SIMD data types in A32/T32 instructions on page B1-53
B1.10 Normal, long, wide, and narrow Advanced SIMD instructions on page B1-56

B1.10 Normal, long, wide, and narrow Advanced SIMD instructions

Many Advanced SIMD data processing instructions are available in Normal, Long, Wide, Narrow, and saturating variants.

Normal operation

The operands can be any of the vector types. The result vector is the same width, and usually the same type, as the operand vectors, for example:

```
VADD.I16 D0, D1, D2
```

You can specify that the operands and result of a normal Advanced SIMD instruction must all be quadwords by appending a `Q` to the instruction mnemonic. If you do this, `armasm` produces an error if the operands or result are not quadwords.

Long operation

The operands are doubleword vectors and the result is a quadword vector. The elements of the result are usually twice the width of the elements of the operands, and the same type.

Long operation is specified using an `L` appended to the instruction mnemonic, for example:

```
VADDL.S16 Q0, D2, D3
```

Wide operation

One operand vector is doubleword and the other is quadword. The result vector is quadword. The elements of the result and the first operand are twice the width of the elements of the second operand.

Wide operation is specified using a `W` appended to the instruction mnemonic, for example:

```
VADDW.S16 Q0, Q1, D4
```

Narrow operation

The operands are quadword vectors and the result is a doubleword vector. The elements of the result are half the width of the elements of the operands.

Narrow operation is specified using an `N` appended to the instruction mnemonic, for example:

```
VADDHN.I16 D0, Q1, Q2
```

Related concepts

B1.9 Advanced SIMD vectors on page B1-55

B1.11 Saturating Advanced SIMD instructions

Saturating instructions saturate the result to the value of the upper limit or lower limit if the result overflows or underflows.

The saturation limits depend on the datatype of the instruction. The following table shows the ranges that Advanced SIMD saturating instructions saturate to, where *x* is the result of the operation.

Table B1-2 Advanced SIMD saturation ranges

Data type	Saturation range of *x*
Signed byte (`S8`)	$-2^7 <= x < 2^7$
Signed halfword (`S16`)	$-2^{15} <= x < 2^{15}$
Signed word (`S32`)	$-2^{31} <= x < 2^{31}$
Signed doubleword (`S64`)	$-2^{63} <= x < 2^{63}$
Unsigned byte (`U8`)	$0 <= x < 2^8$
Unsigned halfword (`U16`)	$0 <= x < 2^{16}$
Unsigned word (`U32`)	$0 <= x < 2^{32}$
Unsigned doubleword (`U64`)	$0 <= x < 2^{64}$

Saturating Advanced SIMD arithmetic instructions set the QC bit in the floating-point status register (FPSCR) to indicate that saturation has occurred.

Saturating instructions are specified using a `Q` prefix, which is inserted between the `V` and the instruction mnemonic.

Related references

C2.7 Saturating instructions on page C2-118

B1.12 Advanced SIMD scalars

Some Advanced SIMD instructions act on scalars in combination with vectors. Advanced SIMD scalars can be 8-bit, 16-bit, 32-bit, or 64-bit.

The instruction syntax refers to a single element in a vector register using an index, *x*, into the vector, so that `Dm[x]` is the *x*th element in vector `Dm`.

Except for Advanced SIMD multiply instructions, instructions that access scalars can access any element in the register bank.

Advanced SIMD multiply instructions only allow 16-bit or 32-bit scalars, and can only access the first 32 scalars in the register bank.

In multiply instructions:

- 16-bit scalars are restricted to registers D0-D7, with *x* in the range 0-3.
- 32-bit scalars are restricted to registers D0-D15, with *x* either 0 or 1.

Related concepts

B1.9 Advanced SIMD vectors on page B1-55
B1.2 Extension register bank mapping for Advanced SIMD in AArch32 state on page B1-47

B1.13 Extended notation extension for Advanced SIMD

`armasm` implements an extension to the architectural Advanced SIMD assembly syntax, called *extended notation*. This extension allows you to include datatype information or scalar indexes in register names.

If you use extended notation, you do not have to include the data type or scalar index information in every instruction.

Register names can be any of the following:

Untyped
The register name specifies the register, but not what datatype it contains, nor any index to a particular scalar within the register.

Untyped with scalar index
The register name specifies the register, but not what datatype it contains, It specifies an index to a particular scalar within the register.

Typed
The register name specifies the register, and what datatype it contains, but not any index to a particular scalar within the register.

Typed with scalar index
The register name specifies the register, what datatype it contains, and an index to a particular scalar within the register.

Use the `DN` and `QN` directives to define names for typed and scalar registers.

Related concepts

B1.9 Advanced SIMD vectors on page B1-55
B1.7 Advanced SIMD data types in A32/T32 instructions on page B1-53
B1.12 Advanced SIMD scalars on page B1-58

B1.14 Advanced SIMD system registers in AArch32 state

Advanced SIMD system registers are accessible in all implementations of Advanced SIMD.

For exception levels using AArch32, the following Advanced SIMD system registers are accessible in all Advanced SIMD implementations:

- FPSCR, the floating-point status and control register.
- FPEXC, the floating-point exception register.
- FPSID, the floating-point system ID register.

A particular Advanced SIMD implementation can have additional registers. For more information, see the Technical Reference Manual for your processor.

Note

Advanced SIMD technology shares the same set of system registers as floating-point.

Related information

Arm Architecture Reference Manual

B1.15 Flush-to-zero mode in Advanced SIMD

Flush-to-zero mode replaces denormalized numbers with zero. This does not comply with IEEE 754 arithmetic, but in some circumstances can improve performance considerably.

Flush-to-zero mode in Advanced SIMD always preserves the sign bit.

Advanced SIMD always uses flush-to-zero mode.

Related concepts

B1.17 The effects of using flush-to-zero mode in Advanced SIMD on page B1-63

Related references

B1.16 When to use flush-to-zero mode in Advanced SIMD on page B1-62
B1.18 Advanced SIMD operations not affected by flush-to-zero mode on page B1-64

B1.16 When to use flush-to-zero mode in Advanced SIMD

You can change between flush-to-zero mode and normal mode, depending on the requirements of different parts of your code.

You can change between flush-to-zero mode and normal mode, depending on the requirements of different parts of your code.

You must select flush-to-zero mode if all the following are true:

- IEEE 754 compliance is not a requirement for your system.
- The algorithms you are using sometimes generate denormalized numbers.
- Your system uses support code to handle denormalized numbers.
- The algorithms you are using do not depend for their accuracy on the preservation of denormalized numbers.
- The algorithms you are using do not generate frequent exceptions as a result of replacing denormalized numbers with 0.

You select flush-to-zero mode by setting the FZ bit in the FPSCR to 1. You do this using the `VMRS` and `VMSR` instructions.

You can change between flush-to-zero and normal mode at any time, if different parts of your code have different requirements. Numbers already in registers are not affected by changing mode.

Related concepts

B1.15 Flush-to-zero mode in Advanced SIMD on page B1-61
B1.17 The effects of using flush-to-zero mode in Advanced SIMD on page B1-63

B1.17 The effects of using flush-to-zero mode in Advanced SIMD

In flush-to-zero mode, denormalized inputs are treated as zero. Results that are too small to be represented in a normalized number are replaced with zero.

With certain exceptions, flush-to-zero mode has the following effects on floating-point operations:

- A denormalized number is treated as 0 when used as an input to a floating-point operation. The source register is not altered.
- If the result of a single-precision floating-point operation, before rounding, is in the range -2^{-126} to $+2^{-126}$, it is replaced by 0.
- If the result of a double-precision floating-point operation, before rounding, is in the range -2^{-1022} to $+2^{-1022}$, it is replaced by 0.

In flush-to-zero mode, an Input Denormal exception occurs whenever a denormalized number is used as an operand. An Underflow exception occurs when a result is flushed-to-zero.

Related concepts

B1.15 Flush-to-zero mode in Advanced SIMD on page B1-61

Related references

B1.18 Advanced SIMD operations not affected by flush-to-zero mode on page B1-64

B1.18 Advanced SIMD operations not affected by flush-to-zero mode

Some Advanced SIMD instructions can be carried out on denormalized numbers even in flush-to-zero mode, without flushing the results to zero.

These instructions are as follows:

- Copy, absolute value, and negate (`VMOV`, `VMVN`, `V{Q}ABS`, and `V{Q}NEG`).
- Duplicate (`VDUP`).
- Swap (`VSWP`).
- Load and store (`VLDR` and `VSTR`).
- Load multiple and store multiple (`VLDM` and `VSTM`).
- Transfer between extension registers and AArch32 general-purpose registers (`VMOV`).

Related concepts

B1.15 Flush-to-zero mode in Advanced SIMD on page B1-61

Related references

C3.9 VABS on page C3-401

C4.2 VABS (floating-point) on page C4-549

C3.41 VDUP on page C3-433

C3.54 VLDM on page C3-449

C3.55 VLDR on page C3-450

C3.69 VMOV (register) on page C3-464

C3.70 VMOV (between two general-purpose registers and a 64-bit extension register) on page C3-465

C3.71 VMOV (between a general-purpose register and an Advanced SIMD scalar) on page C3-466

C3.139 VSWP on page C3-536

Chapter B2
Floating-point Programming

Describes floating-point assembly language programming.

It contains the following sections:

B2.1 Architecture support for floating-point

Floating-point is an optional extension to the Arm architecture. There are versions that provide additional instructions.

The floating-point instruction set is based on VFPv4, but with the addition of some new instructions, including the following:

- Floating-point round to integral.
- Conversion from floating-point to integer with a directed rounding mode.
- Direct conversion between half-precision and double-precision floating-point.
- Floating-point conditional select.

The register bank consists of thirty-two 64-bit registers, and smaller registers are packed into larger ones, as in Armv7 and earlier.

B2.2 Extension register bank mapping for floating-point in AArch32 state

The floating-point extension register bank is a collection of registers that can be accessed as either 32-bit or 64-bit registers. It is distinct from the Arm core register bank.

The following figure shows the views of the extension register bank, and the overlap between the different size registers. For example, the 64-bit register `D0` is an alias for two consecutive 32-bit registers `S0` and `S1`. The 64-bit registers `D16` and `D17` do not have an alias.

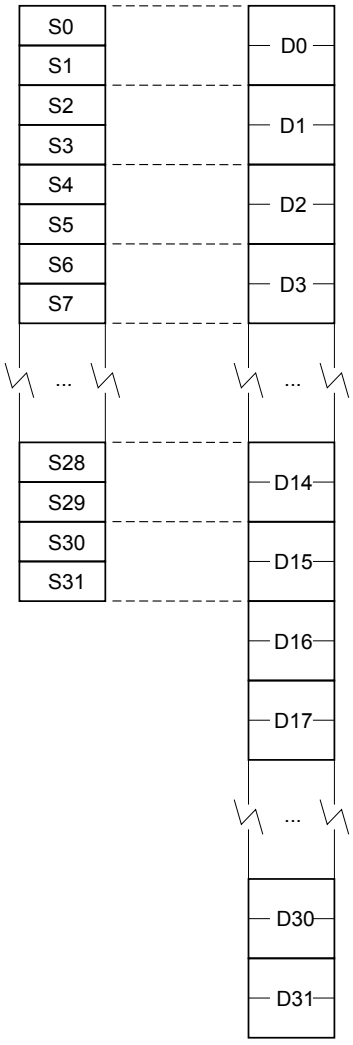


Figure B2-1 Extension register bank for floating-point in AArch32 state

The aliased views enable half-precision, single-precision, and double-precision values to coexist in different non-overlapped registers at the same time.

You can also use the same overlapped registers to store half-precision, single-precision, and double-precision values at different times.

Do not attempt to use overlapped 32-bit and 64-bit registers at the same time because it creates meaningless results.

The mapping between the registers is as follows:

- `S<2n>` maps to the least significant half of `D<n>`
- `S<2n+1>` maps to the most significant half of `D<n>`

For example, you can access the least significant half of register `D6` by referring to `S12`, and the most significant half of `D6` by referring to `S13`.

Related concepts

B2.3 Views of the floating-point extension register bank in AArch32 state on page B2-69

B2.3 Views of the floating-point extension register bank in AArch32 state

Floating-point can have different views of the extension register bank in AArch32 state.

The floating-point extension register bank can be viewed as:

- Thirty-two 64-bit registers, `D0-D31`.
- Thirty-two 32-bit registers, `S0-S31`. Only half of the register bank is accessible in this view.
- A combination of registers from these views.

64-bit floating-point registers are called double-precision registers and can contain double-precision floating-point values. 32-bit floating-point registers are called single-precision registers and can contain either a single-precision or two half-precision floating-point values.

Related concepts

B2.2 Extension register bank mapping for floating-point in AArch32 state on page B2-67

B2.4 Load values to floating-point registers

To load a register with a floating-point immediate value, use `VMOV`. This instruction has scalar and vector forms.

Related references

VLDR pseudo-instruction (floating-point)

C4.22 VMOV (floating-point) on page C4-569

B2.5 Conditional execution of A32/T32 floating-point instructions

You can execute floating-point instructions conditionally, in the same way as most A32 and T32 instructions.

You cannot use any of the following floating-point instructions in an IT block:

- `VRINT{A, N, P, M}`.
- `VSEL`.
- `VCVT{A, N, P, M}`.
- `VMAXNM`.
- `VMINNM`.

In addition, specifying any other floating-point instruction in an IT block is deprecated.

Most A32 floating-point instructions can be conditionally executed, by appending a condition code suffix to the instruction.

Related concepts

C1.2 Conditional execution in A32 code on page C1-85

C1.3 Conditional execution in T32 code on page C1-86

Related references

C1.11 Comparison of condition code meanings in integer and floating-point code on page C1-94

C1.9 Condition code suffixes on page C1-92

B2.6 Floating-point exceptions for floating-point in A32/T32 instructions

The floating-point extension records floating-point exceptions in the FPSCR cumulative flags.

It records the following exceptions:

Invalid operation
The exception is caused if the result of an operation has no mathematical value or cannot be represented.

Division by zero
The exception is caused if a divide operation has a zero divisor and a dividend that is not zero, an infinity or a NaN.

Overflow
The exception is caused if the absolute value of the result of an operation, produced after rounding, is greater than the maximum positive normalized number for the destination precision.

Underflow
The exception is caused if the absolute value of the result of an operation, produced before rounding, is less than the minimum positive normalized number for the destination precision, and the rounded result is inexact.

Inexact
The exception is caused if the result of an operation is not equivalent to the value that would be produced if the operation were performed with unbounded precision and exponent range.

Input denormal
The exception is caused if a denormalized input operand is replaced in the computation by a zero.

The descriptions of the floating-point instructions that can cause floating-point exceptions include a subsection listing the exceptions. If there is no such subsection, that instruction cannot cause any floating-point exception.

Related concepts

B2.10 Flush-to-zero mode in floating-point on page B2-76

Related references

Chapter C4 Floating-point Instructions (32-bit) on page C4-545

Related information

Arm Architecture Reference Manual

B2.7 Floating-point data types in A32/T32 instructions

Most floating-point instructions use a data type specifier to define the size and type of data that the instruction operates on.

Data type specifiers in floating-point instructions consist of a letter indicating the type of data, usually followed by a number indicating the width. They are separated from the instruction mnemonic by a point.

The following data types are available in floating-point instructions:

16-bit

`F16`

32-bit

`F32` (or `F`)

64-bit

`F64` (or `D`)

The datatype of the second (or only) operand is specified in the instruction.

Note

- Most instructions have a restricted range of permitted data types. See the instruction descriptions for details. However, the data type description is flexible:
  - If the description specifies I, you can also use the S or U data types.
  - If only the data size is specified, you can specify a type (S, U, P or F).
  - If no data type is specified, you can specify a data type.

Related concepts

B1.8 Polynomial arithmetic over {0,1} on page B1-54

B2.8 Extended notation extension for floating-point code

`armasm` implements an extension to the architectural floating-point assembly syntax, called *extended notation*. This extension allows you to include datatype information or scalar indexes in register names.

If you use extended notation, you do not have to include the data type or scalar index information in every instruction.

Register names can be any of the following:

Untyped
The register name specifies the register, but not what datatype it contains, nor any index to a particular scalar within the register.

Untyped with scalar index
The register name specifies the register, but not what datatype it contains, It specifies an index to a particular scalar within the register.

Typed
The register name specifies the register, and what datatype it contains, but not any index to a particular scalar within the register.

Typed with scalar index
The register name specifies the register, what datatype it contains, and an index to a particular scalar within the register.

Use the `SN` and `DN` directives to define names for typed and scalar registers.

Related concepts

B2.7 Floating-point data types in A32/T32 instructions on page B2-73

B2.9 Floating-point system registers in AArch32 state

Floating-point system registers are accessible in all implementations of floating-point.

For exception levels using AArch32, the following floating-point system registers are accessible in all floating-point implementations:

- FPSCR, the floating-point status and control register.
- FPEXC, the floating-point exception register.
- FPSID, the floating-point system ID register.

A particular floating-point implementation can have additional registers. For more information, see the Technical Reference Manual for your processor.

Related information

Arm Architecture Reference Manual

B2.10 Flush-to-zero mode in floating-point

Flush-to-zero mode replaces denormalized numbers with zero. This does not comply with IEEE 754 arithmetic, but in some circumstances can improve performance considerably.

Some implementations of floating-point use support code to handle denormalized numbers. The performance of such systems, in calculations involving denormalized numbers, is much less than it is in normal calculations.

Flush-to-zero mode in floating-point always preserves the sign bit.

Related concepts

B2.12 The effects of using flush-to-zero mode in floating-point on page B2-78

Related references

B2.11 When to use flush-to-zero mode in floating-point on page B2-77
B2.13 Floating-point operations not affected by flush-to-zero mode on page B2-79

B2.11 When to use flush-to-zero mode in floating-point

You can change between flush-to-zero mode and normal mode, depending on the requirements of different parts of your code.

You can change between flush-to-zero mode and normal mode, depending on the requirements of different parts of your code.

You must select flush-to-zero mode if all the following are true:

- IEEE 754 compliance is not a requirement for your system.
- The algorithms you are using sometimes generate denormalized numbers.
- Your system uses support code to handle denormalized numbers.
- The algorithms you are using do not depend for their accuracy on the preservation of denormalized numbers.
- The algorithms you are using do not generate frequent exceptions as a result of replacing denormalized numbers with 0.

You select flush-to-zero mode by setting the FZ bit in the FPSCR to 1. You do this using the `VMRS` and `VMSR` instructions.

You can change between flush-to-zero and normal mode at any time, if different parts of your code have different requirements. Numbers already in registers are not affected by changing mode.

Related concepts

B2.10 Flush-to-zero mode in floating-point on page B2-76
B2.12 The effects of using flush-to-zero mode in floating-point on page B2-78

B2.12 The effects of using flush-to-zero mode in floating-point

In flush-to-zero mode, denormalized inputs are treated as zero. Results that are too small to be represented in a normalized number are replaced with zero.

With certain exceptions, flush-to-zero mode has the following effects on floating-point operations:

- A denormalized number is treated as 0 when used as an input to a floating-point operation. The source register is not altered.
- If the result of a single-precision floating-point operation, before rounding, is in the range -2^{-126} to $+2^{-126}$, it is replaced by 0.
- If the result of a double-precision floating-point operation, before rounding, is in the range -2^{-1022} to $+2^{-1022}$, it is replaced by 0.

In flush-to-zero mode, an Input Denormal exception occurs whenever a denormalized number is used as an operand. An Underflow exception occurs when a result is flushed-to-zero.

Related concepts

B2.10 Flush-to-zero mode in floating-point on page B2-76

Related references

B2.13 Floating-point operations not affected by flush-to-zero mode on page B2-79

B2.13 Floating-point operations not affected by flush-to-zero mode

Some floating-point instructions can be carried out on denormalized numbers even in flush-to-zero mode, without flushing the results to zero.

These instructions are as follows:

- Absolute value and negate (`VABS` and `VNEG`).
- Load and store (`VLDR` and `VSTR`).
- Load multiple and store multiple (`VLDM` and `VSTM`).
- Transfer between extension registers and general-purpose registers (`VMOV`).

Related concepts

B2.10 Flush-to-zero mode in floating-point on page B2-76

Related references

C4.2 VABS (floating-point) on page C4-549

C4.14 VLDM (floating-point) on page C4-561

C4.15 VLDR (floating-point) on page C4-562

C4.38 VSTM (floating-point) on page C4-585

C4.39 VSTR (floating-point) on page C4-586

C3.54 VLDM on page C3-449

C3.55 VLDR on page C3-450

C3.131 VSTM on page C3-526

C3.134 VSTR on page C3-531

C4.23 VMOV (between one general-purpose register and single precision floating-point register) on page C4-570

C3.70 VMOV (between two general-purpose registers and a 64-bit extension register) on page C3-465

C4.29 VNEG (floating-point) on page C4-576

C3.83 VNEG on page C3-478

Part C
A32/T32 Instruction Set Reference

Chapter C1
Condition Codes

Describes condition codes and conditional execution of A32 and T32 code.

It contains the following sections:

C1.1 Conditional instructions

A32 and T32 instructions can execute conditionally on the condition flags set by a previous instruction.

The conditional instruction can occur either:

- Immediately after the instruction that updated the flags.
- After any number of intervening instructions that have not updated the flags.

In AArch32 state, whether an instruction can be conditional or not depends on the instruction set state that the processor is in.

To make an instruction conditional, you must add a condition code suffix to the instruction mnemonic. The condition code suffix enables the processor to test a condition based on the flags. If the condition test of a conditional instruction fails, the instruction:

- Does not execute.
- Does not write any value to its destination register.
- Does not affect any of the flags.
- Does not generate any exception.

Related concepts

C1.2 Conditional execution in A32 code on page C1-85
C1.3 Conditional execution in T32 code on page C1-86

Related references

C1.10 Condition code suffixes and related flags on page C1-93
C1.5 Updates to the condition flags in A32/T32 code on page C1-88

C1.2 Conditional execution in A32 code

Almost all A32 instructions can be executed conditionally on the value of the condition flags in the APSR. You can either add a condition code suffix to the instruction or you can conditionally skip over the instruction using a conditional branch instruction.

Using conditional branch instructions to control the flow of execution can be more efficient when a series of instructions depend on the same condition.

Conditional instructions to control execution

```
; flags set by a previous instruction
    LSLEQ r0, r0, #24
    ADDEQ r0, r0, #2
    ;…
```

Conditional branch to control execution

```
; flags set by a previous instruction
    BNE over
    LSL r0, r0, #24
    ADD r0, r0, #2
over
    ;…
```

Related concepts

C1.3 Conditional execution in T32 code on page C1-86

C1.3 Conditional execution in T32 code

In T32 code, there are several ways to achieve conditional execution. You can conditionally skip over the instruction using a conditional branch instruction.

Instructions can also be conditionally executed by using either of the following:

- `CBZ` and `CBNZ`.
- The `IT` (If-Then) instruction.

The T32 `CBZ` (Conditional Branch on Zero) and `CBNZ` (Conditional Branch on Non-Zero) instructions compare the value of a register against zero and branch on the result.

`IT` is a 16-bit instruction that enables a single subsequent 16-bit T32 instruction from a restricted set to be conditionally executed, based on the value of the condition flags, and the condition code suffix specified.

Conditional instructions using IT block

```
; flags set by a previous instruction
    IT   EQ
    LSLEQ r0, r0, #24
    ;…
```

The use of the IT instruction is deprecated when any of the following are true:

- There is more than one instruction in the IT block.
- There is a 32-bit instruction in the IT block.
- The instruction in the IT block references the PC.

Related concepts

C1.2 Conditional execution in A32 code on page C1-85

Related references

C2.41 IT on page C2-169

C2.23 CBZ and CBNZ on page C2-145

C1.4 Condition flags

The N, Z, C, and V condition flags are held in the APSR.

The condition flags are held in the APSR. They are set or cleared as follows:

N
: Set to 1 when the result of the operation is negative, cleared to 0 otherwise.

Z
: Set to 1 when the result of the operation is zero, cleared to 0 otherwise.

C
: Set to 1 when the operation results in a carry, or when a subtraction results in no borrow, cleared to 0 otherwise.

V
: Set to 1 when the operation causes overflow, cleared to 0 otherwise.

C is set in one of the following ways:

- For an addition, including the comparison instruction `CMN`, C is set to 1 if the addition produced a carry (that is, an unsigned overflow), and to 0 otherwise.
- For a subtraction, including the comparison instruction `CMP`, C is set to 0 if the subtraction produced a borrow (that is, an unsigned underflow), and to 1 otherwise.
- For non-addition/subtractions that incorporate a shift operation, C is set to the last bit shifted out of the value by the shifter.
- For other non-addition/subtractions, C is normally left unchanged, but see the individual instruction descriptions for any special cases.

Overflow occurs if the result of a signed add, subtract, or compare is greater than or equal to 2^{31}, or less than -2^{31}.

Related references

C1.5 Updates to the condition flags in A32/T32 code on page C1-88
C1.10 Condition code suffixes and related flags on page C1-93

C1.5 Updates to the condition flags in A32/T32 code

In AArch32 state, the condition flags are held in the *Application Program Status Register* (APSR). You can read and modify the flags using the read-modify-write procedure.

Most A32 and T32 data processing instructions have an option to update the condition flags according to the result of the operation. Instructions with the optional S suffix update the flags. Conditional instructions that are not executed have no effect on the flags.

Which flags are updated depends on the instruction. Some instructions update all flags, and some update a subset of the flags. If a flag is not updated, the original value is preserved. The description of each instruction mentions the effect that it has on the flags.

Note

Most instructions update the condition flags only if the S suffix is specified. The instructions `CMP`, `CMN`, `TEQ`, and `TST` always update the flags.

Related concepts

C1.1 Conditional instructions on page C1-84

Related references

C1.4 Condition flags on page C1-87

C1.10 Condition code suffixes and related flags on page C1-93

Chapter C2 A32 and T32 Instructions on page C2-101

C1.6 Floating-point instructions that update the condition flags

The only A32/T32 floating-point instructions that can update the condition flags are `VCMP` and `VCMPE`. Other floating-point or Advanced SIMD instructions cannot modify the flags.

`VCMP` and `VCMPE` do not update the flags directly, but update a separate set of flags in the *Floating-Point Status and Control Register* (FPSCR). To use these flags to control conditional instructions, including conditional floating-point instructions, you must first update the condition flags yourself. To do this, copy the flags from the FPSCR into the APSR using a `VMRS` instruction:

```
VMRS APSR_nzcv, FPSCR
```

Related concepts

C1.7 Carry flag on page C1-90
C1.8 Overflow flag on page C1-91

Related references

C4.4 VCMP, VCMPE on page C4-551
C3.75 VMRS on page C3-470
C4.26 VMRS (floating-point) on page C4-573

Related information

Arm Architecture Reference Manual

C1.7 Carry flag

The carry (C) flag is set when an operation results in a carry, or when a subtraction results in no borrow.

In A32/T32 code, C is set in one of the following ways:

- For an addition, including the comparison instruction `CMN`, C is set to 1 if the addition produced a carry (that is, an unsigned overflow), and to 0 otherwise.
- For a subtraction, including the comparison instruction `CMP`, C is set to 0 if the subtraction produced a borrow (that is, an unsigned underflow), and to 1 otherwise.
- For non-additions/subtractions that incorporate a shift operation, C is set to the last bit shifted out of the value by the shifter.
- For other non-additions/subtractions, C is normally left unchanged, but see the individual instruction descriptions for any special cases.
- The floating-point compare instructions, `VCMP` and `VCMPE` set the C flag and the other condition flags in the FPSCR to the result of the comparison.

Related concepts

C1.8 Overflow flag on page C1-91

Related references

A1.8 Predeclared core register names in AArch32 state on page A1-34
C1.10 Condition code suffixes and related flags on page C1-93
C1.5 Updates to the condition flags in A32/T32 code on page C1-88

C1.8 Overflow flag

Overflow can occur for add, subtract, and compare operations.

In A32/T32 code, overflow occurs if the result of the operation is greater than or equal to 2^{31}, or less than -2^{31}.

Related concepts

C1.7 Carry flag on page C1-90

Related references

A1.8 Predeclared core register names in AArch32 state on page A1-34
C1.5 Updates to the condition flags in A32/T32 code on page C1-88

C1.9 Condition code suffixes

Instructions that can be conditional have an optional two character condition code suffix.

Condition codes are shown in syntax descriptions as {*cond*}. The following table shows the condition codes that you can use:

Table C1-1 Condition code suffixes

Suffix	Meaning
EQ	Equal
NE	Not equal
CS	Carry set (identical to HS)
HS	Unsigned higher or same (identical to CS)
CC	Carry clear (identical to LO)
LO	Unsigned lower (identical to CC)
MI	Minus or negative result
PL	Positive or zero result
VS	Overflow
VC	No overflow
HI	Unsigned higher
LS	Unsigned lower or same
GE	Signed greater than or equal
LT	Signed less than
GT	Signed greater than
LE	Signed less than or equal
AL	Always (this is the default)

Note

The meaning of some of these condition codes depends on whether the instruction that last updated the condition flags is a floating-point or integer instruction.

Related references

C1.11 Comparison of condition code meanings in integer and floating-point code on page C1-94
C2.41 IT on page C2-169
C3.75 VMRS on page C3-470
C4.26 VMRS (floating-point) on page C4-573

C1.10 Condition code suffixes and related flags

Condition code suffixes define the conditions that must be met for the instruction to execute.

The following table shows the condition codes that you can use and the flag settings they depend on:

Table C1-2 Condition code suffixes and related flags

Suffix	Flags	Meaning
EQ	Z set	Equal
NE	Z clear	Not equal
CS or HS	C set	Higher or same (unsigned >=)
CC or LO	C clear	Lower (unsigned <)
MI	N set	Negative
PL	N clear	Positive or zero
VS	V set	Overflow
VC	V clear	No overflow
HI	C set and Z clear	Higher (unsigned >)
LS	C clear or Z set	Lower or same (unsigned <=)
GE	N and V the same	Signed >=
LT	N and V differ	Signed <
GT	Z clear, N and V the same	Signed >
LE	Z set, N and V differ	Signed <=
AL	Any	Always. This suffix is normally omitted.

The optional condition code is shown in syntax descriptions as {*cond*}. This condition is encoded in A32 instructions. For T32 instructions, the condition is encoded in a preceding `IT` instruction. An instruction with a condition code is only executed if the condition flags meet the specified condition.

The following is an example of conditional execution in A32 code:

```
ADD     r0, r1, r2    ; r0 = r1 + r2, don't update flags
ADDS    r0, r1, r2    ; r0 = r1 + r2, and update flags
ADDSCS  r0, r1, r2    ; If C flag set then r0 = r1 + r2,
; and update flags
CMP     r0, r1        ; update flags based on r0-r1.
```

Related concepts

C1.1 Conditional instructions on page C1-84

Related references

C1.4 Condition flags on page C1-87

C1.11 Comparison of condition code meanings in integer and floating-point code on page C1-94

C1.5 Updates to the condition flags in A32/T32 code on page C1-88

Chapter C2 A32 and T32 Instructions on page C2-101

C1.11 Comparison of condition code meanings in integer and floating-point code

The meaning of the condition code mnemonic suffixes depends on whether the condition flags were set by a floating-point instruction or by an A32 or T32 data processing instruction.

This is because:

- Floating-point values are never unsigned, so the unsigned conditions are not required.
- Not-a-Number (NaN) values have no ordering relationship with numbers or with each other, so additional conditions are required to account for unordered results.

The meaning of the condition code mnemonic suffixes is shown in the following table:

Table C1-3 Condition codes

Suffix	Meaning after integer data processing instruction	Meaning after floating-point instruction
EQ	Equal	Equal
NE	Not equal	Not equal, or unordered
CS	Carry set	Greater than or equal, or unordered
HS	Unsigned higher or same	Greater than or equal, or unordered
CC	Carry clear	Less than
LO	Unsigned lower	Less than
MI	Negative	Less than
PL	Positive or zero	Greater than or equal, or unordered
VS	Overflow	Unordered (at least one NaN operand)
VC	No overflow	Not unordered
HI	Unsigned higher	Greater than, or unordered
LS	Unsigned lower or same	Less than or equal
GE	Signed greater than or equal	Greater than or equal
LT	Signed less than	Less than, or unordered
GT	Signed greater than	Greater than
LE	Signed less than or equal	Less than or equal, or unordered
AL	Always (normally omitted)	Always (normally omitted)

Note

The type of the instruction that last updated the condition flags determines the meaning of the condition codes.

Related concepts

C1.1 Conditional instructions on page C1-84

Related references

C1.10 Condition code suffixes and related flags on page C1-93

C1.5 Updates to the condition flags in A32/T32 code on page C1-88

C4.4 VCMP, VCMPE on page C4-551

C3.75 VMRS on page C3-470

C4.26 VMRS (floating-point) on page C4-573

Related information

Arm Architecture Reference Manual

C1.12 Benefits of using conditional execution in A32 and T32 code

It can be more efficient to use conditional instructions rather than conditional branches.

You can use conditional execution of A32 instructions to reduce the number of branch instructions in your code, and improve code density. The `IT` instruction in T32 achieves a similar improvement.

Branch instructions are also expensive in processor cycles. On Arm processors without branch prediction hardware, it typically takes three processor cycles to refill the processor pipeline each time a branch is taken.

Some Arm processors have branch prediction hardware. In systems using these processors, the pipeline only has to be flushed and refilled when there is a misprediction.

Related concepts

C1.13 Example showing the benefits of conditional instructions in A32 and T32 code on page C1-97

C1.13 Example showing the benefits of conditional instructions in A32 and T32 code

Using conditional instructions rather than conditional branches can save both code size and cycles.

This example shows the difference between using branches and using conditional instructions. It uses the Euclid algorithm for the *Greatest Common Divisor* (gcd) to show how conditional instructions improve code size and speed.

In C the gcd algorithm can be expressed as:

```
int gcd(int a, int b)
{
    while (a != b)
      {
        if (a > b)
            a = a - b;
        else
            b = b - a;
      }
    return a;
}
```

The following examples show implementations of the gcd algorithm with and without conditional instructions.

Example of conditional execution using branches in A32 code

This example is an A32 code implementation of the gcd algorithm. It achieves conditional execution by using conditional branches, rather than individual conditional instructions:

```
gcd     CMP     r0, r1
        BEQ     end
        BLT     less
        SUBS    r0, r0, r1  ; could be SUB r0, r0, r1 for A32
        B       gcd
less
        SUBS    r1, r1, r0  ; could be SUB r1, r1, r0 for A32
        B       gcd
end
```

The code is seven instructions long because of the number of branches. Every time a branch is taken, the processor must refill the pipeline and continue from the new location. The other instructions and non-executed branches use a single cycle each.

The following table shows the number of cycles this implementation uses on an Arm7™ processor when R0 equals 1 and R1 equals 2.

Table C1-4 Conditional branches only

R0: a	R1: b	Instruction	Cycles (Arm7)
1	2	`CMP r0, r1`	1
1	2	`BEQ end`	1 (not executed)
1	2	`BLT less`	3
1	2	`SUB r1, r1, r0`	1
1	2	`B gcd`	3
1	1	`CMP r0, r1`	1
1	1	`BEQ end`	3
			Total = 13

Example of conditional execution using conditional instructions in A32 code

This example is an A32 code implementation of the gcd algorithm using individual conditional instructions in A32 code. The gcd algorithm only takes four instructions:

```
gcd
        CMP     r0, r1
        SUBGT   r0, r0, r1
        SUBLE   r1, r1, r0
        BNE     gcd
```

In addition to improving code size, in most cases this code executes faster than the version that uses only branches.

The following table shows the number of cycles this implementation uses on an Arm7 processor when R0 equals 1 and R1 equals 2.

Table C1-5 All instructions conditional

R0: a	R1: b	Instruction	Cycles (Arm7)
1	2	`CMP r0, r1`	1
1	2	`SUBGT r0,r0,r1`	1 (not executed)
1	1	`SUBLT r1,r1,r0`	1
1	1	`BNE gcd`	3
1	1	`CMP r0,r1`	1
1	1	`SUBGT r0,r0,r1`	1 (not executed)
1	1	`SUBLT r1,r1,r0`	1 (not executed)
1	1	`BNE gcd`	1 (not executed)
			Total = 10

Comparing this with the example that uses only branches:

- Replacing branches with conditional execution of all instructions saves three cycles.
- Where R0 equals R1, both implementations execute in the same number of cycles. For all other cases, the implementation that uses conditional instructions executes in fewer cycles than the implementation that uses branches only.

Example of conditional execution using conditional instructions in T32 code

You can use the `IT` instruction to write conditional instructions in T32 code. The T32 code implementation of the gcd algorithm using conditional instructions is similar to the implementation in A32 code. The implementation in T32 code is:

```
gcd
        CMP     r0, r1
        ITE     GT
        SUBGT   r0, r0, r1
        SUBLE   r1, r1, r0
        BNE     gcd
```

These instructions assemble equally well to A32 or T32 code. The assembler checks the `IT` instructions, but omits them on assembly to A32 code.

It requires one more instruction in T32 code (the `IT` instruction) than in A32 code, but the overall code size is 10 bytes in T32 code, compared with 16 bytes in A32 code.

Example of conditional execution code using branches in T32 code

In architectures before Armv6T2, there is no `IT` instruction and therefore T32 instructions cannot be executed conditionally except for the `B` branch instruction. The gcd algorithm must be written with

conditional branches and is similar to the A32 code implementation using branches, without conditional instructions.

The T32 code implementation of the gcd algorithm without conditional instructions requires seven instructions. The overall code size is 14 bytes. This figure is even less than the A32 implementation that uses conditional instructions, which uses 16 bytes.

In addition, on a system using 16-bit memory this T32 implementation runs faster than both A32 implementations because only one memory access is required for each 16-bit T32 instruction, whereas each 32-bit A32 instruction requires two fetches.

Related concepts

C1.12 Benefits of using conditional execution in A32 and T32 code on page C1-96

C1.14 Optimization for execution speed on page C1-100

Related references

C2.41 IT on page C2-169

C1.10 Condition code suffixes and related flags on page C1-93

Related information

Arm Architecture Reference Manual

C1.14 Optimization for execution speed

To optimize code for execution speed you must have detailed knowledge of the instruction timings, branch prediction logic, and cache behavior of your target system.

For more information, see the Technical Reference Manual for your processor.

Related information

Arm Architecture Reference Manual

Further reading

Chapter C2
A32 and T32 Instructions

Describes the A32 and T32 instructions supported in AArch32 state.

It contains the following sections:

C2.1 A32 and T32 instruction summary

An overview of the instructions available in the A32 and T32 instruction sets.

Table C2-1 Summary of instructions

Mnemonic	Brief description
ADC, ADD	Add with Carry, Add
ADR	Load program or register-relative address (short range)
AND	Logical AND
ASR	Arithmetic Shift Right
B	Branch
BFC, BFI	Bit Field Clear and Insert
BIC	Bit Clear
BKPT	Software breakpoint
BL	Branch with Link
BLX, BLXNS	Branch with Link, change instruction set, Branch with Link and Exchange (Non-secure)
BX, BXNS	Branch, change instruction set, Branch and Exchange (Non-secure)
CBZ, CBNZ	Compare and Branch if {Non}Zero
CDP	Coprocessor Data Processing operation
CDP2	Coprocessor Data Processing operation
CLREX	Clear Exclusive
CLZ	Count leading zeros
CMN, CMP	Compare Negative, Compare
CPS	Change Processor State
CRC32	CRC32
CRC32C	CRC32C
CSDB	Consumption of Speculative Data Barrier
DBG	Debug
DCPS1	Debug switch to exception level 1
DCPS2	Debug switch to exception level 2
DCPS3	Debug switch to exception level 3
DMB, DSB	Data Memory Barrier, Data Synchronization Barrier
DSB	Data Synchronization Barrier
EOR	Exclusive OR
ERET	Exception Return
ESB	Error Synchronization Barrier
HLT	Halting breakpoint
HVC	Hypervisor Call

Table C2-1 Summary of instructions (continued)

Mnemonic	Brief description
ISB	Instruction Synchronization Barrier
IT	If-Then
LDAEX, LDAEXB, LDAEXH, LDAEXD	Load-Acquire Register Exclusive Word, Byte, Halfword, Doubleword
LDC, LDC2	Load Coprocessor
LDM	Load Multiple registers
LDR	Load Register with word
LDA, LDAB, LDAH	Load-Acquire Register Word, Byte, Halfword
LDRB	Load Register with Byte
LDRBT	Load Register with Byte, user mode
LDRD	Load Registers with two words
LDREX, LDREXB, LDREXH, LDREXD	Load Register Exclusive Word, Byte, Halfword, Doubleword
LDRH	Load Register with Halfword
LDRHT	Load Register with Halfword, user mode
LDRSB	Load Register with Signed Byte
LDRSBT	Load Register with Signed Byte, user mode
LDRSH	Load Register with Signed Halfword
LDRSHT	Load Register with Signed Halfword, user mode
LDRT	Load Register with word, user mode
LSL, LSR	Logical Shift Left, Logical Shift Right
MCR	Move from Register to Coprocessor
MCRR	Move from Registers to Coprocessor
MLA	Multiply Accumulate
MLS	Multiply and Subtract
MOV	Move
MOVT	Move Top
MRC	Move from Coprocessor to Register
MRRC	Move from Coprocessor to Registers
MRS	Move from PSR to Register
MSR	Move from Register to PSR
MUL	Multiply
MVN	Move Not
NOP	No Operation
ORN	Logical OR NOT
ORR	Logical OR
PKHBT, PKHTB	Pack Halfwords

Table C2-1 Summary of instructions (continued)

Mnemonic	Brief description
PLD	Preload Data
PLDW	Preload Data with intent to Write
PLI	Preload Instruction
PUSH, POP	PUSH registers to stack, POP registers from stack
QADD, QDADD, QDSUB, QSUB	Saturating arithmetic
QADD8, QADD16, QASX, QSUB8, QSUB16, QSAX	Parallel signed saturating arithmetic
RBIT	Reverse Bits
REV, REV16, REVSH	Reverse byte order
RFE	Return From Exception
ROR	Rotate Right Register
RRX	Rotate Right with Extend
RSB	Reverse Subtract
RSC	Reverse Subtract with Carry
SADD8, SADD16, SASX	Parallel Signed arithmetic
SBC	Subtract with Carry
SBFX, UBFX	Signed, Unsigned Bit Field eXtract
SDIV	Signed Divide
SEL	Select bytes according to APSR GE flags
SETEND	Set Endianness for memory accesses
SETPAN	Set Privileged Access Never
SEV	Set Event
SEVL	Set Event Locally
SG	Secure Gateway
SHADD8, SHADD16, SHASX, SHSUB8, SHSUB16, SHSAX	Parallel Signed Halving arithmetic
SMC	Secure Monitor Call
SMLAxy	Signed Multiply with Accumulate (32 <= 16 x 16 + 32)
SMLAD	Dual Signed Multiply Accumulate
	(32 <= 32 + 16 x 16 + 16 x 16)
SMLAL	Signed Multiply Accumulate (64 <= 64 + 32 x 32)
SMLALxy	Signed Multiply Accumulate (64 <= 64 + 16 x 16)
SMLALD	Dual Signed Multiply Accumulate Long
	(64 <= 64 + 16 x 16 + 16 x 16)
SMLAWy	Signed Multiply with Accumulate (32 <= 32 x 16 + 32)

Table C2-1 Summary of instructions (continued)

Mnemonic	Brief description
SMLSD	Dual Signed Multiply Subtract Accumulate
	(32 <= 32 + 16 x 16 – 16 x 16)
SMLSLD	Dual Signed Multiply Subtract Accumulate Long
	(64 <= 64 + 16 x 16 – 16 x 16)
SMMLA	Signed top word Multiply with Accumulate (32 <= TopWord(32 x 32 + 32))
SMMLS	Signed top word Multiply with Subtract (32 <= TopWord(32 - 32 x 32))
SMMUL	Signed top word Multiply (32 <= TopWord(32 x 32))
SMUAD, SMUSD	Dual Signed Multiply, and Add or Subtract products
SMULxy	Signed Multiply (32 <= 16 x 16)
SMULL	Signed Multiply (64 <= 32 x 32)
SMULWy	Signed Multiply (32 <= 32 x 16)
SRS	Store Return State
SSAT	Signed Saturate
SSAT16	Signed Saturate, parallel halfwords
SSUB8, SSUB16, SSAX	Parallel Signed arithmetic
STC	Store Coprocessor
STM	Store Multiple registers
STR	Store Register with word
STRB	Store Register with Byte
STRBT	Store Register with Byte, user mode
STRD	Store Registers with two words
STREX, STREXB, STREXH,STREXD	Store Register Exclusive Word, Byte, Halfword, Doubleword
STRH	Store Register with Halfword
STRHT	Store Register with Halfword, user mode
STL, STLB, STLH	Store-Release Word, Byte, Halfword
STLEX, STLEXB, STLEXH, STLEXD	Store-Release Exclusive Word, Byte, Halfword, Doubleword
STRT	Store Register with word, user mode
SUB	Subtract
SUBS pc, lr	Exception return, no stack
SVC (formerly SWI)	Supervisor Call
SXTAB, SXTAB16, SXTAH	Signed extend, with Addition
SXTB, SXTH	Signed extend
SXTB16	Signed extend
SYS	Execute System coprocessor instruction
TBB, TBH	Table Branch Byte, Halfword

Table C2-1 Summary of instructions (continued)

Mnemonic	Brief description
TEQ	Test Equivalence
TST	Test
TT, TTT, TTA, TTAT	Test Target (Alternate Domain, Unprivileged)
UADD8, UADD16, UASX	Parallel Unsigned arithmetic
UDF	Permanently Undefined
UDIV	Unsigned Divide
UHADD8, UHADD16, UHASX, UHSUB8, UHSUB16, UHSAX	Parallel Unsigned Halving arithmetic
UMAAL	Unsigned Multiply Accumulate Accumulate Long
	(64 <= 32 + 32 + 32 x 32)
UMLAL, UMULL	Unsigned Multiply Accumulate, Unsigned Multiply
	(64 <= 32 x 32 + 64), (64 <= 32 x 32)
UQADD8, UQADD16, UQASX, UQSUB8, UQSUB16, UQSAX	Parallel Unsigned Saturating arithmetic
USAD8	Unsigned Sum of Absolute Differences
USADA8	Accumulate Unsigned Sum of Absolute Differences
USAT	Unsigned Saturate
USAT16	Unsigned Saturate, parallel halfwords
USUB8, USUB16, USAX	Parallel Unsigned arithmetic
UXTAB, UXTAB16, UXTAH	Unsigned extend with Addition
UXTB, UXTH	Unsigned extend
UXTB16	Unsigned extend
V*	See *Chapter C3 Advanced SIMD Instructions (32-bit)* on page C3-387 and *Chapter C4 Floating-point Instructions (32-bit)* on page C4-545
WFE, WFI, YIELD	Wait For Event, Wait For Interrupt, Yield

C2.2 Instruction width specifiers

The instruction width specifiers `.W` and `.N` control the size of T32 instruction encodings.

In T32 code the `.W` width specifier forces the assembler to generate a 32-bit encoding, even if a 16-bit encoding is available. The `.W` specifier has no effect when assembling to A32 code.

In T32 code the `.N` width specifier forces the assembler to generate a 16-bit encoding. In this case, if the instruction cannot be encoded in 16 bits or if `.N` is used in A32 code, the assembler generates an error.

If you use an instruction width specifier, you must place it immediately after the instruction mnemonic and any condition code, for example:

```
BCS.W   label   ; forces 32-bit instruction even for a short branch
B.N     label   ; faults if label out of range for 16-bit instruction
```

C2.3 Flexible second operand (Operand2)

Many A32 and T32 general data processing instructions have a flexible second operand.

This is shown as `Operand2` in the descriptions of the syntax of each instruction.

`Operand2` can be a:

- Constant.
- Register with optional shift.

Related concepts

C2.6 Shift operations on page C2-115

Related references

C2.4 Syntax of Operand2 as a constant on page C2-113

C2.5 Syntax of Operand2 as a register with optional shift on page C2-114

C2.4 Syntax of Operand2 as a constant

An Operand2 constant in an instruction has a limited range of values.

Syntax

`#constant`

where *`constant`* is an expression evaluating to a numeric value.

Usage

In A32 instructions, *`constant`* can have any value that can be produced by rotating an 8-bit value right by any even number of bits within a 32-bit word.

In T32 instructions, *`constant`* can be:

- Any constant that can be produced by shifting an 8-bit value left by any number of bits within a 32-bit word.
- Any constant of the form `0x00XY00XY`.
- Any constant of the form `0xXY00XY00`.
- Any constant of the form `0xXYXYXYXY`.

Note

In these constants, `X` and `Y` are hexadecimal digits.

In addition, in a small number of instructions, *`constant`* can take a wider range of values. These are listed in the individual instruction descriptions.

When an Operand2 constant is used with the instructions `MOVS`, `MVNS`, `ANDS`, `ORRS`, `ORNS`, `EORS`, `BICS`, `TEQ` or `TST`, the carry flag is updated to bit[31] of the constant, if the constant is greater than 255 and can be produced by shifting an 8-bit value. These instructions do not affect the carry flag if Operand2 is any other constant.

Instruction substitution

If the value of an Operand2 constant is not available, but its logical inverse or negation is available, then the assembler produces an equivalent instruction and inverts or negates the constant.

For example, an assembler might assemble the instruction `CMP` *`Rd`*, `#0xFFFFFFFE` as the equivalent instruction `CMN` *`Rd`*, `#0x2`.

Be aware of this when comparing disassembly listings with source code.

Related concepts

C2.6 Shift operations on page C2-115

Related references

C2.3 Flexible second operand (Operand2) on page C2-112
C2.5 Syntax of Operand2 as a register with optional shift on page C2-114

C2.5 Syntax of Operand2 as a register with optional shift

When you use an Operand2 register in an instruction, you can optionally also specify a shift value.

Syntax

Rm {, *shift*}

where:

Rm

is the register holding the data for the second operand.

shift

is an optional constant or register-controlled shift to be applied to *Rm*. It can be one of:

ASR #*n*

arithmetic shift right *n* bits, $1 \le n \le 32$.

LSL #*n*

logical shift left *n* bits, $1 \le n \le 31$.

LSR #*n*

logical shift right *n* bits, $1 \le n \le 32$.

ROR #*n*

rotate right *n* bits, $1 \le n \le 31$.

RRX

rotate right one bit, with extend.

type Rs

register-controlled shift is available in Arm code only, where:

type

is one of ASR, LSL, LSR, ROR.

Rs

is a register supplying the shift amount, and only the least significant byte is used.

-

if omitted, no shift occurs, equivalent to LSL #0.

Usage

If you omit the shift, or specify LSL #0, the instruction uses the value in *Rm*.

If you specify a shift, the shift is applied to the value in *Rm*, and the resulting 32-bit value is used by the instruction. However, the contents of the register *Rm* remain unchanged. Specifying a register with shift also updates the carry flag when used with certain instructions.

Related concepts

C2.6 Shift operations on page C2-115

Related references

C2.3 Flexible second operand (Operand2) on page C2-112
C2.4 Syntax of Operand2 as a constant on page C2-113

C2.6 Shift operations

Register shift operations move the bits in a register left or right by a specified number of bits, called the shift length.

Register shift can be performed:

- Directly by the instructions `ASR`, `LSR`, `LSL`, `ROR`, and `RRX`, and the result is written to a destination register.
- During the calculation of *`Operand2`* by the instructions that specify the second operand as a register with shift. The result is used by the instruction.

The permitted shift lengths depend on the shift type and the instruction, see the individual instruction description or the flexible second operand description. If the shift length is 0, no shift occurs. Register shift operations update the carry flag except when the specified shift length is 0.

Arithmetic shift right (ASR)

Arithmetic shift right by n bits moves the left-hand 32-n bits of a register to the right by n places, into the right-hand 32-n bits of the result. It copies the original bit[31] of the register into the left-hand n bits of the result.

You can use the `ASR #`*`n`* operation to divide the value in the register *`Rm`* by 2^n, with the result being rounded towards negative-infinity.

When the instruction is `ASRS` or when `ASR #`*`n`* is used in *`Operand2`* with the instructions `MOVS`, `MVNS`, `ANDS`, `ORRS`, `ORNS`, `EORS`, `BICS`, `TEQ` or `TST`, the carry flag is updated to the last bit shifted out, bit[*n*-1], of the register *`Rm`*.

Note

- If *n* is 32 or more, then all the bits in the result are set to the value of bit[31] of *`Rm`*.
- If *n* is 32 or more and the carry flag is updated, it is updated to the value of bit[31] of *`Rm`*.

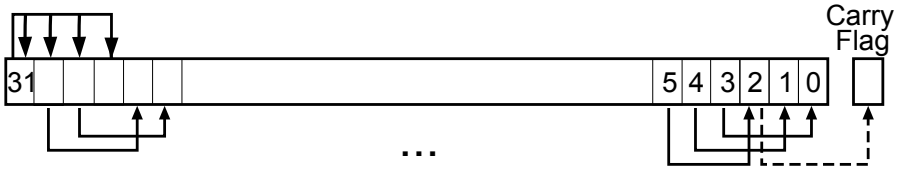


Figure C2-1 ASR #3

Logical shift right (LSR)

Logical shift right by n bits moves the left-hand 32-n bits of a register to the right by n places, into the right-hand 32-n bits of the result. It sets the left-hand n bits of the result to 0.

You can use the `LSR #`*`n`* operation to divide the value in the register *`Rm`* by 2^n, if the value is regarded as an unsigned integer.

When the instruction is `LSRS` or when `LSR #`*`n`* is used in *`Operand2`* with the instructions `MOVS`, `MVNS`, `ANDS`, `ORRS`, `ORNS`, `EORS`, `BICS`, `TEQ` or `TST`, the carry flag is updated to the last bit shifted out, bit[*n*-1], of the register *`Rm`*.

Note

- If *n* is 32 or more, then all the bits in the result are cleared to 0.
- If *n* is 33 or more and the carry flag is updated, it is updated to 0.

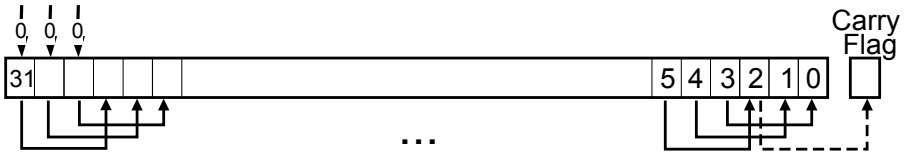


Figure C2-2 LSR #3

Logical shift left (LSL)

Logical shift left by n bits moves the right-hand 32-n bits of a register to the left by n places, into the left-hand 32-n bits of the result. It sets the right-hand *n* bits of the result to 0.

You can use the `LSL #n` operation to multiply the value in the register *Rm* by 2^n, if the value is regarded as an unsigned integer or a two's complement signed integer. Overflow can occur without warning.

When the instruction is `LSLS` or when `LSL #n`, with non-zero *n*, is used in *Operand2* with the instructions `MOVS`, `MVNS`, `ANDS`, `ORRS`, `ORNS`, `EORS`, `BICS`, `TEQ` or `TST`, the carry flag is updated to the last bit shifted out, bit[32-*n*], of the register *Rm*. These instructions do not affect the carry flag when used with `LSL #0`.

Note

- If *n* is 32 or more, then all the bits in the result are cleared to 0.
- If *n* is 33 or more and the carry flag is updated, it is updated to 0.

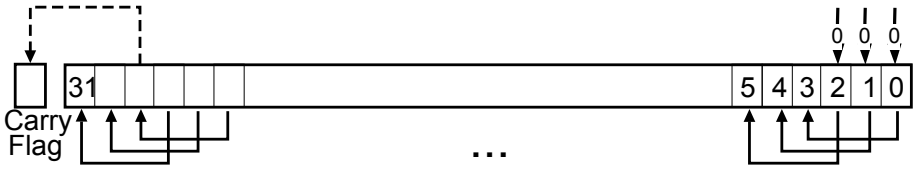


Figure C2-3 LSL #3

Rotate right (ROR)

Rotate right by n bits moves the left-hand 32-n bits of a register to the right by n places, into the right-hand 32-n bits of the result. It also moves the right-hand n bits of the register into the left-hand n bits of the result.

When the instruction is `RORS` or when `ROR #n` is used in *Operand2* with the instructions `MOVS`, `MVNS`, `ANDS`, `ORRS`, `ORNS`, `EORS`, `BICS`, `TEQ` or `TST`, the carry flag is updated to the last bit rotation, bit[*n*-1], of the register *Rm*.

Note

- If *n* is 32, then the value of the result is same as the value in *Rm*, and if the carry flag is updated, it is updated to bit[31] of *Rm*.
- `ROR` with shift length, *n*, more than 32 is the same as `ROR` with shift length *n*-32.

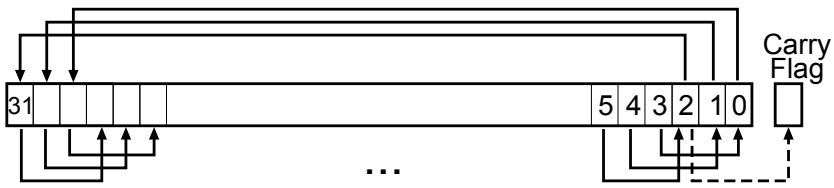


Figure C2-4 ROR #3

Rotate right with extend (RRX)

Rotate right with extend moves the bits of a register to the right by one bit. It copies the carry flag into bit[31] of the result.

When the instruction is `RRXS` or when `RRX` is used in *`Operand2`* with the instructions `MOVS`, `MVNS`, `ANDS`, `ORRS`, `ORNS`, `EORS`, `BICS`, `TEQ` or `TST`, the carry flag is updated to bit[0] of the register *`Rm`*.

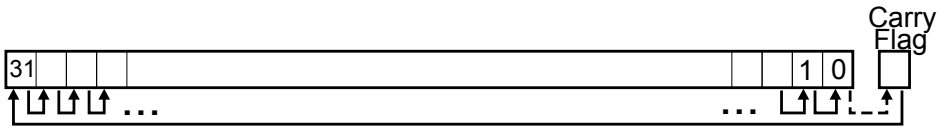


Figure C2-5 RRX

Related references

C2.3 Flexible second operand (Operand2) on page C2-112

C2.4 Syntax of Operand2 as a constant on page C2-113

C2.5 Syntax of Operand2 as a register with optional shift on page C2-114

C2.7 Saturating instructions

Some A32 and T32 instructions perform saturating arithmetic.

The saturating instructions are:

- `QADD`.
- `QDADD`.
- `QDSUB`.
- `QSUB`.
- `SSAT`.
- `USAT`.

Some of the parallel instructions are also saturating.

Saturating arithmetic

Saturation means that, for some value of 2^n that depends on the instruction:

- For a signed saturating operation, if the full result would be less than -2^n, the result returned is -2^n.
- For an unsigned saturating operation, if the full result would be negative, the result returned is zero.
- If the full result would be greater than 2^n-1, the result returned is 2^n-1.

When any of these occurs, it is called saturation. Some instructions set the Q flag when saturation occurs.

Note

Saturating instructions do not clear the Q flag when saturation does not occur. To clear the Q flag, use an `MSR` instruction.

The Q flag can also be set by two other instructions, but these instructions do not saturate.

Related references

C2.75 QADD on page C2-223
C2.82 QSUB on page C2-230
C2.79 QDADD on page C2-227
C2.80 QDSUB on page C2-228
C2.113 SMLAxy on page C2-271
C2.118 SMLAWy on page C2-278
C2.125 SMULxy on page C2-285
C2.127 SMULWy on page C2-287
C2.130 SSAT on page C2-291
C2.181 USAT on page C2-366
C2.65 MSR (general-purpose register to PSR) on page C2-208

C2.8 ADC

Add with Carry.

Syntax

`ADC{S}{cond} {Rd}, Rn, Operand2`

where:

`S`
: is an optional suffix. If `S` is specified, the condition flags are updated on the result of the operation.

`cond`
: is an optional condition code.

`Rd`
: is the destination register.

`Rn`
: is the register holding the first operand.

`Operand2`
: is a flexible second operand.

Usage

The `ADC` (Add with Carry) instruction adds the values in *`Rn`* and *`Operand2`*, together with the carry flag.

You can use `ADC` to synthesize multiword arithmetic.

In certain circumstances, the assembler can substitute one instruction for another. Be aware of this when reading disassembly listings.

Use of PC and SP in T32 instructions

You cannot use PC (`R15`) for *`Rd`*, or any operand with the `ADC` command.

You cannot use SP (`R13`) for *`Rd`*, or any operand with the `ADC` command.

Use of PC and SP in A32 instructions

You cannot use PC for *`Rd`* or any operand in any data processing instruction that has a register-controlled shift.

Use of PC for any operand, in instructions without register-controlled shift, is deprecated.

If you use PC (`R15`) as *`Rn`* or *`Operand2`*, the value used is the address of the instruction plus 8.

If you use PC as *`Rd`*:

- Execution branches to the address corresponding to the result.
- If you use the `S` suffix, see the `SUBS pc,lr` instruction.

Use of SP with the `ADC` A32 instruction is deprecated.

Condition flags

If `S` is specified, the `ADC` instruction updates the N, Z, C and V flags according to the result.

16-bit instructions

The following forms of this instruction are available in T32 code, and are 16-bit instructions:

`ADCS` ***`Rd, Rd, Rm`***

`Rd` and *`Rm`* must both be Lo registers. This form can only be used outside an IT block.

ADC{*cond*} *Rd*, *Rd*, *Rm*

Rd and *Rm* must both be Lo registers. This form can only be used inside an IT block.

Multiword arithmetic examples

These two instructions add a 64-bit integer contained in `R2` and `R3` to another 64-bit integer contained in `R0` and `R1`, and place the result in `R4` and `R5`.

```
    ADDS    r4, r0, r2    ; adding the least significant words
    ADC     r5, r1, r3    ; adding the most significant words
```

Related references

C2.3 Flexible second operand (Operand2) on page C2-112
C1.9 Condition code suffixes on page C1-92

C2.9 ADD

Add without Carry.

Syntax

`ADD{S}{cond} {Rd}, Rn, Operand2`

`ADD{cond} {Rd}, Rn, #imm12 ; T32, 32-bit encoding only`

where:

`S`
: is an optional suffix. If `S` is specified, the condition flags are updated on the result of the operation.

`cond`
: is an optional condition code.

`Rd`
: is the destination register.

`Rn`
: is the register holding the first operand.

`Operand2`
: is a flexible second operand.

`imm12`
: is any value in the range 0-4095.

Operation

The `ADD` instruction adds the values in *`Rn`* and *`Operand2`* or *`imm12`*.

In certain circumstances, the assembler can substitute one instruction for another. Be aware of this when reading disassembly listings.

Use of PC and SP in T32 instructions

Generally, you cannot use PC (`R15`) for *`Rd`*, or any operand.

The exceptions are:

- you can use PC for *`Rn`* in 32-bit encodings of T32 `ADD` instructions, with a constant *`Operand2`* value in the range 0-4095, and no `S` suffix. These instructions are useful for generating PC-relative addresses. Bit[1] of the PC value reads as 0 in this case, so that the base address for the calculation is always word-aligned.
- you can use PC in 16-bit encodings of T32 `ADD{cond} Rd, Rd, Rm` instructions, where both registers cannot be PC. However, the following 16-bit T32 instructions are deprecated:
  — `ADD{cond} PC, SP, PC`.
  — `ADD{cond} SP, SP, PC`.

Generally, you cannot use SP (`R13`) for *`Rd`*, or any operand. Except that:

- You can use SP for `Rn` in `ADD` instructions.
- `ADD{cond} SP, SP, SP` is permitted but is deprecated in Armv6T2 and above.
- `ADD{S}{cond} SP, SP, Rm{,shift}` and `SUB{S}{cond} SP, SP, Rm{,shift}` are permitted if *`shift`* is omitted or `LSL #1`, `LSL #2`, or `LSL #3`.

Use of PC and SP in A32 instructions

You cannot use PC for *`Rd`* or any operand in any data processing instruction that has a register-controlled shift.

In `ADD` instructions without register-controlled shift, use of PC is deprecated except for the following cases:

- Use of PC for `Rd` in instructions that do not add SP to a register.
- Use of PC for `Rn` and use of PC for `Rm` in instructions that add two registers other than SP.
- Use of PC for `Rn` in the instruction `ADD{cond} Rd, Rn, #Constant`.

If you use PC (`R15`) as *Rn* or *Rm*, the value used is the address of the instruction plus 8.

If you use PC as *Rd*:
- Execution branches to the address corresponding to the result.
- If you use the `S` suffix, see the `SUBS pc,lr` instruction.

You can use SP for *Rn* in `ADD` instructions, however, `ADDS PC, SP, #Constant` is deprecated.

You can use SP in `ADD` (register) if *Rn* is SP and *shift* is omitted or `LSL #1`, `LSL #2`, or `LSL #3`.

Other uses of SP in these A32 instructions are deprecated.

Condition flags

If `S` is specified, these instructions update the N, Z, C and V flags according to the result.

16-bit instructions

The following forms of these instructions are available in T32 code, and are 16-bit instructions:

`ADDS Rd, Rn, #imm`
: *imm* range 0-7. *Rd* and *Rn* must both be Lo registers. This form can only be used outside an IT block.

`ADD{cond} Rd, Rn, #imm`
: *imm* range 0-7. *Rd* and *Rn* must both be Lo registers. This form can only be used inside an IT block.

`ADDS Rd, Rn, Rm`
: *Rd*, *Rn* and *Rm* must all be Lo registers. This form can only be used outside an IT block.

`ADD{cond} Rd, Rn, Rm`
: *Rd*, *Rn* and *Rm* must all be Lo registers. This form can only be used inside an IT block.

`ADDS Rd, Rd, #imm`
: *imm* range 0-255. *Rd* must be a Lo register. This form can only be used outside an IT block.

`ADD{cond} Rd, Rd, #imm`
: *imm* range 0-255. *Rd* must be a Lo register. This form can only be used inside an IT block.

`ADD SP, SP, #imm`
: *imm* range 0-508, word aligned.

`ADD Rd, SP, #imm`
: *imm* range 0-1020, word aligned. *Rd* must be a Lo register.

`ADD Rd, pc, #imm`
: *imm* range 0-1020, word aligned. *Rd* must be a Lo register. Bits[1:0] of the PC are read as 0 in this instruction.

Example

```
    ADD     r2, r1, r3
```

Multiword arithmetic example

These two instructions add a 64-bit integer contained in `R2` and `R3` to another 64-bit integer contained in `R0` and `R1`, and place the result in `R4` and `R5`.

```
    ADDS    r4, r0, r2    ; adding the least significant words
    ADC     r5, r1, r3    ; adding the most significant words
```

Related references

C2.3 Flexible second operand (Operand2) on page C2-112
C1.9 Condition code suffixes on page C1-92
C2.144 SUBS pc, lr on page C2-317

C2.10 ADR (PC-relative)

Generate a PC-relative address in the destination register, for a label in the current area.

Syntax

`ADR{cond}{.W} Rd,label`

where:

cond
is an optional condition code.

`.W`
is an optional instruction width specifier.

`Rd`
is the destination register to load.

`label`
is a PC-relative expression.

`label` must be within a limited distance of the current instruction.

Usage

`ADR` produces position-independent code, because the assembler generates an instruction that adds or subtracts a value to the PC.

`label` must evaluate to an address in the same assembler area as the `ADR` instruction.

If you use `ADR` to generate a target for a `BX` or `BLX` instruction, it is your responsibility to set the T32 bit (bit 0) of the address if the target contains T32 instructions.

Offset range and architectures

The assembler calculates the offset from the PC for you. The assembler generates an error if *`label`* is out of range.

The following table shows the possible offsets between the label and the current instruction:

Table C2-2 PC-relative offsets

Instruction	Offset range
A32 `ADR`	See *C2.4 Syntax of Operand2 as a constant* on page C2-113.
T32 `ADR`, 32-bit encoding	±4095
T32 `ADR`, 16-bit encoding [a]	0-1020 [b]

ADR in T32

You can use the `.W` width specifier to force `ADR` to generate a 32-bit instruction in T32 code. `ADR` with `.W` always generates a 32-bit instruction, even if the address can be generated in a 16-bit instruction.

For forward references, `ADR` without `.W` always generates a 16-bit instruction in T32 code, even if that results in failure for an address that could be generated in a 32-bit T32 `ADD` instruction.

Restrictions

In T32 code, `Rd` cannot be `PC` or `SP`.

In A32 code, `Rd` can be `PC` or `SP` but use of `SP` is deprecated.

[a] Rd must be in the range R0-R7.
[b] Must be a multiple of 4.

Related references

C2.4 Syntax of Operand2 as a constant on page C2-113
C1.9 Condition code suffixes on page C1-92

C2.11 ADR (register-relative)

Generate a register-relative address in the destination register, for a label defined in a storage map.

Syntax

`ADR{cond}{.W} Rd,label`

where:

cond
: is an optional condition code.

.W
: is an optional instruction width specifier.

Rd
: is the destination register to load.

label
: is a symbol defined by the `FIELD` directive. *label* specifies an offset from the base register which is defined using the `MAP` directive.

  label must be within a limited distance from the base register.

Usage

`ADR` generates code to easily access named fields inside a storage map.

Restrictions

In T32 code:

- `Rd` cannot be `PC`.
- `Rd` can be `SP` only if the base register is `SP`.

Offset range and architectures

The assembler calculates the offset from the base register for you. The assembler generates an error if *label* is out of range.

The following table shows the possible offsets between the label and the current instruction:

Table C2-3 Register-relative offsets

Instruction	Offset range
A32 `ADR`	See *C2.4 Syntax of Operand2 as a constant* on page C2-113
T32 `ADR`, 32-bit encoding	±4095
T32 `ADR`, 16-bit encoding, base register is SP [c]	0-1020 [d]

ADR in T32

You can use the `.W` width specifier to force `ADR` to generate a 32-bit instruction in T32 code. `ADR` with `.W` always generates a 32-bit instruction, even if the address can be generated in a 16-bit instruction.

For forward references, `ADR` without `.W`, with base register SP, always generates a 16-bit instruction in T32 code, even if that results in failure for an address that could be generated in a 32-bit T32 `ADD` instruction.

Related references

C2.4 Syntax of Operand2 as a constant on page C2-113

[c] *Rd* must be in the range R0-R7 or SP. If *Rd* is SP, the offset range is -508 to 508 and must be a multiple of 4
[d] Must be a multiple of 4.

C1.9 Condition code suffixes on page C1-92

C2.12 AND

Logical AND.

Syntax

`AND{S}{cond} Rd, Rn, Operand2`

where:

`S`
is an optional suffix. If `S` is specified, the condition flags are updated on the result of the operation.

`cond`
is an optional condition code.

`Rd`
is the destination register.

`Rn`
is the register holding the first operand.

`Operand2`
is a flexible second operand.

Operation

The `AND` instruction performs bitwise AND operations on the values in *`Rn`* and *`Operand2`*.

In certain circumstances, the assembler can substitute `BIC` for `AND`, or `AND` for `BIC`. Be aware of this when reading disassembly listings.

Use of PC in T32 instructions

You cannot use PC (`R15`) for *`Rd`* or any operand with the `AND` instruction.

Use of PC and SP in A32 instructions

You can use PC and SP with the `AND` A32 instruction but this is deprecated.

If you use PC as *`Rn`*, the value used is the address of the instruction plus 8.

If you use PC as *`Rd`*:
- Execution branches to the address corresponding to the result.
- If you use the `S` suffix, see the `SUBS pc,lr` instruction.

You cannot use PC for any operand in any data processing instruction that has a register-controlled shift.

Condition flags

If `S` is specified, the `AND` instruction:
- Updates the N and Z flags according to the result.
- Can update the C flag during the calculation of *`Operand2`*.
- Does not affect the V flag.

16-bit instructions

The following forms of this instruction are available in T32 code, and are 16-bit instructions:

`ANDS` ***`Rd, Rd, Rm`***
`Rd` and *`Rm`* must both be Lo registers. This form can only be used outside an IT block.

`AND{`*`cond`*`}` ***`Rd, Rd, Rm`***
`Rd` and *`Rm`* must both be Lo registers. This form can only be used inside an IT block.

It does not matter if you specify `AND{S}` *`Rd, Rm, Rd`*. The instruction is the same.

Examples

```
AND     r9,r2,#0xFF00
ANDS    r9, r8, #0x19
```

Related references

C2.3 Flexible second operand (Operand2) on page C2-112
C2.144 SUBS pc, lr on page C2-317
C1.9 Condition code suffixes on page C1-92

C2.13 ASR

Arithmetic Shift Right. This instruction is a preferred synonym for `MOV` instructions with shifted register operands.

Syntax

`ASR{S}{`*cond*`} Rd, Rm, Rs`

`ASR{S}{`*cond*`} Rd, Rm, #sh`

where:

`S`
: is an optional suffix. If `S` is specified, the condition flags are updated on the result of the operation.

`Rd`
: is the destination register.

`Rm`
: is the register holding the first operand. This operand is shifted right.

`Rs`
: is a register holding a shift value to apply to the value in *`Rm`*. Only the least significant byte is used.

`sh`
: is a constant shift. The range of values permitted is 1-32.

Operation

`ASR` provides the signed value of the contents of a register divided by a power of two. It copies the sign bit into vacated bit positions on the left.

Restrictions in T32 code

T32 instructions must not use PC or SP.

Use of SP and PC in A32 instructions

You can use SP in the `ASR` A32 instruction but this is deprecated.

You cannot use PC in instructions with the `ASR{S}{cond} Rd, Rm, Rs` syntax. You can use PC for `Rd` and `Rm` in the other syntax, but this is deprecated.

If you use PC as *`Rm`*, the value used is the address of the instruction plus 8.

If you use PC as *`Rd`*:

- Execution branches to the address corresponding to the result.
- If you use the `S` suffix, the SPSR of the current mode is copied to the CPSR. You can use this to return from exceptions.

  Note

  The A32 instruction `ASRS{`*cond*`} pc,Rm,#sh` always disassembles to the preferred form `MOVS{`*cond*`} pc,Rm{,shift}`.

Caution

Do not use the `S` suffix when using PC as *`Rd`* in User mode or System mode. The assembler cannot warn you about this because it has no information about what the processor mode is likely to be at execution time.

You cannot use PC for *`Rd`* or any operand in the ASR instruction if it has a register-controlled shift.

Condition flags

If `S` is specified, the ASR instruction updates the N and Z flags according to the result.

The C flag is unaffected if the shift value is 0. Otherwise, the C flag is updated to the last bit shifted out.

16-bit instructions

The following forms of these instructions are available in T32 code, and are 16-bit instructions:

`ASRS Rd, Rm, #sh`
: *`Rd`* and *`Rm`* must both be Lo registers. This form can only be used outside an IT block.

`ASR{cond} Rd, Rm, #sh`
: *`Rd`* and *`Rm`* must both be Lo registers. This form can only be used inside an IT block.

`ASRS Rd, Rd, Rs`
: *`Rd`* and *`Rs`* must both be Lo registers. This form can only be used outside an IT block.

`ASR{cond} Rd, Rd, Rs`
: *`Rd`* and *`Rs`* must both be Lo registers. This form can only be used inside an IT block.

Architectures

This instruction is available in A32 and T32.

Example

```
    ASR     r7, r8, r9
```

Related references

C2.58 MOV on page C2-199
C1.9 Condition code suffixes on page C1-92

C2.14 B

Branch.

Syntax

`B{cond}{.W} label`

where:

cond
is an optional condition code.

.W
is an optional instruction width specifier to force the use of a 32-bit `B` instruction in T32.

label
is a PC-relative expression.

Operation

The `B` instruction causes a branch to *label*.

Instruction availability and branch ranges

The following table shows the branch ranges that are available in A32 and T32 code. Instructions that are not shown in this table are not available.

Table C2-4 B instruction availability and range

Instruction	A32	T32, 16-bit encoding	T32, 32-bit encoding
`B label`	±32MB	±2KB	±16MB [e]
`B{cond} label`	±32MB	-252 to +258	±1MB [e]

Extending branch ranges

Machine-level `B` instructions have restricted ranges from the address of the current instruction. However, you can use these instructions even if *label* is out of range. Often you do not know where the linker places *label*. When necessary, the linker adds code to enable longer branches. The added code is called a veneer.

B in T32

You can use the `.W` width specifier to force `B` to generate a 32-bit instruction in T32 code.

`B.W` always generates a 32-bit instruction, even if the target could be reached using a 16-bit instruction.

For forward references, `B` without `.W` always generates a 16-bit instruction in T32 code, even if that results in failure for a target that could be reached using a 32-bit T32 instruction.

Condition flags

The `B` instruction does not change the flags.

Architectures

See the earlier table for details of availability of the `B` instruction.

Example

```
    B       loopA
```

[e] Use `.W` to instruct the assembler to use this 32-bit instruction.

Related references

C1.9 Condition code suffixes on page C1-92

C2.15 BFC

Bit Field Clear.

Syntax

`BFC{cond} Rd, #lsb, #width`

where:

cond
: is an optional condition code.

Rd
: is the destination register.

lsb
: is the least significant bit that is to be cleared.

width
: is the number of bits to be cleared. *width* must not be 0, and (*width*+*lsb*) must be less than or equal to 32.

Operation

Clears adjacent bits in a register. *width* bits in *Rd* are cleared, starting at *lsb*. Other bits in *Rd* are unchanged.

Register restrictions

You cannot use PC for any register.

You can use SP in the `BFC` A32 instruction but this is deprecated. You cannot use SP in the `BFC` T32 instruction.

Condition flags

The `BFC` instruction does not change the flags.

Architectures

This 32-bit instruction is available in A32 and T32.

There is no 16-bit version of this instruction in T32.

Related references

C1.9 Condition code suffixes on page C1-92

C2.16 BFI

Bit Field Insert.

Syntax

`BFI{`*cond*`} `*Rd*`, `*Rn*`, #`*lsb*`, #`*width*

where:

cond
: is an optional condition code.

Rd
: is the destination register.

Rn
: is the source register.

lsb
: is the least significant bit that is to be copied.

width
: is the number of bits to be copied. *width* must not be 0, and (*width*+*lsb*) must be less than or equal to 32.

Operation

Inserts adjacent bits from one register into another. *width* bits in *Rd*, starting at *lsb*, are replaced by *width* bits from *Rn*, starting at bit[0]. Other bits in *Rd* are unchanged.

Register restrictions

You cannot use PC for any register.

You can use SP in the `BFI` A32 instruction but this is deprecated. You cannot use SP in the `BFI` T32 instruction.

Condition flags

The `BFI` instruction does not change the flags.

Architectures

This 32-bit instruction is available in A32 and T32.

There is no 16-bit version of this instruction in T32.

Related references

C1.9 Condition code suffixes on page C1-92

C2.17 BIC

Bit Clear.

Syntax

`BIC{S}{cond} Rd, Rn, Operand2`

where:

`S`
: is an optional suffix. If `S` is specified, the condition flags are updated on the result of the operation.

`cond`
: is an optional condition code.

`Rd`
: is the destination register.

`Rn`
: is the register holding the first operand.

`Operand2`
: is a flexible second operand.

Operation

The `BIC` (Bit Clear) instruction performs an AND operation on the bits in *`Rn`* with the complements of the corresponding bits in the value of *`Operand2`*.

In certain circumstances, the assembler can substitute `BIC` for `AND`, or `AND` for `BIC`. Be aware of this when reading disassembly listings.

Use of PC in T32 instructions

You cannot use PC (`R15`) for *`Rd`* or any operand in a `BIC` instruction.

Use of PC and SP in A32 instructions

You can use PC and SP with the `BIC` instruction but they are deprecated.

If you use PC as *`Rn`*, the value used is the address of the instruction plus 8.

If you use PC as *`Rd`*:

- Execution branches to the address corresponding to the result.
- If you use the `S` suffix, see the `SUBS pc,lr` instruction.

You cannot use PC for any operand in any data processing instruction that has a register-controlled shift.

Condition flags

If `S` is specified, the `BIC` instruction:

- Updates the N and Z flags according to the result.
- Can update the C flag during the calculation of *`Operand2`*.
- Does not affect the V flag.

16-bit instructions

The following forms of the `BIC` instruction are available in T32 code, and are 16-bit instructions:

`BICS` ***`Rd, Rd, Rm`***
: *`Rd`* and *`Rm`* must both be Lo registers. This form can only be used outside an IT block.

`BIC{`*`cond`*`}` ***`Rd, Rd, Rm`***
: *`Rd`* and *`Rm`* must both be Lo registers. This form can only be used inside an IT block.

Example

```
BIC     r0, r1, #0xab
```

Related references

C2.3 Flexible second operand (Operand2) on page C2-112

C2.144 SUBS pc, lr on page C2-317

C1.9 Condition code suffixes on page C1-92

C2.18 BKPT

Breakpoint.

Syntax

`BKPT #imm`

where:

imm
is an expression evaluating to an integer in the range:
- 0-65535 (a 16-bit value) in an A32 instruction.
- 0-255 (an 8-bit value) in a 16-bit T32 instruction.

Usage

The `BKPT` instruction causes the processor to enter Debug state. Debug tools can use this to investigate system state when the instruction at a particular address is reached.

In both A32 state and T32 state, *imm* is ignored by the Arm hardware. However, a debugger can use it to store additional information about the breakpoint.

`BKPT` is an unconditional instruction. It must not have a condition code in A32 code. In T32 code, the `BKPT` instruction does not require a condition code suffix because `BKPT` always executes irrespective of its condition code suffix.

Architectures

This instruction is available in A32 and T32.

In T32, it is only available as a 16-bit instruction.

C2.19 BL

Branch with Link.

Syntax

`BL{cond}{.W} label`

where:

cond
: is an optional condition code. *cond* is not available on all forms of this instruction.

.W
: is an optional instruction width specifier to force the use of a 32-bit `BL` instruction in T32.

label
: is a PC-relative expression.

Operation

The `BL` instruction causes a branch to *label*, and copies the address of the next instruction into LR (`R14`, the link register).

Instruction availability and branch ranges

The following table shows the `BL` instructions that are available in A32 and T32 state. Instructions that are not shown in this table are not available.

Table C2-5 BL instruction availability and range

Instruction	A32	T32, 16-bit encoding	T32, 32-bit encoding
`BL label`	±32MB	±4MB [f]	±16MB
`BL{cond} label`	±32MB	-	-

Extending branch ranges

Machine-level `BL` instructions have restricted ranges from the address of the current instruction. However, you can use these instructions even if *label* is out of range. Often you do not know where the linker places *label*. When necessary, the linker adds code to enable longer branches. The added code is called a veneer.

Condition flags

The `BL` instruction does not change the flags.

Availability

See the preceding table for details of availability of the `BL` instruction in both instruction sets.

Examples

```
BLE     ng+8
BL      subC
BLLT    rtX
```

Related references

C1.9 Condition code suffixes on page C1-92

[f] `BL label` and `BLX label` are an instruction pair.

C2.20 BLX, BLXNS

Branch with Link and exchange instruction set and Branch with Link and Exchange (Non-secure).

Syntax

`BLX{cond}{q} label`

`BLX{cond}{q} Rm`

`BLXNS{cond}{q} Rm` (Armv8-M only)

Where:

cond
Is an optional condition code. *cond* is not available on all forms of this instruction.

q
Is an optional instruction width specifier. Must be set to `.W` when *label* is used.

label
Is a PC-relative expression.

Rm
Is a register containing an address to branch to.

Operation

The `BLX` instruction causes a branch to *label*, or to the address contained in *Rm*. In addition:

- The `BLX` instruction copies the address of the next instruction into LR (`R14`, the link register).
- The `BLX` instruction can change the instruction set.

  `BLX` *label* always changes the instruction set. It changes a processor in A32 state to T32 state, or a processor in T32 state to A32 state.

  `BLX` *Rm* derives the target instruction set from bit[0] of *Rm*:
  — If bit[0] of *Rm* is 0, the processor changes to, or remains in, A32 state.
  — If bit[0] of *Rm* is 1, the processor changes to, or remains in, T32 state.

Note

- Armv7-M and Armv6-M only support the T32 instruction set. An attempt to change the instruction execution state causes the processor to take an exception on the instruction at the target address.

The `BLXNS` instruction calls a subroutine at an address and instruction set specified by a register, and causes a transition from the Secure to the Non-secure domain. This variant of the instruction must only be used when additional steps required to make such a transition safe are taken.

Instruction availability and branch ranges

The following table shows the instructions that are available in A32 and T32 state. Instructions that are not shown in this table are not available.

Table C2-6 BLX instruction availability and range

Instruction	A32	T32, 16-bit encoding	T32, 32-bit encoding
`BLX label`	±32MB	±4MB [g]	±16MB
`BLX Rm`	Available	Available	Use 16-bit
`BLX{cond} Rm`	Available	-	-
`BLXNS`	-	Available	-

[g] `BLX label` and `BL label` are an instruction pair.

Register restrictions

You can use PC for `Rm` in the A32 `BLX` instruction, but this is deprecated. You cannot use PC in other A32 instructions.

You can use PC for `Rm` in the T32 `BLX` instruction. You cannot use PC in other T32 instructions.

You can use SP for `Rm` in this A32 instruction but this is deprecated.

You can use SP for `Rm` in the T32 `BLX` and `BLXNS` instructions, but this is deprecated. You cannot use SP in the other T32 instructions.

Condition flags

These instructions do not change the flags.

Availability

See the preceding table for details of availability of the `BLX` and `BLXNS` instructions in both instruction sets.

Related references

C1.9 Condition code suffixes on page C1-92

C2.2 Instruction width specifiers on page C2-111

C2.21 BX, BXNS

Branch and exchange instruction set and Branch and Exchange Non-secure.

Syntax

`BX{cond}{q} Rm`

`BXNS{cond}{q} Rm` (Armv8-M only)

Where:

cond
: Is an optional condition code. *cond* is not available on all forms of this instruction.

q
: Is an optional instruction width specifier.

Rm
: Is a register containing an address to branch to.

Operation

The `BX` instruction causes a branch to the address contained in *Rm* and exchanges the instruction set, if necessary. The `BX` instruction can change the instruction set.

`BX` *Rm* derives the target instruction set from bit[0] of *Rm*:
- If bit[0] of *Rm* is 0, the processor changes to, or remains in, A32 state.
- If bit[0] of *Rm* is 1, the processor changes to, or remains in, T32 state.

Note
- Armv7-M and Armv6-M only support the T32 instruction set. An attempt to change the instruction execution state causes the processor to take an exception on the instruction at the target address.

`BX` can also be used for an exception return.

The `BXNS` instruction causes a branch to an address and instruction set specified by a register, and causes a transition from the Secure to the Non-secure domain. This variant of the instruction must only be used when additional steps required to make such a transition safe are taken.

Instruction availability and branch ranges

The following table shows the instructions that are available in A32 and T32 state. Instructions that are not shown in this table are not available.

Table C2-7 BX instruction availability and range

Instruction	A32	T32, 16-bit encoding	T32, 32-bit encoding
`BX Rm`	Available	Available	Use 16-bit
`BX{cond} Rm`	Available	-	-
`BXNS`	-	Available	-

Register restrictions

You can use PC for `Rm` in the A32 `BX` instruction, but this is deprecated. You cannot use PC in other A32 instructions.

You can use PC for `Rm` in the T32 `BX` and `BXNS` instructions. You cannot use PC in other T32 instructions.

You can use SP for `Rm` in the A32 `BX` instruction but this is deprecated.

You can use SP for `Rm` in the T32 `BX` and `BXNS` instructions, but this is deprecated.

Condition flags

These instructions do not change the flags.

Availability

See the preceding table for details of availability of the `BX` and `BXNS` instructions in both instruction sets.

Related references

C1.9 Condition code suffixes on page C1-92
C2.2 Instruction width specifiers on page C2-111

C2.22 BXJ

Branch and change to Jazelle state.

Syntax

`BXJ{cond} Rm`

where:

cond
is an optional condition code. *cond* is not available on all forms of this instruction.

Rm
is a register containing an address to branch to.

Operation

The `BXJ` instruction causes a branch to the address contained in *Rm* and changes the instruction set state to Jazelle.

Note

In Armv8, `BXJ` behaves as a `BX` instruction. This means it causes a branch to an address and instruction set specified by a register.

Instruction availability and branch ranges

The following table shows the `BXJ` instructions that are available in A32 and T32 state. Instructions that are not shown in this table are not available.

Table C2-8 BXJ instruction availability and range

Instruction	A32	T32, 16-bit encoding	T32, 32-bit encoding
`BXJ Rm`	Available	-	Available
`BXJ{cond} Rm`	Available	-	-

Register restrictions

You can use SP for *Rm* in the `BXJ` A32 instruction but this is deprecated.

You cannot use SP in the `BXJ` T32 instruction.

Condition flags

The `BXJ` instruction does not change the flags.

Availability

See the preceding table for details of availability of the `BXJ` instruction in both instruction sets.

Related references

C1.9 Condition code suffixes on page C1-92

C2.23 CBZ and CBNZ

Compare and Branch on Zero, Compare and Branch on Non-Zero.

Syntax

`CBZ{q} Rn, label`

`CBNZ{q} Rn, label`

where:

q
Is an optional instruction width specifier.

Rn
Is the register holding the operand.

label
Is the branch destination.

Usage

You can use the `CBZ` or `CBNZ` instructions to avoid changing the condition flags and to reduce the number of instructions.

Except that it does not change the condition flags, `CBZ Rn, label` is equivalent to:

```
CMP     Rn, #0
BEQ     label
```

Except that it does not change the condition flags, `CBNZ Rn, label` is equivalent to:

```
CMP     Rn, #0
BNE     label
```

Restrictions

The branch destination must be a multiple of 2 in the range 0 to 126 bytes after the instruction and in the same execution state.

These instructions must not be used inside an IT block.

Condition flags

These instructions do not change the flags.

Architectures

These 16-bit instructions are available in Armv7-A T32, Armv8-A T32, and Armv8-M only.

There are no Armv7-A A32, or Armv8-A A32 or 32-bit T32 encodings of these instructions.

Related references
C2.14 B on page C2-132
C2.27 CMP and CMN on page C2-149
C2.2 Instruction width specifiers on page C2-111

C2.24 CDP and CDP2

Coprocessor data operations.

Note

CDP and CDP2 are not supported in Armv8.

Syntax

CDP{*cond*} *coproc*, #*opcode1*, *CRd*, *CRn*, *CRm*{, #*opcode2*}

CDP2{*cond*} *coproc*, #*opcode1*, *CRd*, *CRn*, *CRm*{, #*opcode2*}

where:

cond

is an optional condition code.

In A32 code, *cond* is not permitted for CDP2.

coproc

is the name of the coprocessor the instruction is for. The standard name is p*n*, where *n* is an integer in the range 0-15.

opcode1

is a 4-bit coprocessor-specific opcode.

opcode2

is an optional 3-bit coprocessor-specific opcode.

CRd, *CRn*, *CRm*

are coprocessor registers.

Usage

The use of these instructions depends on the coprocessor. See the coprocessor documentation for details.

Architectures

These 32-bit instructions are available in A32 and T32.

There are no 16-bit versions of these instructions in T32.

Related references

C1.9 Condition code suffixes on page C1-92

C2.25 CLREX

Clear Exclusive.

Syntax

`CLREX{`*cond*`}`

where:

cond

is an optional condition code.

Note

cond is permitted only in T32 code, using a preceding `IT` instruction, but this is deprecated in Armv8. This is an unconditional instruction in A32.

Usage

Use the `CLREX` instruction to clear the local record of the executing processor that an address has had a request for an exclusive access.

`CLREX` returns a closely-coupled exclusive access monitor to its open-access state. This removes the requirement for a dummy store to memory.

It is implementation defined whether `CLREX` also clears the global record of the executing processor that an address has had a request for an exclusive access.

Architectures

This 32-bit instruction is available in A32 and T32.

There is no 16-bit `CLREX` instruction in T32.

Related references

C1.9 Condition code suffixes on page C1-92

Related information

Arm Architecture Reference Manual

C2.26 CLZ

Count Leading Zeros.

Syntax

`CLZ{cond} Rd, Rm`

where:

cond
: is an optional condition code.

Rd
: is the destination register.

Rm
: is the operand register.

Operation

The `CLZ` instruction counts the number of leading zeros in the value in *Rm* and returns the result in *Rd*. The result value is 32 if no bits are set in the source register, and zero if bit 31 is set.

Register restrictions

You cannot use PC for any operand.

You can use SP in these A32 instructions but this is deprecated.

You cannot use SP in T32 instructions.

Condition flags

This instruction does not change the flags.

Architectures

This 32-bit instruction is available in A32 and T32.

There is no 16-bit version of this instruction in T32.

Examples

```
CLZ     r4,r9
CLZNE   r2,r3
```

Use the `CLZ` T32 instruction followed by a left shift of *Rm* by the resulting *Rd* value to normalize the value of register *Rm*. Use `MOVS`, rather than `MOV`, to flag the case where *Rm* is zero:

```
CLZ r5, r9
MOVS r9, r9, LSL r5
```

Related references

C1.9 Condition code suffixes on page C1-92

C2.27 CMP and CMN

Compare and Compare Negative.

Syntax

`CMP{cond} Rn, Operand2`

`CMN{cond} Rn, Operand2`

where:

cond
: is an optional condition code.

Rn
: is the register holding the first operand.

Operand2
: is a flexible second operand.

Operation

These instructions compare the value in a register with *Operand2*. They update the condition flags on the result, but do not place the result in any register.

The `CMP` instruction subtracts the value of *Operand2* from the value in *Rn*. This is the same as a `SUBS` instruction, except that the result is discarded.

The `CMN` instruction adds the value of *Operand2* to the value in *Rn*. This is the same as an `ADDS` instruction, except that the result is discarded.

In certain circumstances, the assembler can substitute `CMN` for `CMP`, or `CMP` for `CMN`. Be aware of this when reading disassembly listings.

Use of PC in A32 and T32 instructions

You cannot use PC for any operand in any data processing instruction that has a register-controlled shift.

You can use PC (`R15`) in these A32 instructions without register controlled shift but this is deprecated.

If you use PC as *Rn* in A32 instructions, the value used is the address of the instruction plus 8.

You cannot use PC for any operand in these T32 instructions.

Use of SP in A32 and T32 instructions

You can use SP for *Rn* in A32 and T32 instructions.

You can use SP for *Rm* in A32 instructions but this is deprecated.

You can use SP for *Rm* in a 16-bit T32 `CMP Rn, Rm` instruction but this is deprecated. Other uses of SP for *Rm* are not permitted in T32.

Condition flags

These instructions update the N, Z, C and V flags according to the result.

16-bit instructions

The following forms of these instructions are available in T32 code, and are 16-bit instructions:

`CMP Rn, Rm`
: Lo register restriction does not apply.

`CMN Rn, Rm`
: *Rn* and *Rm* must both be Lo registers.

CMP *Rn, #imm*

Rn must be a Lo register. *imm* range 0-255.

Correct examples

```
    CMP     r2, r9
    CMN     r0, #6400
    CMPGT   sp, r7, LSL #2
```

Incorrect example

```
    CMP     r2, pc, ASR r0 ; PC not permitted with register-controlled
                           ; shift.
```

Related references

C2.3 Flexible second operand (Operand2) on page C2-112
C1.9 Condition code suffixes on page C1-92

C2.28 CPS

Change Processor State.

Syntax

`CPS`*effect iflags*`{, #`*mode*`}`

`CPS #`*mode*

where:

effect

is one of:

IE
Interrupt or abort enable.

ID
Interrupt or abort disable.

iflags

is a sequence of one or more of:

a
Enables or disables imprecise aborts.

i
Enables or disables IRQ interrupts.

f
Enables or disables FIQ interrupts.

mode
specifies the number of the mode to change to.

Usage

Changes one or more of the mode, A, I, and F bits in the CPSR, without changing the other CPSR bits.

`CPS` is only permitted in privileged software execution, and has no effect in User mode.

`CPS` cannot be conditional, and is not permitted in an IT block.

Condition flags

This instruction does not change the condition flags.

16-bit instructions

The following forms of these instructions are available in T32 code, and are 16-bit instructions:

- `CPSIE` *iflags*.
- `CPSID` *iflags*.

You cannot specify a mode change in a 16-bit T32 instruction.

Architectures

This instruction is available in A32 and T32.

In T32, 16-bit and 32-bit versions of this instruction are available.

Examples

```
CPSIE if      ; Enable IRQ and FIQ interrupts.
CPSID A       ; Disable imprecise aborts.
CPSID ai, #17 ; Disable imprecise aborts and interrupts, and enter
```

```
                ; FIQ mode.
    CPS #16     ; Enter User mode.
```

Related concepts

A1.3 Processor modes, and privileged and unprivileged software execution on page A1-28

C2.29 CRC32

`CRC32` performs a cyclic redundancy check (CRC) calculation on a value held in a general-purpose register.

Syntax

```
CRC32B{q} Rd, Rn, Rm ; A1 Wd = CRC32(Wn, Rm[<7:0>])
CRC32H{q} Rd, Rn, Rm ; A1 Wd = CRC32(Wn, Rm[<15:0>])
CRC32W{q} Rd, Rn, Rm ; A1 Wd = CRC32(Wn, Rm[<31:0>])
CRC32B{q} Rd, Rn, Rm ; T1 Wd = CRC32(Wn, Rm[<7:0>])
CRC32H{q} Rd, Rn, Rm ; T1 Wd = CRC32(Wn, Rm[<15:0>])
CRC32W{q} Rd, Rn, Rm ; T1 Wd = CRC32(Wn, Rm[<31:0>])
```

Where:

q
Is an optional instruction width specifier. See *C2.2 Instruction width specifiers* on page C2-111. A `CRC32` instruction must be unconditional.

Rd
Is the general-purpose accumulator output register.

Rn
Is the general-purpose accumulator input register.

Rm
Is the general-purpose data source register.

Architectures supported

Supported in architecture Armv8.1 and later. Optionally supported in the Armv8-A architecture.

Usage

`CRC32` 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.

Note

ID_ISAR5.CRC32 indicates whether this instruction is supported in the T32 and A32 instruction sets.

Note

For more information about the CONSTRAINED UNPREDICTABLE behavior, see *Architectural Constraints on UNPREDICTABLE behaviors* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

C2.29 CRC32 on page C2-153

C2.1 A32 and T32 instruction summary on page C2-106

C2.30 CRC32C

`CRC32C` performs a cyclic redundancy check (CRC) calculation on a value held in a general-purpose register.

Syntax

```
CRC32CB{q} Rd, Rn, Rm ; A1 Wd = CRC32C(Wn, Rm[<7:0>])
CRC32CH{q} Rd, Rn, Rm ; A1 Wd = CRC32C(Wn, Rm[<15:0>])
CRC32CW{q} Rd, Rn, Rm ; A1 Wd = CRC32C(Wn, Rm[<31:0>])
CRC32CB{q} Rd, Rn, Rm ; T1 Wd = CRC32C(Wn, Rm[<7:0>])
CRC32CH{q} Rd, Rn, Rm ; T1 Wd = CRC32C(Wn, Rm[<15:0>])
CRC32CW{q} Rd, Rn, Rm ; T1 Wd = CRC32C(Wn, Rm[<31:0>])
```

Where:

q
Is an optional instruction width specifier. See *C2.2 Instruction width specifiers* on page C2-111. A `CRC32C` instruction must be unconditional.

Rd
Is the general-purpose accumulator output register.

Rn
Is the general-purpose accumulator input register.

Rm
Is the general-purpose data source register.

Architectures supported

Supported in architecture Armv8-A.1 and later. Optionally supported in the Armv8-A architecture.

Usage

`CRC32C` 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.

Note

ID_ISAR5.CRC32 indicates whether this instruction is supported in the T32 and A32 instruction sets.

Note

For more information about the CONSTRAINED UNPREDICTABLE behavior, see *Architectural Constraints on UNPREDICTABLE behaviors* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

C2.29 CRC32 on page C2-153

C2.1 A32 and T32 instruction summary on page C2-106

C2.31 CSDB

Consumption of Speculative Data Barrier.

Syntax

`CSDB{c}{q} ; A32`

`CSDB{c}.W ; T32`

Where:

q

Is an optional instruction width specifier. See *C2.2 Instruction width specifiers* on page C2-111.

c

Is an optional condition code. See *Chapter C1 Condition Codes* on page C1-83.

Usage

Consumption of Speculative Data Barrier is a memory barrier that controls Speculative execution and data value prediction. Arm Compiler supports the mitigation of the Variant 1 mechanism that is described in the whitepaper at *Vulnerability of Speculative Processors to Cache Timing Side-Channel Mechanism*.

The `CSDB` instruction allows Speculative execution of:

- Branch instructions.
- Instructions that write to the PC.
- Instructions that are not a result of data value predictions.
- Instructions that are the result of PSTATE.{N,Z,C,V} predictions from conditional branch instructions or from conditional instructions that write to the PC.

The `CSDB` instruction prevents Speculative execution of:

- Non-branch instructions.
- Instructions that do not write to the PC.
- Instructions that are the result of data value predictions.
- Instructions that are the result of PSTATE.{N,Z,C,V} predictions from instructions other than conditional branch instructions and conditional instructions that write to the PC.

CONSTRAINED UNPREDICTABLE behavior

For conditional CSDB instructions that specify a condition `{c}` other than `AL` in A32, and for any condition `{c}` used inside an `IT` block in T32, then how the instructions are rejected depends on your assembler implementation. See your assembler documentation for details.

Note

For more information about the CONSTRAINED UNPREDICTABLE behavior, see *Architectural Constraints on UNPREDICTABLE behaviors* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Examples

The following example shows a code sequence that could result in the processor loading data from an untrusted location that is provided by a user as the result of Speculative execution of instructions:

```
CMP R0, R1
BGE out_of_range
LDRB R4, [R5, R0]  ; load data from list A
                   ; speculative execution of this instruction
                   ; must be prevented
AND R4, R4, #1
LSL R4, R4, #8
ADD R4, R4, #0x200
CMP R4, R6
BGE out_of_range
```

```
    LDRB R7, [R8, R4]  ; load data from list B
out_of_range
```

In this example:

- There are two list objects `A` and `B`.
- `A` contains a list of values that are used to calculate offsets from which data can be loaded from `B`.
- R1 is the length of `A`.
- R5 is the base address of `A`.
- R6 is the length of `B`.
- R8 is the base address of `B`.
- R0 is an untrusted offset that is provided by a user, and is used to load an element from `A`.

When R0 is greater-than or equal-to the length of `A`, it is outside the address range of `A`. Therefore, the first branch instruction `BGE out_of_range` is taken, and instructions `LDRB R4, [R5, R0]` through `LDRB R7, [R8, R4]` are skipped.

Without a `CSDB` instruction, these skipped instructions can still be speculatively executed, and could result in:

- If R0 is maliciously set to an incorrect value, then data can be loaded into R4 from an address outside the address range of `A`.
- Data can be loaded into R7 from an address outside the address range of `B`.

To mitigate against these untrusted accesses, add a pair of `MOVGE` and `CSDB` instructions between the `BGE out_of_range` and `LDRB R4, [R5, R0]` instructions as follows:

```
    CMP R0, R1
    BGE out_of_range

    MOVGE R0, #0      ; conditonally clears the untrusted
                      ; offset provided by the user so that
                      ; it cannot affect any other code

    CSDB              ; new barrier instruction

    LDRB R4, [R5, R0] ; load data from list A
                      ; speculative execution of this instruction
                      ; is prevented
    AND R4, R4, #1
    LSL R4, R4, #8
    ADD R4, R4, #0x200
    CMP R4, R6
    BGE out_of_range
    LDRB R7, [R8, R4] ; load data from list B
out_of_range
```

Related references

C2.1 A32 and T32 instruction summary on page C2-106
C2.58 MOV on page C2-199

Related information

Arm Processor Security Update
Compiler support for mitigations

C2.32 DBG

Debug.

Syntax

`DBG{cond} {option}`

where:

cond
is an optional condition code.

option
is an optional limitation on the operation of the hint. The range is 0-15.

Usage

`DBG` is a hint instruction. It is optional whether it is implemented or not. If it is not implemented, it behaves as a `NOP`. The assembler produces a diagnostic message if the instruction executes as `NOP` on the target.

Debug hint provides a hint to a debugger and related tools. See your debugger and related tools documentation to determine the use, if any, of this instruction.

Architectures

This 32-bit instruction is available in A32 and T32.

There is no 16-bit version of this instruction in T32.

Related references
C2.68 NOP on page C2-213
C1.9 Condition code suffixes on page C1-92

C2.33 DMB

Data Memory Barrier.

Syntax

`DMB{`*cond*`} {`*option*`}`

where:

cond

is an optional condition code.

Note

cond is permitted only in T32 code. This is an unconditional instruction in A32 code.

option

is an optional limitation on the operation of the hint. Permitted values are:

SY
Full system barrier operation. This is the default and can be omitted.

LD
Barrier operation that waits only for loads to complete.

ST
Barrier operation that waits only for stores to complete.

ISH
Barrier operation only to the inner shareable domain.

ISHLD
Barrier operation that waits only for loads to complete, and only applies to the inner shareable domain.

ISHST
Barrier operation that waits only for stores to complete, and only to the inner shareable domain.

NSH
Barrier operation only out to the point of unification.

NSHLD
Barrier operation that waits only for loads to complete and only applies out to the point of unification.

NSHST
Barrier operation that waits only for stores to complete and only out to the point of unification.

OSH
Barrier operation only to the outer shareable domain.

OSHLD
`DMB` operation that waits only for loads to complete, and only applies to the outer shareable domain.

OSHST
Barrier operation that waits only for stores to complete, and only to the outer shareable domain.

Note

The options `LD`, `ISHLD`, `NSHLD`, and `OSHLD` are supported only in the Armv8-A and Armv8-R architectures.

Operation

Data Memory Barrier acts as a memory barrier. It ensures that all explicit memory accesses that appear in program order before the `DMB` instruction are observed before any explicit memory accesses that appear in program order after the `DMB` instruction. It does not affect the ordering of any other instructions executing on the processor.

Alias

The following alternative values of *`option`* are supported, but Arm recommends that you do not use them:

- `SH` is an alias for `ISH`.
- `SHST` is an alias for `ISHST`.
- `UN` is an alias for `NSH`.
- `UNST` is an alias for `NSHST`.

Architectures

This 32-bit instruction is available in A32 and T32.

There is no 16-bit version of this instruction in T32.

Related references

C1.9 Condition code suffixes on page C1-92

C2.34 DSB

Data Synchronization Barrier.

Syntax

`DSB{`*cond*`} {`*option*`}`

where:

cond

is an optional condition code.

Note

cond is permitted only in T32 code. This is an unconditional instruction in A32 code.

option

is an optional limitation on the operation of the hint. Permitted values are:

SY
Full system barrier operation. This is the default and can be omitted.

LD
Barrier operation that waits only for loads to complete.

ST
Barrier operation that waits only for stores to complete.

ISH
Barrier operation only to the inner shareable domain.

ISHLD
Barrier operation that waits only for loads to complete, and only applies to the inner shareable domain.

ISHST
Barrier operation that waits only for stores to complete, and only to the inner shareable domain.

NSH
Barrier operation only out to the point of unification.

NSHLD
Barrier operation that waits only for loads to complete and only applies out to the point of unification.

NSHST
Barrier operation that waits only for stores to complete and only out to the point of unification.

OSH
Barrier operation only to the outer shareable domain.

OSHLD
`DMB` operation that waits only for loads to complete, and only applies to the outer shareable domain.

OSHST
Barrier operation that waits only for stores to complete, and only to the outer shareable domain.

Note

The options `LD`, `ISHLD`, `NSHLD`, and `OSHLD` are supported only in the Armv8-A and Armv8-R architectures.

Operation

Data Synchronization Barrier acts as a special kind of memory barrier. No instruction in program order after this instruction executes until this instruction completes. This instruction completes when:

- All explicit memory accesses before this instruction complete.
- All Cache, Branch predictor and TLB maintenance operations before this instruction complete.

Alias

The following alternative values of *option* are supported for `DSB`, but Arm recommends that you do not use them:

- `SH` is an alias for `ISH`.
- `SHST` is an alias for `ISHST`.
- `UN` is an alias for `NSH`.
- `UNST` is an alias for `NSHST`.

Architectures

This 32-bit instruction is available in A32 and T32.

There is no 16-bit version of this instruction in T32.

Related references

C1.9 Condition code suffixes on page C1-92

C2.35 EOR

Logical Exclusive OR.

Syntax

`EOR{S}{cond} Rd, Rn, Operand2`

where:

`S`
is an optional suffix. If `S` is specified, the condition flags are updated on the result of the operation.

`cond`
is an optional condition code.

`Rd`
is the destination register.

`Rn`
is the register holding the first operand.

`Operand2`
is a flexible second operand.

Operation

The `EOR` instruction performs bitwise Exclusive OR operations on the values in *`Rn`* and *`Operand2`*.

Use of PC in T32 instructions

You cannot use PC (`R15`) for *`Rd`* or any operand in an `EOR` instruction.

Use of PC and SP in A32 instructions

You can use PC and SP with the `EOR` instruction but they are deprecated.

If you use PC as *`Rn`*, the value used is the address of the instruction plus 8.

If you use PC as *`Rd`*:
- Execution branches to the address corresponding to the result.
- If you use the `S` suffix, see the `SUBS pc,lr` instruction.

You cannot use PC for any operand in any data processing instruction that has a register-controlled shift.

Condition flags

If `S` is specified, the `EOR` instruction:
- Updates the N and Z flags according to the result.
- Can update the C flag during the calculation of *`Operand2`*.
- Does not affect the V flag.

16-bit instructions

The following forms of the `EOR` instruction are available in T32 code, and are 16-bit instructions:

`EORS` ***`Rd, Rd, Rm`***
`Rd` and *`Rm`* must both be Lo registers. This form can only be used outside an IT block.

`EOR{`*`cond`***`}`** ***`Rd, Rd, Rm`***
`Rd` and *`Rm`* must both be Lo registers. This form can only be used inside an IT block.

It does not matter if you specify `EOR{S} Rd, Rm, Rd`. The instruction is the same.

Correct examples

```
EORS    r0,r0,r3,ROR r6
EORS    r7, r11, #0x18181818
```

Incorrect example

```
EORS    r0,pc,r3,ROR r6     ; PC not permitted with register
                            ; controlled shift
```

Related references

C2.3 Flexible second operand (Operand2) on page C2-112

C2.144 SUBS pc, lr on page C2-317

C1.9 Condition code suffixes on page C1-92

C2.36 ERET

Exception Return.

Syntax

`ERET{`*cond*`}`

where:

cond

is an optional condition code.

Usage

In a processor that implements the Virtualization Extensions, you can use `ERET` to perform a return from an exception taken to Hyp mode.

Operation

When executed in Hyp mode, `ERET` loads the PC from ELR_hyp and loads the CPSR from SPSR_hyp. When executed in any other mode, apart from User or System, it behaves as:

- `MOVS PC, LR` in the A32 instruction set.
- `SUBS PC, LR, #0` in the T32 instruction set.

Notes

You must not use `ERET` in User or System mode. The assembler cannot warn you about this because it has no information about what the processor mode is likely to be at execution time.

`ERET` is the preferred synonym for `SUBS PC, LR, #0` in the T32 instruction set.

Architectures

This 32-bit instruction is available in A32 and T32.

There is no 16-bit version of this instruction in T32.

Related concepts

A1.3 Processor modes, and privileged and unprivileged software execution on page A1-28

Related references

C2.58 MOV on page C2-199

C2.144 SUBS pc, lr on page C2-317

C1.9 Condition code suffixes on page C1-92

C2.39 HVC on page C2-167

C2.37 ESB

Error Synchronization Barrier.

Syntax

`ESB{c}{q}`

`ESB{c}.W`

Where:

q
Is an optional instruction width specifier. See *C2.2 Instruction width specifiers* on page C2-111.

c
Is an optional condition code. See *Chapter C1 Condition Codes* on page C1-83.

Architectures supported

Supported in the Armv8-A and Armv8-R architectures.

Usage

Error Synchronization Barrier.

Related references
C2.1 A32 and T32 instruction summary on page C2-106

C2.38 HLT

Halting breakpoint.

Note

This instruction is supported only in the Armv8 architecture.

Syntax

`HLT{Q} #imm`

Where:

`Q`
is an optional suffix. It only has an effect when Halting debug-mode is disabled. In this case, if `Q` is specified, the instruction behaves as a `NOP`. If `Q` is not specified, the instruction is UNDEFINED.

`imm`
is an expression evaluating to an integer in the range:
- 0-65535 (a 16-bit value) in an A32 instruction.
- 0-63 (a 6-bit value) in a 16-bit T32 instruction.

Usage

The `HLT` instruction causes the processor to enter Debug state if Halting debug-mode is enabled.

In both A32 state and T32 state, imm is ignored by the Arm hardware. However, a debugger can use it to store additional information about the breakpoint.

`HLT` is an unconditional instruction. It must not have a condition code in A32 code. In T32 code, the `HLT` instruction does not require a condition code suffix because it always executes irrespective of its condition code suffix.

Availability

This instruction is available in A32 and T32.

In T32, it is only available as a 16-bit instruction.

Related references
C2.68 NOP on page C2-213

C2.39 HVC

Hypervisor Call.

Syntax

`HVC #imm`

where:

imm
is an expression evaluating to an integer in the range 0-65535.

Operation

In a processor that implements the Virtualization Extensions, the `HVC` instruction causes a Hypervisor Call exception. This means that the processor enters Hyp mode, the CPSR value is saved to the Hyp mode SPSR, and execution branches to the HVC vector.

`HVC` must not be used if the processor is in Secure state, or in User mode in Non-secure state.

imm is ignored by the processor. However, it can be retrieved by the exception handler to determine what service is being requested.

`HVC` cannot be conditional, and is not permitted in an IT block.

Notes

The `ERET` instruction performs an exception return from Hyp mode.

Architectures

This 32-bit instruction is available in A32 and T32. It is available in Armv7 architectures that include the Virtualization Extensions.

There is no 16-bit version of this instruction in T32.

Related concepts

A1.3 Processor modes, and privileged and unprivileged software execution on page A1-28

Related references

C2.36 ERET on page C2-164

C2.40 ISB

Instruction Synchronization Barrier.

Syntax

`ISB{`*`cond`*`} {`*`option`*`}`

where:

`cond`

is an optional condition code.

——— **Note** ———

`cond` is permitted only in T32 code. This is an unconditional instruction in A32 code.

———————————

`option`

is an optional limitation on the operation of the hint. The permitted value is:

`SY`

Full system barrier operation. This is the default and can be omitted.

Operation

Instruction Synchronization Barrier flushes the pipeline in the processor, so that all instructions following the `ISB` are fetched from cache or memory, after the instruction has been completed. It ensures that the effects of context altering operations, such as changing the ASID, or completed TLB maintenance operations, or branch predictor maintenance operations, in addition to all changes to the CP15 registers, executed before the `ISB` instruction are visible to the instructions fetched after the `ISB`.

In addition, the `ISB` instruction ensures that any branches that appear in program order after it are always written into the branch prediction logic with the context that is visible after the `ISB` instruction. This is required to ensure correct execution of the instruction stream.

——— **Note** ———

When the target architecture is Armv7-M, you cannot use an `ISB` instruction in an IT block, unless it is the last instruction in the block.

———————————

Architectures

This 32-bit instructions are available in A32 and T32.

There is no 16-bit version of this instruction in T32.

Related references

C1.9 Condition code suffixes on page C1-92

C2.41 IT

The `IT` (If-Then) instruction makes a single following instruction (the *IT block*) conditional. The conditional instruction must be from a restricted set of 16-bit instructions.

Syntax

`IT` *cond*

where:

cond
: specifies the condition for the following instruction.

Deprecated syntax

`IT{`*x*`{`*y*`{`*z*`}}} {`*cond*`}`

where:

x
: specifies the condition switch for the second instruction in the IT block.

y
: specifies the condition switch for the third instruction in the IT block.

z
: specifies the condition switch for the fourth instruction in the IT block.

cond
: specifies the condition for the first instruction in the IT block.

The condition switches for the second, third, and fourth instructions in the IT block can be either:

T
: Then. Applies the condition *cond* to the instruction.

E
: Else. Applies the inverse condition of *cond* to the instruction.

Usage

The *IT block* can contain between two and four conditional instructions, where the conditions can be all the same, or some of them can be the logical inverse of the others, but this is deprecated in Armv8.

The conditional instruction (including branches, but excluding the `BKPT` instruction) must specify the condition in the `{`*cond*`}` part of its syntax.

You are not required to write `IT` instructions in your code, because the assembler generates them for you automatically according to the conditions specified on the following instructions. However, if you do write `IT` instructions, the assembler validates the conditions specified in the `IT` instructions against the conditions specified in the following instructions.

Writing the `IT` instructions ensures that you consider the placing of conditional instructions, and the choice of conditions, in the design of your code.

When assembling to A32 code, the assembler performs the same checks, but does not generate any `IT` instructions.

With the exception of `CMP`, `CMN`, and `TST`, the 16-bit instructions that normally affect the condition flags, do not affect them when used inside an IT block.

A BKPT instruction in an IT block is always executed, so it does not require a condition in the {*cond*} part of its syntax. The IT block continues from the next instruction. Using a BKPT or HLT instruction inside an IT block is deprecated.

Note

You can use an IT block for unconditional instructions by using the AL condition.

Conditional branches inside an IT block have a longer branch range than those outside the IT block.

Restrictions

The following instructions are not permitted in an IT block:

- IT.
- CBZ and CBNZ.
- TBB and TBH.
- CPS, CPSID and CPSIE.
- SETEND.

Other restrictions when using an IT block are:

- A branch or any instruction that modifies the PC is only permitted in an IT block if it is the last instruction in the block.
- You cannot branch to any instruction in an IT block, unless when returning from an exception handler.
- You cannot use any assembler directives in an IT block.

Note

armasm shows a diagnostic message when any of these instructions are used in an IT block.

Using any instruction not listed in the following table in an IT block is deprecated. Also, any explicit reference to R15 (the PC) in the IT block is deprecated.

Table C2-9 Permitted instructions inside an IT block

16-bit instruction	When deprecated
MOV, MVN	When R*m* or R*d* is the PC
LDR, LDRB, LDRH, LDRSB, LDRSH	For PC-relative forms
STR, STRB, STRH	-
ADD, ADC, RSB, SBC, SUB	ADD SP, SP, #imm or SUB SP, SP, #imm or when R*m*, R*dn* or R*dm* is the PC
CMP, CMN	When R*m* or R*n* is the PC
MUL	-
ASR, LSL, LSR, ROR	-
AND, BIC, EOR, ORR, TST	-
BX, BLX	When R*m* is the PC

Condition flags

This instruction does not change the flags.

Exceptions

Exceptions can occur between an `IT` instruction and the corresponding IT block, or within an IT block. This exception results in entry to the appropriate exception handler, with suitable return information in LR and SPSR.

Instructions designed for use as exception returns can be used as normal to return from the exception, and execution of the IT block resumes correctly. This is the only way that a PC-modifying instruction can branch to an instruction in an IT block.

Availability

This 16-bit instruction is available in T32 only.

In A32 code, `IT` is a pseudo-instruction that does not generate any code.

There is no 32-bit version of this instruction.

Correct examples

```
IT     GT
LDRGT  r0, [r1,#4]

IT     EQ
ADDEQ  r0, r1, r2
```

Incorrect examples

```
IT     NE
ADD    r0,r0,r1  ; syntax error: no condition code used in IT block

ITT    EQ
MOVEQ  r0,r1
ADDEQ  r0,r0,#1  ; IT block covering more than one instruction is deprecated


IT     GT
LDRGT  r0,label  ; LDR (PC-relative) is deprecated in an IT block

IT     EQ
ADDEQ  PC,r0     ; ADD is deprecated when Rdn is the PC
```

C2.42 LDA

Load-Acquire Register.

Note

This instruction is supported only in Armv8.

Syntax

`LDA{cond} Rt, [Rn]`

`LDAB{cond} Rt, [Rn]`

`LDAH{cond} Rt, [Rn]`

where:

cond
: is an optional condition code.

Rt
: is the register to load.

Rn
: is the register on which the memory address is based.

Operation

`LDA` loads data from memory. If any loads or stores appear after a load-acquire in program order, then all observers are guaranteed to observe the load-acquire before observing the loads and stores. Loads and stores appearing before a load-acquire are unaffected.

If a store-release follows a load-acquire, each observer is guaranteed to observe them in program order.

There is no requirement that a load-acquire be paired with a store-release.

Restrictions

The address specified must be naturally aligned, or an alignment fault is generated.

The PC must not be used for `Rt` or `Rn`.

Availability

This 32-bit instruction is available in A32 and T32.

There is no 16-bit version of this instruction.

Related references

C2.43 LDAEX on page C2-173
C2.136 STL on page C2-301
C2.137 STLEX on page C2-302
C1.9 Condition code suffixes on page C1-92

C2.43 LDAEX

Load-Acquire Register Exclusive.

Note

This instruction is supported only in Armv8.

Syntax

`LDAEX{cond} Rt, [Rn]`

`LDAEXB{cond} Rt, [Rn]`

`LDAEXH{cond} Rt, [Rn]`

`LDAEXD{cond} Rt, Rt2, [Rn]`

where:

cond
: is an optional condition code.

Rt
: is the register to load.

Rt2
: is the second register for doubleword loads.

Rn
: is the register on which the memory address is based.

Operation

`LDAEX` loads data from memory.

- If the physical address has the Shared TLB attribute, `LDAEX` tags the physical address as exclusive access for the current processor, and clears any exclusive access tag for this processor for any other physical address.
- Otherwise, it tags the fact that the executing processor has an outstanding tagged physical address.
- If any loads or stores appear after `LDAEX` in program order, then all observers are guaranteed to observe the `LDAEX` before observing the loads and stores. Loads and stores appearing before `LDAEX` are unaffected.

Restrictions

The PC must not be used for any of *Rt*, *Rt2*, or *Rn*.

For A32 instructions:

- SP can be used but use of SP for any of *Rt*, or *Rt2* is deprecated.
- For `LDAEXD`, *Rt* must be an even numbered register, and not LR.
- *Rt2* must be *R(t+1)*.

For T32 instructions:

- SP can be used for *Rn*, but must not be used for any of *Rt*, or *Rt2*.
- For `LDAEXD`, *Rt* and *Rt2* must not be the same register.

Usage

Use `LDAEX` and `STLEX` to implement interprocess communication in multiple-processor and shared-memory systems.

For reasons of performance, keep the number of instructions between corresponding `LDAEX` and `STLEX` instructions to a minimum.

Note

The address used in a `STLEX` instruction must be the same as the address in the most recently executed `LDAEX` instruction.

Availability

These 32-bit instructions are available in A32 and T32.

There are no 16-bit versions of these instructions.

Related references

C2.136 STL on page C2-301
C2.42 LDA on page C2-172
C2.137 STLEX on page C2-302
C1.9 Condition code suffixes on page C1-92

C2.44 LDC and LDC2

Transfer Data from memory to Coprocessor.

Note

LDC2 is not supported in Armv8.

Syntax

```
op{L}{cond} coproc, CRd, [Rn]
op{L}{cond} coproc, CRd, [Rn, #{-}offset] ; offset addressing
op{L}{cond} coproc, CRd, [Rn, #{-}offset]! ; pre-index addressing
op{L}{cond} coproc, CRd, [Rn], #{-}offset ; post-index addressing
op{L}{cond} coproc, CRd, label
op{L}{cond} coproc, CRd, [Rn], {option}
```

where:

op
: is LDC or LDC2.

cond
: is an optional condition code.

  In A32 code, *cond* is not permitted for LDC2.

L
: is an optional suffix specifying a long transfer.

coproc
: is the name of the coprocessor the instruction is for. The standard name is p*n*, where *n* is an integer whose value must be:
  - In the range 0 to 15 in Armv7 and earlier.
  - 14 in Armv8.

CRd
: is the coprocessor register to load.

Rn
: is the register on which the memory address is based. If PC is specified, the value used is the address of the current instruction plus eight.

-
: is an optional minus sign. If - is present, the offset is subtracted from *Rn*. Otherwise, the offset is added to *Rn*.

offset
: is an expression evaluating to a multiple of 4, in the range 0 to 1020.

!
: is an optional suffix. If ! is present, the address including the offset is written back into *Rn*.

label
: is a word-aligned PC-relative expression.

  label must be within 1020 bytes of the current instruction.

option
: is a coprocessor option in the range 0-255, enclosed in braces.

Usage

The use of these instructions depends on the coprocessor. See the coprocessor documentation for details.

Architectures

These 32-bit instructions are available in A32 and T32.

There are no 16-bit versions of these instructions in T32.

Register restrictions

You cannot use PC for `Rn` in the pre-index and post-index instructions. These are the forms that write back to `Rn`.

Related references

C1.9 Condition code suffixes on page C1-92

C2.45 LDM

Load Multiple registers.

Syntax

`LDM{addr_mode}{cond} Rn{!}, reglist{^}`

where:

addr_mode

is any one of the following:

IA
Increment address After each transfer. This is the default, and can be omitted.

IB
Increment address Before each transfer (A32 only).

DA
Decrement address After each transfer (A32 only).

DB
Decrement address Before each transfer.

You can also use the stack oriented addressing mode suffixes, for example, when implementing stacks.

cond
is an optional condition code.

Rn
is the *base register*, the AArch32 register holding the initial address for the transfer. *Rn* must not be PC.

!
is an optional suffix. If ! is present, the final address is written back into *Rn*.

reglist
is a list of one or more registers to be loaded, enclosed in braces. It can contain register ranges. It must be comma separated if it contains more than one register or register range. Any combination of registers R0 to R15 (PC) can be transferred in A32 state, but there are some restrictions in T32 state.

^
is an optional suffix, available in A32 state only. You must not use it in User mode or System mode. It has the following purposes:

- If *reglist* contains the PC (R15), in addition to the normal multiple register transfer, the SPSR is copied into the CPSR. This is for returning from exception handlers. Use this only from exception modes.
- Otherwise, data is transferred into or out of the User mode registers instead of the current mode registers.

Restrictions on reglist in 32-bit T32 instructions

In 32-bit T32 instructions:

- The SP cannot be in the list.
- The PC and LR cannot both be in the list.
- There must be two or more registers in the list.

If you write an LDM instruction with only one register in reglist, the assembler automatically substitutes the equivalent LDR instruction. Be aware of this when comparing disassembly listings with source code.

Restrictions on reglist in A32 instructions

A32 load instructions can have SP and PC in the *reglist* but these instructions that include SP in the *reglist* or both PC and LR in the *reglist* are deprecated.

16-bit instructions

16-bit versions of a subset of these instructions are available in T32 code.

The following restrictions apply to the 16-bit instructions:

- All registers in *reglist* must be Lo registers.
- *Rn* must be a Lo register.
- *addr_mode* must be omitted (or IA), meaning increment address after each transfer.
- Writeback must be specified for LDM instructions where *Rn* is not in the *reglist*.

In addition, the PUSH and POP instructions are subsets of the STM and LDM instructions and can therefore be expressed using the STM and LDM instructions. Some forms of PUSH and POP are also 16-bit instructions.

Loading to the PC

A load to the PC causes a branch to the instruction at the address loaded.

Also:

- Bits[1:0] must not be 0b10.
- If bit[0] is 1, execution continues in T32 state.
- If bit[0] is 0, execution continues in A32 state.

Loading or storing the base register, with writeback

In A32 or 16-bit T32 instructions, if *Rn* is in *reglist*, and writeback is specified with the ! suffix:

- If the instruction is STM{*addr_mode*}{*cond*} and *Rn* is the lowest-numbered register in *reglist*, the initial value of *Rn* is stored. These instructions are deprecated.
- Otherwise, the loaded or stored value of Rn cannot be relied on, so these instructions are not permitted.

32-bit T32 instructions are not permitted if *Rn* is in *reglist*, and writeback is specified with the ! suffix.

Correct example

```
    LDM     r8,{r0,r2,r9}      ; LDMIA is a synonym for LDM
```

Incorrect example

```
    LDMDA   r2, {}         ; must be at least one register in list
```

Related references

C2.73 POP on page C2-221
C1.9 Condition code suffixes on page C1-92

C2.46 LDR (immediate offset)

Load with immediate offset, pre-indexed immediate offset, or post-indexed immediate offset.

Syntax

```
LDR{type}{cond} Rt, [Rn {, #offset}] ; immediate offset
LDR{type}{cond} Rt, [Rn, #offset]! ; pre-indexed
LDR{type}{cond} Rt, [Rn], #offset ; post-indexed
LDRD{cond} Rt, Rt2, [Rn {, #offset}] ; immediate offset, doubleword
LDRD{cond} Rt, Rt2, [Rn, #offset]! ; pre-indexed, doubleword
LDRD{cond} Rt, Rt2, [Rn], #offset ; post-indexed, doubleword
```

where:

type

can be any one of:

B
unsigned Byte (Zero extend to 32 bits on loads.)

SB
signed Byte (`LDR` only. Sign extend to 32 bits.)

H
unsigned Halfword (Zero extend to 32 bits on loads.)

SH
signed Halfword (`LDR` only. Sign extend to 32 bits.)

-
omitted, for Word.

cond
is an optional condition code.

Rt
is the register to load.

Rn
is the register on which the memory address is based.

offset
is an offset. If *offset* is omitted, the address is the contents of *Rn*.

Rt2
is the additional register to load for doubleword operations.

Not all options are available in every instruction set and architecture.

Offset ranges and architectures

The following table shows the ranges of offsets and availability of these instructions:

Table C2-10 Offsets and architectures, LDR, word, halfword, and byte

Instruction	Immediate offset	Pre-indexed	Post-indexed
A32, word or byte [h]	-4095 to 4095	-4095 to 4095	-4095 to 4095
A32, signed byte, halfword, or signed halfword	-255 to 255	-255 to 255	-255 to 255
A32, doubleword	-255 to 255	-255 to 255	-255 to 255
T32 32-bit encoding, word, halfword, signed halfword, byte, or signed byte [h]	-255 to 4095	-255 to 255	-255 to 255
T32 32-bit encoding, doubleword	-1020 to 1020 [i]	-1020 to 1020 [i]	-1020 to 1020 [i]

Table C2-10 Offsets and architectures, LDR, word, halfword, and byte (continued)

Instruction	Immediate offset	Pre-indexed	Post-indexed
T32 16-bit encoding, word [j]	0 to 124 [i]	Not available	Not available
T32 16-bit encoding, unsigned halfword [j]	0 to 62 [k]	Not available	Not available
T32 16-bit encoding, unsigned byte [j]	0 to 31	Not available	Not available
T32 16-bit encoding, word, `Rn` is SP [l]	0 to 1020 [i]	Not available	Not available

Register restrictions

`Rn` must be different from `Rt` in the pre-index and post-index forms.

Doubleword register restrictions

`Rn` must be different from `Rt2` in the pre-index and post-index forms.

For T32 instructions, you must not specify SP or PC for either *`Rt`* or *`Rt2`*.

For A32 instructions:

- *`Rt`* must be an even-numbered register.
- *`Rt`* must not be LR.
- Arm strongly recommends that you do not use `R12` for *`Rt`*.
- *`Rt2`* must be `R(t + 1)`.

Use of PC

In A32 code you can use PC for `Rt` in `LDR` word instructions and PC for `Rn` in `LDR` instructions.

Other uses of PC are not permitted in these A32 instructions.

In T32 code you can use PC for `Rt` in `LDR` word instructions and PC for `Rn` in `LDR` instructions. Other uses of PC in these T32 instructions are not permitted.

Use of SP

You can use SP for `Rn`.

In A32 code, you can use SP for `Rt` in word instructions. You can use SP for `Rt` in non-word instructions in A32 code but this is deprecated.

In T32 code, you can use SP for `Rt` in word instructions only. All other use of SP for `Rt` in these instructions are not permitted in T32 code.

Examples

```
LDR     r8,[r10]        ; loads R8 from the address in R10.
LDRNE   r2,[r5,#960]!   ; (conditionally) loads R2 from a word
                        ; 960 bytes above the address in R5, and
                        ; increments R5 by 960.
```

Related references

C1.9 Condition code suffixes on page C1-92

[h] For word loads, Rt can be the PC. A load to the PC causes a branch to the address loaded. In Armv4, bits[1:0] of the address loaded must be `0b00`. In Armv5T and above, bits[1:0] must not be 0b10, and if bit[0] is 1, execution continues in T32 state, otherwise execution continues in A32 state.
[i] Must be divisible by 4.
[j] Rt and Rn must be in the range R0-R7.
[k] Must be divisible by 2.
[l] Rt must be in the range R0-R7.

C2.47 LDR (PC-relative)

Load register. The address is an offset from the PC.

Syntax

`LDR{type}{cond}{.W} Rt, label`

`LDRD{cond} Rt, Rt2, label ; Doubleword`

where:

type

can be any one of:

B
unsigned Byte (Zero extend to 32 bits on loads.)

SB
signed Byte (`LDR` only. Sign extend to 32 bits.)

H
unsigned Halfword (Zero extend to 32 bits on loads.)

SH
signed Halfword (`LDR` only. Sign extend to 32 bits.)

-
omitted, for Word.

cond
is an optional condition code.

.W
is an optional instruction width specifier.

Rt
is the register to load or store.

Rt2
is the second register to load or store.

label
is a PC-relative expression.

label must be within a limited distance of the current instruction.

Note

Equivalent syntaxes are available for the `STR` instruction in A32 code but they are deprecated.

Offset range and architectures

The assembler calculates the offset from the PC for you. The assembler generates an error if *label* is out of range.

The following table shows the possible offsets between the label and the current instruction:

Table C2-11 PC-relative offsets

Instruction	Offset range
A32 `LDR`, `LDRB`, `LDRSB`, `LDRH`, `LDRSH` [m]	±4095
A32 `LDRD`	±255

[m] For word loads, Rt can be the PC. A load to the PC causes a branch to the address loaded. In Armv4, bits[1:0] of the address loaded must be `0b00`. In Armv5T and above, bits[1:0] must not be 0b10, and if bit[0] is 1, execution continues in T32 state, otherwise execution continues in A32 state.

Table C2-11 PC-relative offsets (continued)

Instruction	Offset range
32-bit T32 LDR, LDRB, LDRSB, LDRH, LDRSH [m]	±4095
32-bit T32 LDRD [n]	±1020 [o]
16-bit T32 LDR [p]	0-1020 [o]

LDR (PC-relative) in T32

You can use the `.W` width specifier to force `LDR` to generate a 32-bit instruction in T32 code. `LDR.W` always generates a 32-bit instruction, even if the target could be reached using a 16-bit `LDR`.

For forward references, `LDR` without `.W` always generates a 16-bit instruction in T32 code, even if that results in failure for a target that could be reached using a 32-bit T32 `LDR` instruction.

Doubleword register restrictions

For 32-bit T32 instructions, you must not specify SP or PC for either *`Rt`* or *`Rt2`*.

For A32 instructions:

- *`Rt`* must be an even-numbered register.
- *`Rt`* must not be LR.
- Arm strongly recommends that you do not use `R12` for *`Rt`*.
- *`Rt2`* must be `R(t + 1)`.

Use of SP

In A32 code, you can use SP for `Rt` in `LDR` word instructions. You can use SP for `Rt` in `LDR` non-word A32 instructions but this is deprecated.

In T32 code, you can use SP for `Rt` in `LDR` word instructions only. All other uses of SP in these instructions are not permitted in T32 code.

Related references

C1.9 Condition code suffixes on page C1-92

[m] For word loads, Rt can be the PC. A load to the PC causes a branch to the address loaded. In Armv4, bits[1:0] of the address loaded must be `0b00`. In Armv5T and above, bits[1:0] must not be 0b10, and if bit[0] is 1, execution continues in T32 state, otherwise execution continues in A32 state.

[n] In Armv7-M, `LDRD` (PC-relative) instructions must be on a word-aligned address.

[o] Must be a multiple of 4.

[p] Rt must be in the range R0-R7. There are no byte, halfword, or doubleword 16-bit instructions.

C2.48 LDR (register offset)

Load with register offset, pre-indexed register offset, or post-indexed register offset.

Syntax

```
LDR{type}{cond} Rt, [Rn, ±Rm {, shift}] ; register offset
LDR{type}{cond} Rt, [Rn, ±Rm {, shift}]! ; pre-indexed ; A32 only
LDR{type}{cond} Rt, [Rn], ±Rm {, shift} ; post-indexed ; A32 only
LDRD{cond} Rt, Rt2, [Rn, ±Rm] ; register offset, doubleword ; A32 only
LDRD{cond} Rt, Rt2, [Rn, ±Rm]! ; pre-indexed, doubleword ; A32 only
LDRD{cond} Rt, Rt2, [Rn], ±Rm ; post-indexed, doubleword ; A32 only
```

where:

type

can be any one of:

B
: unsigned Byte (Zero extend to 32 bits on loads.)

SB
: signed Byte (`LDR` only. Sign extend to 32 bits.)

H
: unsigned Halfword (Zero extend to 32 bits on loads.)

SH
: signed Halfword (`LDR` only. Sign extend to 32 bits.)

-
: omitted, for Word.

cond
: is an optional condition code.

Rt
: is the register to load.

Rn
: is the register on which the memory address is based.

Rm
: is a register containing a value to be used as the offset. *-Rm* is not permitted in T32 code.

shift
: is an optional shift.

Rt2
: is the additional register to load for doubleword operations.

Not all options are available in every instruction set and architecture.

Offset register and shift options

The following table shows the ranges of offsets and availability of these instructions:

Table C2-12 Options and architectures, LDR (register offsets)

Instruction	±*Rm* [q]	shift		
A32, word or byte [r]	±*Rm*	`LSL #0-31`	`LSR #1-32`	
		`ASR #1-32`	`ROR #1-31`	`RRX`
A32, signed byte, halfword, or signed halfword	±*Rm*	Not available		
A32, doubleword	±*Rm*	Not available		
T32 32-bit encoding, word, halfword, signed halfword, byte, or signed byte [r]	+*Rm*	`LSL #0-3`		
T32 16-bit encoding, all except doubleword [s]	+*Rm*	Not available		

Register restrictions

In the pre-index and post-index forms, `Rn` must be different from `Rt`.

Doubleword register restrictions

For A32 instructions:

- *Rt* must be an even-numbered register.
- *Rt* must not be LR.
- Arm strongly recommends that you do not use `R12` for *Rt*.
- *Rt2* must be `R(t + 1)`.
- *Rm* must be different from `Rt` and *Rt2* in `LDRD` instructions.
- `Rn` must be different from *Rt2* in the pre-index and post-index forms.

Use of PC

In A32 instructions you can use PC for `Rt` in `LDR` word instructions, and you can use PC for `Rn` in `LDR` instructions with register offset syntax (that is the forms that do not writeback to the `Rn`).

Other uses of PC are not permitted in A32 instructions.

In T32 instructions you can use PC for `Rt` in `LDR` word instructions. Other uses of PC in these T32 instructions are not permitted.

Use of SP

You can use SP for `Rn`.

In A32 code, you can use SP for `Rt` in word instructions. You can use SP for `Rt` in non-word A32 instructions but this is deprecated.

You can use SP for `Rm` in A32 instructions but this is deprecated.

In T32 code, you can use SP for `Rt` in word instructions only. All other use of SP for `Rt` in these instructions are not permitted in T32 code.

Use of SP for `Rm` is not permitted in T32 state.

Related references

C1.9 Condition code suffixes on page C1-92

[q] Where ±Rm is shown, you can use –Rm, +Rm, or Rm. Where +Rm is shown, you cannot use –Rm.
[r] For word loads, Rt can be the PC. A load to the PC causes a branch to the address loaded. In Armv4, bits[1:0] of the address loaded must be `0b00`. In Armv5T and above, bits[1:0] must not be 0b10, and if bit[0] is 1, execution continues in T32 state, otherwise execution continues in A32 state.
[s] Rt, Rn, and Rm must all be in the range R0-R7.

C2.49 LDR (register-relative)

Load register. The address is an offset from a base register.

Syntax

`LDR{type}{cond}{.W} Rt, label`

`LDRD{cond} Rt, Rt2, label ; Doubleword`

where:

type

can be any one of:

B
unsigned Byte (Zero extend to 32 bits on loads.)

SB
signed Byte (`LDR` only. Sign extend to 32 bits.)

H
unsigned Halfword (Zero extend to 32 bits on loads.)

SH
signed Halfword (`LDR` only. Sign extend to 32 bits.)

-
omitted, for Word.

cond
is an optional condition code.

.W
is an optional instruction width specifier.

Rt
is the register to load or store.

Rt2
is the second register to load or store.

label
is a symbol defined by the `FIELD` directive. *label* specifies an offset from the base register which is defined using the `MAP` directive.

label must be within a limited distance of the value in the base register.

Offset range and architectures

The assembler calculates the offset from the base register for you. The assembler generates an error if *label* is out of range.

The following table shows the possible offsets between the label and the current instruction:

Table C2-13 Register-relative offsets

Instruction	Offset range
A32 `LDR`, `LDRB` [t]	±4095
A32 `LDRSB`, `LDRH`, `LDRSH`	±255
A32 `LDRD`	±255
T32, 32-bit `LDR`, `LDRB`, `LDRSB`, `LDRH`, `LDRSH` [t]	-255 to 4095

[t] For word loads, Rt can be the PC. A load to the PC causes a branch to the address loaded. In Armv4, bits[1:0] of the address loaded must be `0b00`. In Armv5T and above, bits[1:0] must not be 0b10, and if bit[0] is 1, execution continues in T32 state, otherwise execution continues in A32 state.
[u] Must be a multiple of 4.

Table C2-13 Register-relative offsets (continued)

Instruction	Offset range
T32, 32-bit LDRD	±1020 [u]
T32, 16-bit LDR [v]	0 to 124 [u]
T32, 16-bit LDRH [v]	0 to 62 [w]
T32, 16-bit LDRB [v]	0 to 31
T32, 16-bit LDR, base register is SP [x]	0 to 1020 [u]

LDR (register-relative) in T32

You can use the `.W` width specifier to force `LDR` to generate a 32-bit instruction in T32 code. `LDR.W` always generates a 32-bit instruction, even if the target could be reached using a 16-bit `LDR`.

For forward references, `LDR` without `.W` always generates a 16-bit instruction in T32 code, even if that results in failure for a target that could be reached using a 32-bit T32 `LDR` instruction.

Doubleword register restrictions

For 32-bit T32 instructions, you must not specify SP or PC for either *`Rt`* or *`Rt2`*.

For A32 instructions:

- *`Rt`* must be an even-numbered register.
- *`Rt`* must not be LR.
- Arm strongly recommends that you do not use `R12` for *`Rt`*.
- *`Rt2`* must be `R(t + 1)`.

Use of PC

You can use PC for `Rt` in word instructions. Other uses of PC are not permitted in these instructions.

Use of SP

In A32 code, you can use SP for `Rt` in word instructions. You can use SP for `Rt` in non-word A32 instructions but this is deprecated.

In T32 code, you can use SP for `Rt` in word instructions only. All other use of SP for `Rt` in these instructions are not permitted in T32 code.

Related references

C1.9 Condition code suffixes on page C1-92

[v] Rt and base register must be in the range R0-R7.
[w] Must be a multiple of 2.
[x] Rt must be in the range R0-R7.

C2.50 LDR, unprivileged

Unprivileged load byte, halfword, or word.

Syntax

```
LDR{type}T{cond} Rt, [Rn {, #offset}] ; immediate offset (32-bit T32 encoding only)
LDR{type}T{cond} Rt, [Rn] {, #offset} ; post-indexed (A32 only)
LDR{type}T{cond} Rt, [Rn], ±Rm {, shift} ; post-indexed (register) (A32 only)
```

where:

type

can be any one of:

B
: unsigned Byte (Zero extend to 32 bits on loads.)

SB
: signed Byte (Sign extend to 32 bits.)

H
: unsigned Halfword (Zero extend to 32 bits on loads.)

SH
: signed Halfword (Sign extend to 32 bits.)

-
: omitted, for Word.

cond
: is an optional condition code.

Rt
: is the register to load.

Rn
: is the register on which the memory address is based.

offset
: is an offset. If offset is omitted, the address is the value in *Rn*.

Rm
: is a register containing a value to be used as the offset. *Rm* must not be PC.

shift
: is an optional shift.

Operation

When these instructions are executed by privileged software, they access memory with the same restrictions as they would have if they were executed by unprivileged software.

When executed by unprivileged software these instructions behave in exactly the same way as the corresponding load instruction, for example LDRSBT behaves in the same way as LDRSB.

Offset ranges and architectures

The following table shows the ranges of offsets and availability of these instructions.

Table C2-14 Offsets and architectures, LDR (User mode)

Instruction	Immediate offset	Post-indexed	±*Rm* [y]	shift
A32, word or byte	Not available	-4095 to 4095	±*Rm*	LSL #0-31
				LSR #1-32

[y] You can use –Rm, +Rm, or Rm.

Table C2-14 Offsets and architectures, LDR (User mode) (continued)

Instruction	Immediate offset	Post-indexed	±*Rm* [y]	shift
				`ASR #1-32`
				`ROR #1-31`
				`RRX`
A32, signed byte, halfword, or signed halfword	Not available	-255 to 255	±*Rm*	Not available
T32, 32-bit encoding, word, halfword, signed halfword, byte, or signed byte	0 to 255	Not available	Not available	

Related references

C1.9 Condition code suffixes on page C1-92

C2.51 LDREX

Load Register Exclusive.

Syntax

`LDREX{cond} Rt, [Rn {, #offset}]`

`LDREXB{cond} Rt, [Rn]`

`LDREXH{cond} Rt, [Rn]`

`LDREXD{cond} Rt, Rt2, [Rn]`

where:

cond
: is an optional condition code.

Rt
: is the register to load.

Rt2
: is the second register for doubleword loads.

Rn
: is the register on which the memory address is based.

offset
: is an optional offset applied to the value in *Rn*. *offset* is permitted only in 32-bit T32 instructions. If *offset* is omitted, an offset of zero is assumed.

Operation

LDREX loads data from memory.

- If the physical address has the Shared TLB attribute, LDREX tags the physical address as exclusive access for the current processor, and clears any exclusive access tag for this processor for any other physical address.
- Otherwise, it tags the fact that the executing processor has an outstanding tagged physical address.

LDREXB and LDREXH zero extend the value loaded.

Restrictions

PC must not be used for any of *Rt*, *Rt2*, or *Rn*.

For A32 instructions:

- SP can be used but use of SP for any of *Rt*, or *Rt2* is deprecated.
- For LDREXD, *Rt* must be an even numbered register, and not LR.
- *Rt2* must be *R(t+1)*.
- *offset* is not permitted.

For T32 instructions:

- SP can be used for R*n*, but must not be used for *Rt* or *Rt2*.
- For LDREXD, *Rt* and *Rt2* must not be the same register.
- The value of *offset* can be any multiple of four in the range 0-1020.

Usage

Use LDREX and STREX to implement interprocess communication in multiple-processor and shared-memory systems.

For reasons of performance, keep the number of instructions between corresponding `LDREX` and `STREX` instructions to a minimum.

Note

The address used in a `STREX` instruction must be the same as the address in the most recently executed `LDREX` instruction.

Architectures

These 32-bit instructions are available in A32 and T32.

The `LDREXD` instruction is not available in the Armv7-M architecture.

There are no 16-bit versions of these instructions in T32.

Examples

```
    MOV r1, #0x1                ; load the 'lock taken' value
try
    LDREX r0, [LockAddr]        ; load the lock value
    CMP r0, #0                  ; is the lock free?
    STREXEQ r0, r1, [LockAddr]  ; try and claim the lock
    CMPEQ r0, #0                ; did this succeed?
    BNE try                     ; no - try again
    ....                        ; yes - we have the lock
```

Related references

C1.9 Condition code suffixes on page C1-92

C2.52 LSL

Logical Shift Left. This instruction is a preferred synonym for `MOV` instructions with shifted register operands.

Syntax

`LSL{S}{`*`cond`*`} Rd, Rm, Rs`

`LSL{S}{`*`cond`*`} Rd, Rm, #sh`

where:

`S`
: is an optional suffix. If `S` is specified, the condition flags are updated on the result of the operation.

`Rd`
: is the destination register.

`Rm`
: is the register holding the first operand. This operand is shifted left.

`Rs`
: is a register holding a shift value to apply to the value in *`Rm`*. Only the least significant byte is used.

`sh`
: is a constant shift. The range of values permitted is 0-31.

Operation

`LSL` provides the value of a register multiplied by a power of two, inserting zeros into the vacated bit positions.

Restrictions in T32 code

T32 instructions must not use PC or SP.

You cannot specify zero for the *`sh`* value in an `LSL` instruction in an IT block.

Use of SP and PC in A32 instructions

You can use SP in these A32 instructions but this is deprecated.

You cannot use PC in instructions with the `LSL{S}{cond} Rd, Rm, Rs` syntax. You can use PC for `R`*`d`* and `R`*`m`* in the other syntax, but this is deprecated.

If you use PC as *`Rm`*, the value used is the address of the instruction plus 8.

If you use PC as *`Rd`*:

- Execution branches to the address corresponding to the result.
- If you use the `S` suffix, the SPSR of the current mode is copied to the CPSR. You can use this to return from exceptions.

  Note

  The A32 instruction `LSLS{`*`cond`*`} pc,Rm,#sh` always disassembles to the preferred form `MOVS{`*`cond`*`} pc,Rm{,shift}`.

Caution

Do not use the `S` suffix when using PC as *`Rd`* in User mode or System mode. The assembler cannot warn you about this because it has no information about what the processor mode is likely to be at execution time.

You cannot use PC for *Rd* or any operand in the LSL instruction if it has a register-controlled shift.

Condition flags

If S is specified, the LSL instruction updates the N and Z flags according to the result.

The C flag is unaffected if the shift value is 0. Otherwise, the C flag is updated to the last bit shifted out.

16-bit instructions

The following forms of these instructions are available in T32 code, and are 16-bit instructions:

LSLS *Rd, Rm, #sh*
Rd and *Rm* must both be Lo registers. This form can only be used outside an IT block.

LSL{*cond*} *Rd, Rm, #sh*
Rd and *Rm* must both be Lo registers. This form can only be used inside an IT block.

LSLS *Rd, Rd, Rs*
Rd and *Rs* must both be Lo registers. This form can only be used outside an IT block.

LSL{*cond*} *Rd, Rd, Rs*
Rd and *Rs* must both be Lo registers. This form can only be used inside an IT block.

Architectures

This 32-bit instruction is available in A32 and T32.

This 16-bit T32 instruction is available in T32.

Example

```
    LSLS    r1, r2, r3
```

Related references
C2.58 MOV on page C2-199
C1.9 Condition code suffixes on page C1-92

C2.53 LSR

Logical Shift Right. This instruction is a preferred synonym for `MOV` instructions with shifted register operands.

Syntax

`LSR{S}{`*cond*`} Rd, Rm, Rs`

`LSR{S}{`*cond*`} Rd, Rm, #sh`

where:

`S`
: is an optional suffix. If `S` is specified, the condition flags are updated on the result of the operation.

`Rd`
: is the destination register.

`Rm`
: is the register holding the first operand. This operand is shifted right.

`Rs`
: is a register holding a shift value to apply to the value in *`Rm`*. Only the least significant byte is used.

`sh`
: is a constant shift. The range of values permitted is 1-32.

Operation

`LSR` provides the unsigned value of a register divided by a variable power of two, inserting zeros into the vacated bit positions.

Restrictions in T32 code

T32 instructions must not use PC or SP.

Use of SP and PC in A32 instructions

You can use SP in these A32 instructions but they are deprecated.

You cannot use PC in instructions with the `LSR{S}{cond} Rd, Rm, Rs` syntax. You can use PC for `Rd` and `Rm` in the other syntax, but this is deprecated.

If you use PC as *`Rm`*, the value used is the address of the instruction plus 8.

If you use PC as *`Rd`*:

- Execution branches to the address corresponding to the result.
- If you use the `S` suffix, the SPSR of the current mode is copied to the CPSR. You can use this to return from exceptions.

  Note

  The A32 instruction `LSRS{`*cond*`} pc,`*`Rm`*`,#`*`sh`* always disassembles to the preferred form `MOVS{`*cond*`} pc,`*`Rm`*`{,shift}`.

Caution

Do not use the `S` suffix when using PC as *`Rd`* in User mode or System mode. The assembler cannot warn you about this because it has no information about what the processor mode is likely to be at execution time.

You cannot use PC for *`Rd`* or any operand in the `LSR` instruction if it has a register-controlled shift.

Condition flags

If `S` is specified, the instruction updates the N and Z flags according to the result.

The C flag is unaffected if the shift value is 0. Otherwise, the C flag is updated to the last bit shifted out.

16-bit instructions

The following forms of these instructions are available in T32 code, and are 16-bit instructions:

`LSRS Rd, Rm, #sh`
: `Rd` and `Rm` must both be Lo registers. This form can only be used outside an IT block.

`LSR{cond} Rd, Rm, #sh`
: `Rd` and `Rm` must both be Lo registers. This form can only be used inside an IT block.

`LSRS Rd, Rd, Rs`
: `Rd` and `Rs` must both be Lo registers. This form can only be used outside an IT block.

`LSR{cond} Rd, Rd, Rs`
: `Rd` and `Rs` must both be Lo registers. This form can only be used inside an IT block.

Architectures

This 32-bit instruction is available in A32 and T32.

This 16-bit T32 instruction is available in T32.

Example

```
    LSR     r4, r5, r6
```

Related references

C2.58 MOV on page C2-199
C1.9 Condition code suffixes on page C1-92

C2.54 MCR and MCR2

Move to Coprocessor from general-purpose register. Depending on the coprocessor, you might be able to specify various additional operations.

Note

`MCR2` is not supported in Armv8.

Syntax

`MCR{cond} coproc, #opcode1, Rt, CRn, CRm{, #opcode2}`

`MCR2{cond} coproc, #opcode1, Rt, CRn, CRm{, #opcode2}`

where:

cond

is an optional condition code.

In A32 code, *cond* is not permitted for `MCR2`.

coproc

is the name of the coprocessor the instruction is for. The standard name is p*n*, where *n* is an integer whose value must be:

- In the range 0-15 in Armv7 and earlier.
- 14 or 15 in Armv8.

opcode1

is a 3-bit coprocessor-specific opcode.

opcode2

is an optional 3-bit coprocessor-specific opcode.

Rt

is a general-purpose register. `Rt` must not be PC.

CRn, CRm

are coprocessor registers.

Usage

The use of these instructions depends on the coprocessor. See the coprocessor documentation for details.

Architectures

These 32-bit instructions are available in A32 and T32.

There are no 16-bit versions of these instructions in T32.

Related references

C1.9 Condition code suffixes on page C1-92

C2.55 MCRR and MCRR2

Move to Coprocessor from two general-purpose registers. Depending on the coprocessor, you might be able to specify various additional operations.

Note

MCRR2 is not supported in Armv8.

Syntax

```
MCRR{cond} coproc, #opcode, Rt, Rt2, CRn
MCRR2{cond} coproc, #opcode, Rt, Rt2, CRn
```

where:

cond
: is an optional condition code.

  In A32 code, *cond* is not permitted for MCRR2.

coproc
: is the name of the coprocessor the instruction is for. The standard name is p*n*, where *n* is an integer whose value must be:
  - In the range 0-15 in Armv7 and earlier.
  - 14 or 15 in Armv8.

opcode
: is a 4-bit coprocessor-specific opcode.

Rt, Rt2
: are general-purpose registers. Rt and Rt2 must not be PC.

CRn
: is a coprocessor register.

Usage

The use of these instructions depends on the coprocessor. See the coprocessor documentation for details.

Architectures

These 32-bit instructions are available in A32 and T32.

There are no 16-bit versions of these instructions in T32.

Related references

C1.9 Condition code suffixes on page C1-92

C2.56 MLA

Multiply-Accumulate with signed or unsigned 32-bit operands, giving the least significant 32 bits of the result.

Syntax

`MLA{S}{cond} Rd, Rn, Rm, Ra`

where:

cond
: is an optional condition code.

S
: is an optional suffix. If `S` is specified, the condition flags are updated on the result of the operation.

Rd
: is the destination register.

Rn, Rm
: are registers holding the values to be multiplied.

Ra
: is a register holding the value to be added.

Operation

The `MLA` instruction multiplies the values from *Rn* and *Rm*, adds the value from *Ra*, and places the least significant 32 bits of the result in *Rd*.

Register restrictions

You cannot use PC for any register.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

Condition flags

If `S` is specified, the `MLA` instruction:

- Updates the N and Z flags according to the result.
- Does not affect the C or V flag.

Architectures

This instruction is available in A32 and T32.

There is no 16-bit version of this instruction in T32.

Example

```
    MLA     r10, r2, r1, r5
```

Related references

C1.9 Condition code suffixes on page C1-92

C2.57 MLS

Multiply-Subtract, with signed or unsigned 32-bit operands, giving the least significant 32 bits of the result.

Syntax

`MLS{cond} Rd, Rn, Rm, Ra`

where:

cond
: is an optional condition code.

S
: is an optional suffix. If `S` is specified, the condition flags are updated on the result of the operation.

Rd
: is the destination register.

Rn, Rm
: are registers holding the values to be multiplied.

Ra
: is a register holding the value to be subtracted from.

Operation

The `MLS` instruction multiplies the values in *Rn* and *Rm*, subtracts the result from the value in *Ra*, and places the least significant 32 bits of the final result in *Rd*.

Register restrictions

You cannot use PC for any register.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

Architectures

This instruction is available in A32 and T32.

There is no 16-bit version of this instruction in T32.

Example

```
    MLS     r4, r5, r6, r7
```

Related references

C1.9 Condition code suffixes on page C1-92

C2.58 MOV

Move.

Syntax

`MOV{S}{`*cond*`} Rd, Operand2`

`MOV{`*cond*`} Rd, #imm16`

where:

`S`
: is an optional suffix. If `S` is specified, the condition flags are updated on the result of the operation.

cond
: is an optional condition code.

Rd
: is the destination register.

Operand2
: is a flexible second operand.

imm16
: is any value in the range 0-65535.

Operation

The `MOV` instruction copies the value of *Operand2* into *Rd*.

In certain circumstances, the assembler can substitute `MVN` for `MOV`, or `MOV` for `MVN`. Be aware of this when reading disassembly listings.

Use of PC and SP in 32-bit T32 encodings

You cannot use PC (`R15`) for *Rd*, or in *Operand2*, in 32-bit T32 `MOV` instructions. With the following exceptions, you cannot use SP (`R13`) for *Rd*, or in *Operand2*:

- `MOV{`*cond*`}.W Rd, SP`, where *Rd* is not SP.
- `MOV{`*cond*`}.W SP, Rm`, where *Rm* is not SP.

Use of PC and SP in 16-bit T32 encodings

You can use PC or SP in 16-bit T32 `MOV{`*cond*`} Rd, Rm` instructions but these instructions in which both *Rd* and *Rm* are SP or PC are deprecated.

You cannot use PC or SP in any other `MOV{S}` 16-bit T32 instructions.

Use of PC and SP in A32 MOV

You cannot use PC for *Rd* or any operand in any data processing instruction that has a register-controlled shift.

In instructions without register-controlled shift, the use of PC is deprecated except for the following cases:

- `MOVS PC, LR`.
- `MOV PC, R`*m* when R*m* is not PC or SP.
- `MOV Rd, PC` when R*d* is not PC or SP.

You can use SP for R*d* or R*m*. But this is deprecated except for the following cases:

- `MOV SP, R`*m* when R*m* is not PC or SP.
- `MOV Rd, SP` when R*d* is not PC or SP.

Note

- You cannot use PC for `Rd` in `MOV Rd, #imm16` if the `#imm16` value is not a permitted `Operand2` value. You can use PC in forms with `Operand2` without register-controlled shift.

If you use PC as `Rm`, the value used is the address of the instruction plus 8.

If you use PC as `Rd`:
- Execution branches to the address corresponding to the result.
- If you use the `S` suffix, see the `SUBS pc,lr` instruction.

Condition flags

If `S` is specified, the instruction:
- Updates the N and Z flags according to the result.
- Can update the C flag during the calculation of `Operand2`.
- Does not affect the V flag.

16-bit instructions

The following forms of this instruction are available in T32 code, and are 16-bit instructions:

`MOVS Rd, #imm`
`Rd` must be a Lo register. `imm` range 0-255. This form can only be used outside an IT block.

`MOV{cond} Rd, #imm`
`Rd` must be a Lo register. `imm` range 0-255. This form can only be used inside an IT block.

`MOVS Rd, Rm`
`Rd` and `Rm` must both be Lo registers. This form can only be used outside an IT block.

`MOV{cond} Rd, Rm`
`Rd` or `Rm` can be Lo or Hi registers.

Availability

These instructions are available in A32 and T32.

In T32, 16-bit and 32-bit versions of these instructions are available.

Related references

C2.3 Flexible second operand (Operand2) on page C2-112
C2.144 SUBS pc, lr on page C2-317
C1.9 Condition code suffixes on page C1-92

C2.59 MOVT

Move Top.

Syntax

`MOVT{cond} Rd, #imm16`

where:

cond
: is an optional condition code.

Rd
: is the destination register.

imm16
: is a 16-bit immediate value.

Usage

`MOVT` writes *imm16* to *Rd*[31:16], without affecting *Rd*[15:0].

You can generate any 32-bit immediate with a `MOV`, `MOVT` instruction pair.

Register restrictions

You cannot use PC in A32 or T32 instructions.

You can use SP for *Rd* in A32 instructions but this is deprecated.

You cannot use SP in T32 instructions.

Condition flags

This instruction does not change the flags.

Architectures

This 32-bit instruction is available in A32 and T32.

There is no 16-bit version of this instruction in T32.

Related references

C1.9 Condition code suffixes on page C1-92

C2.60 MRC and MRC2

Move to general-purpose register from Coprocessor. Depending on the coprocessor, you might be able to specify various additional operations.

Note

`MRC2` is not supported in Armv8.

Syntax

`MRC{cond} coproc, #opcode1, Rt, CRn, CRm{, #opcode2}`

`MRC2{cond} coproc, #opcode1, Rt, CRn, CRm{, #opcode2}`

where:

cond

is an optional condition code.

In A32 code, *cond* is not permitted for `MRC2`.

coproc

is the name of the coprocessor the instruction is for. The standard name is p*n*, where *n* is an integer whose value must be:

- In the range 0-15 in Armv7 and earlier.
- 14 or 15 in Armv8.

opcode1

is a 3-bit coprocessor-specific opcode.

opcode2

is an optional 3-bit coprocessor-specific opcode.

Rt

is the general-purpose register. `Rt` must not be PC.

Rt can be `APSR_nzcv`. This means that the coprocessor executes an instruction that changes the value of the condition flags in the APSR.

CRn, CRm

are coprocessor registers.

Usage

The use of these instructions depends on the coprocessor. See the coprocessor documentation for details.

Architectures

These 32-bit instructions are available in A32 and T32.

There are no 16-bit versions of these instructions in T32.

Related references

C1.9 Condition code suffixes on page C1-92

C2.61 MRRC and MRRC2

Move to two general-purpose registers from coprocessor. Depending on the coprocessor, you might be able to specify various additional operations.

Note

MRRC2 is not supported in Armv8.

Syntax

```
MRRC{cond} coproc, #opcode, Rt, Rt2, CRm
MRRC2{cond} coproc, #opcode, Rt, Rt2, CRm
```

where:

cond
: is an optional condition code.

  In A32 code, *cond* is not permitted for MRRC2.

coproc
: is the name of the coprocessor the instruction is for. The standard name is p*n*, where *n* is an integer whose value must be:
  - In the range 0-15 in Armv7 and earlier.
  - 14 or 15 in Armv8.

opcode
: is a 4-bit coprocessor-specific opcode.

Rt, Rt2
: are general-purpose registers. Rt and Rt2 must not be PC.

CRm
: is a coprocessor register.

Usage

The use of these instructions depends on the coprocessor. See the coprocessor documentation for details.

Architectures

These 32-bit instructions are available in A32 and T32.

There are no 16-bit versions of these instructions in T32.

Related references

C1.9 Condition code suffixes on page C1-92

C2.62 MRS (PSR to general-purpose register)

Move the contents of a PSR to a general-purpose register.

Syntax

`MRS{cond} Rd, psr`

where:

cond
is an optional condition code.

Rd
is the destination register.

psr
is one of:

APSR
on any processor, in any mode.

CPSR
deprecated synonym for APSR and for use in Debug state, on any processor except Armv7-M and Armv6-M.

SPSR
on any processor, except Armv6-M, Armv7-M, Armv8-M Baseline, and Armv8-M Mainline, in privileged software execution only.

Mpsr
on Armv6-M, Armv7-M, Armv8-M Baseline, and Armv8-M Mainline processors only.

Mpsr
can be any of: `IPSR`, `EPSR`, `IEPSR`, `IAPSR`, `EAPSR`, `MSP`, `PSP`, `XPSR`, `PRIMASK`, `BASEPRI`, `BASEPRI_MAX`, `FAULTMASK`, or `CONTROL`.

Usage

Use `MRS` in combination with `MSR` as part of a read-modify-write sequence for updating a PSR, for example to change processor mode, or to clear the Q flag.

In process swap code, the programmers' model state of the process being swapped out must be saved, including relevant PSR contents. Similarly, the state of the process being swapped in must also be restored. These operations make use of `MRS`/store and load/`MSR` instruction sequences.

SPSR

You must not attempt to access the SPSR when the processor is in User or System mode. This is your responsibility. The assembler cannot warn you about this, because it has no information about the processor mode at execution time.

CPSR

Arm deprecates reading the CPSR endianness bit (E) with an `MRS` instruction.

The CPSR execution state bits, other than the E bit, can only be read when the processor is in Debug state, halting debug-mode. Otherwise, the execution state bits in the CPSR read as zero.

The condition flags can be read in any mode on any processor. Use APSR if you are only interested in accessing the condition flags in User mode.

Register restrictions

You cannot use PC for `Rd` in A32 instructions. You can use SP for `Rd` in A32 instructions but this is deprecated.

You cannot use PC or SP for `Rd` in T32 instructions.

Condition flags

This instruction does not change the flags.

Architectures

This instruction is available in A32 and T32.

There is no 16-bit version of this instruction in T32.

Related concepts

A1.13 Current Program Status Register in AArch32 state on page A1-39

Related references

C2.63 MRS (system coprocessor register to general-purpose register) on page C2-206

C2.64 MSR (general-purpose register to system coprocessor register) on page C2-207

C2.65 MSR (general-purpose register to PSR) on page C2-208

C1.9 Condition code suffixes on page C1-92

C2.63 MRS (system coprocessor register to general-purpose register)

Move to general-purpose register from system coprocessor register.

Syntax

`MRS{cond} Rn, coproc_register`

`MRS{cond} APSR_nzcv, special_register`

where:

cond
: is an optional condition code.

coproc_register
: is the name of the coprocessor register.

special_register
: is the name of the coprocessor register that can be written to APSR_nzcv. This is only possible for the coprocessor register DBGDSCRint.

Rn
: is the general-purpose register. `Rn` must not be PC.

Usage

You can use this pseudo-instruction to read CP14 or CP15 coprocessor registers, with the exception of write-only registers. A complete list of the applicable coprocessor register names is in the *Arm®v7-AR Architecture Reference Manual*. For example:

```
    MRS R1, SCTLR ; writes the contents of the CP15 coprocessor
                  ; register SCTLR into R1
```

Architectures

This pseudo-instruction is available in Armv7-A and Armv7-R in A32 and 32-bit T32 code.

There is no 16-bit version of this pseudo-instruction in T32.

Related references

C2.62 MRS (PSR to general-purpose register) on page C2-204

C2.64 MSR (general-purpose register to system coprocessor register) on page C2-207

C2.65 MSR (general-purpose register to PSR) on page C2-208

C1.9 Condition code suffixes on page C1-92

Related information

Arm Architecture Reference Manual

C2.64 MSR (general-purpose register to system coprocessor register)

Move to system coprocessor register from general-purpose register.

Syntax

`MSR{cond} coproc_register, Rn`

where:

cond
: is an optional condition code.

coproc_register
: is the name of the coprocessor register.

Rn
: is the general-purpose register. `Rn` must not be PC.

Usage

You can use this pseudo-instruction to write to any CP14 or CP15 coprocessor writable register. A complete list of the applicable coprocessor register names is in the *Arm Architecture Reference Manual*. For example:

```
MSR SCTLR, R1 ; writes the contents of R1 into the CP15
              ;  coprocessor register SCTLR
```

Availability

This pseudo-instruction is available in A32 and T32.

This pseudo-instruction is available in Armv7-A and Armv7-R in A32 and 32-bit T32 code.

There is no 16-bit version of this pseudo-instruction in T32.

Related references

C2.62 MRS (PSR to general-purpose register) on page C2-204
C2.63 MRS (system coprocessor register to general-purpose register) on page C2-206
C2.65 MSR (general-purpose register to PSR) on page C2-208
C1.9 Condition code suffixes on page C1-92
C2.153 SYS on page C2-332

Related information

Arm Architecture Reference Manual

C2.65 MSR (general-purpose register to PSR)

Load an immediate value, or the contents of a general-purpose register, into the specified fields of a Program Status Register (PSR).

Syntax

`MSR{cond} APSR_flags, Rm`

where:

cond
is an optional condition code.

flags
specifies the APSR flags to be moved. *flags* can be one or more of:

nzcvq
ALU flags field mask, PSR[31:27] (User mode)

g
SIMD GE flags field mask, PSR[19:16] (User mode).

Rm
is the general-purpose register. R*m* must not be PC.

Syntax on architectures other than Armv6-M, Armv7-M, Armv8-M Baseline, and Armv8-M Mainline

`MSR{cond} APSR_flags, #constant`

`MSR{cond} psr_fields, #constant`

`MSR{cond} psr_fields, Rm`

where:

cond
is an optional condition code.

flags
specifies the APSR flags to be moved. *flags* can be one or more of:

nzcvq
ALU flags field mask, PSR[31:27] (User mode)

g
SIMD GE flags field mask, PSR[19:16] (User mode).

constant
is an expression evaluating to a numeric value. The value must correspond to an 8-bit pattern rotated by an even number of bits within a 32-bit word. Not available in T32.

Rm
is the source register. R*m* must not be PC.

psr
is one of:

CPSR
for use in Debug state, also deprecated synonym for APSR

SPSR
on any processor, in privileged software execution only.

fields
specifies the SPSR or CPSR fields to be moved. *fields* can be one or more of:

c
control field mask byte, PSR[7:0] (privileged software execution)

x
extension field mask byte, PSR[15:8] (privileged software execution)

s
status field mask byte, PSR[23:16] (privileged software execution)

f
flags field mask byte, PSR[31:24] (privileged software execution).

Syntax on architectures Armv6-M, Armv7-M, Armv8-M Baseline, and Armv8-M Mainline only

`MSR{`*cond*`} psr, Rm`

where:

cond
is an optional condition code.

Rm
is the source register. `Rm` must not be PC.

psr
can be any of: `APSR`, `IPSR`, `EPSR`, `IEPSR`, `IAPSR`, `EAPSR`, `XPSR`, `MSP`, `PSP`, `PRIMASK`, `BASEPRI`, `BASEPRI_MAX`, `FAULTMASK`, or `CONTROL`.

Usage

In User mode:

- Use APSR to access the condition flags, Q, or GE bits.
- Writes to unallocated, privileged or execution state bits in the CPSR are ignored. This ensures that User mode programs cannot change to privileged software execution.

Arm deprecates using `MSR` to change the endianness bit (E) of the CPSR, in any mode.

You must not attempt to access the SPSR when the processor is in User or System mode.

Register restrictions

You cannot use PC in A32 instructions. You can use SP for `Rm` in A32 instructions but this is deprecated.

You cannot use PC or SP in T32 instructions.

Condition flags

This instruction updates the flags explicitly if the `APSR_nzcvq` or `CPSR_f` field is specified.

Architectures

This instruction is available in A32 and T32.

There is no 16-bit version of this instruction in T32.

Related references
C2.62 MRS (PSR to general-purpose register) on page C2-204
C2.63 MRS (system coprocessor register to general-purpose register) on page C2-206
C2.64 MSR (general-purpose register to system coprocessor register) on page C2-207
C1.9 Condition code suffixes on page C1-92

C2.66 MUL

Multiply with signed or unsigned 32-bit operands, giving the least significant 32 bits of the result.

Syntax

`MUL{S}{cond} {Rd}, Rn, Rm`

where:

cond
: is an optional condition code.

S
: is an optional suffix. If S is specified, the condition flags are updated on the result of the operation.

Rd
: is the destination register.

Rn, Rm
: are registers holding the values to be multiplied.

Operation

The MUL instruction multiplies the values from *Rn* and *Rm*, and places the least significant 32 bits of the result in *Rd*.

Register restrictions

You cannot use PC for any register.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

Condition flags

If S is specified, the MUL instruction:

- Updates the N and Z flags according to the result.
- Does not affect the C or V flag.

16-bit instructions

The following forms of the MUL instruction are available in T32 code, and are 16-bit instructions:

MULS *Rd, Rn, Rd*
: *Rd* and *Rn* must both be Lo registers. This form can only be used outside an IT block.

MUL{*cond*} *Rd, Rn, Rd*
: *Rd* and *Rn* must both be Lo registers. This form can only be used inside an IT block.

There are no other T32 multiply instructions that can update the condition flags.

Availability

This instruction is available in A32 and T32.

The MULS instruction is available in T32 in a 16-bit encoding.

Examples

```
MUL     r10, r2, r5
MULS    r0, r2, r2
MULLT   r2, r3, r2
```

Related references

C1.9 Condition code suffixes on page C1-92

C2.67 MVN

Move Not.

Syntax

`MVN{S}{cond} Rd, Operand2`

where:

`S`
: is an optional suffix. If `S` is specified, the condition flags are updated on the result of the operation.

`cond`
: is an optional condition code.

`Rd`
: is the destination register.

`Operand2`
: is a flexible second operand.

Operation

The `MVN` instruction takes the value of *`Operand2`*, performs a bitwise logical NOT operation on the value, and places the result into *`Rd`*.

In certain circumstances, the assembler can substitute `MVN` for `MOV`, or `MOV` for `MVN`. Be aware of this when reading disassembly listings.

Use of PC and SP in 32-bit T32 MVN

You cannot use PC (`R15`) for *`Rd`*, or in *`Operand2`*, in 32-bit T32 `MVN` instructions. You cannot use SP (`R13`) for *`Rd`*, or in *`Operand2`*.

Use of PC and SP in 16-bit T32 instructions

You cannot use PC or SP in any `MVN{S}` 16-bit T32 instructions.

Use of PC and SP in A32 MVN

You cannot use PC for *`Rd`* or any operand in any data processing instruction that has a register-controlled shift.

In instructions without register-controlled shift, use of PC is deprecated.

You can use SP for `Rd` or `Rm`, but this is deprecated.

Note

- PC and SP in A32 instructions are deprecated.

If you use PC as *`Rm`*, the value used is the address of the instruction plus 8.

If you use PC as *`Rd`*:

- Execution branches to the address corresponding to the result.
- If you use the `S` suffix, see the `SUBS pc,lr` instruction.

Condition flags

If `S` is specified, the instruction:

- Updates the N and Z flags according to the result.
- Can update the C flag during the calculation of *`Operand2`*.
- Does not affect the V flag.

16-bit instructions

The following forms of this instruction are available in T32 code, and are 16-bit instructions:

MVNS *Rd, Rm*

Rd and *Rm* must both be Lo registers. This form can only be used outside an IT block.

MVN{*cond***}** *Rd, Rm*

Rd and *Rm* must both be Lo registers. This form can only be used inside an IT block.

Architectures

This instruction is available in A32 and T32.

Correct example

```
    MVNNE   r11, #0xF000000B ; A32 only. This immediate value is not
                             ; available in T32.
```

Incorrect example

```
    MVN     pc,r3,ASR r0     ; PC not permitted with
                             ; register-controlled shift
```

Related references

C2.3 Flexible second operand (Operand2) on page C2-112

C2.144 SUBS pc, lr on page C2-317

C1.9 Condition code suffixes on page C1-92

C2.68 NOP

No Operation.

Syntax

`NOP{`*cond*`}`

where:

cond
is an optional condition code.

Usage

`NOP` does nothing. If `NOP` is not implemented as a specific instruction on your target architecture, the assembler treats it as a pseudo-instruction and generates an alternative instruction that does nothing, such as `MOV r0, r0` (A32) or `MOV r8, r8` (T32).

`NOP` is not necessarily a time-consuming `NOP`. The processor might remove it from the pipeline before it reaches the execution stage.

You can use `NOP` for padding, for example to place the following instruction on a 64-bit boundary in A32, or a 32-bit boundary in T32.

Architectures

This instruction is available in A32 and T32.

Related references
C1.9 Condition code suffixes on page C1-92

C2.69 ORN (T32 only)

Logical OR NOT.

Syntax

`ORN{S}{cond} Rd, Rn, Operand2`

where:

`S`
: is an optional suffix. If `S` is specified, the condition flags are updated on the result of the operation.

`cond`
: is an optional condition code.

`Rd`
: is the destination register.

`Rn`
: is the register holding the first operand.

`Operand2`
: is a flexible second operand.

Operation

The `ORN` T32 instruction performs an OR operation on the bits in *`Rn`* with the complements of the corresponding bits in the value of *`Operand2`*.

In certain circumstances, the assembler can substitute `ORN` for `ORR`, or `ORR` for `ORN`. Be aware of this when reading disassembly listings.

Use of PC

You cannot use PC (`R15`) for *`Rd`* or any operand in the `ORN` instruction.

Condition flags

If `S` is specified, the `ORN` instruction:

- Updates the N and Z flags according to the result.
- Can update the C flag during the calculation of *`Operand2`*.
- Does not affect the V flag.

Examples

```
ORN     r7, r11, lr, ROR #4
ORNS    r7, r11, lr, ASR #32
```

Architectures

This 32-bit instruction is available in T32.

There is no A32 or 16-bit T32 `ORN` instruction.

Related references

C2.3 Flexible second operand (Operand2) on page C2-112
C2.144 SUBS pc, lr on page C2-317
C1.9 Condition code suffixes on page C1-92

C2.70 ORR

Logical OR.

Syntax

`ORR{S}{cond} Rd, Rn, Operand2`

where:

`S`
is an optional suffix. If `S` is specified, the condition flags are updated on the result of the operation.

`cond`
is an optional condition code.

`Rd`
is the destination register.

`Rn`
is the register holding the first operand.

`Operand2`
is a flexible second operand.

Operation

The `ORR` instruction performs bitwise OR operations on the values in *`Rn`* and *`Operand2`*.

In certain circumstances, the assembler can substitute `ORN` for `ORR`, or `ORR` for `ORN`. Be aware of this when reading disassembly listings.

Use of PC in 32-bit T32 instructions

You cannot use PC (`R15`) for *`Rd`* or any operand with the `ORR` instruction.

Use of PC and SP in A32 instructions

You can use PC and SP with the `ORR` instruction but this is deprecated.

If you use PC as *`Rn`*, the value used is the address of the instruction plus 8.

If you use PC as *`Rd`*:
- Execution branches to the address corresponding to the result.
- If you use the `S` suffix, see the `SUBS pc,lr` instruction.

You cannot use PC for any operand in any data processing instruction that has a register-controlled shift.

Condition flags

If `S` is specified, the `ORR` instruction:
- Updates the N and Z flags according to the result.
- Can update the C flag during the calculation of *`Operand2`*.
- Does not affect the V flag.

16-bit instructions

The following forms of the `ORR` instruction are available in T32 code, and are 16-bit instructions:

`ORRS` ***`Rd, Rd, Rm`***
`Rd` and *`Rm`* must both be Lo registers. This form can only be used outside an IT block.

`ORR{`*`cond`*`}` ***`Rd, Rd, Rm`***
`Rd` and *`Rm`* must both be Lo registers. This form can only be used inside an IT block.

It does not matter if you specify `ORR{S} Rd, Rm, Rd`. The instruction is the same.

Example

```
    ORREQ   r2,r0,r5
```

Related references

C2.3 Flexible second operand (Operand2) on page C2-112

C2.144 SUBS pc, lr on page C2-317

C1.9 Condition code suffixes on page C1-92

C2.71 PKHBT and PKHTB

Halfword Packing instructions that combine a halfword from one register with a halfword from another register. One of the operands can be shifted before extraction of the halfword.

Syntax

`PKHBT{cond} {Rd}, Rn, Rm{, LSL #leftshift}`

`PKHTB{cond} {Rd}, Rn, Rm{, ASR #rightshift}`

where:

PKHBT

Combines bits[15:0] of *Rn* with bits[31:16] of the shifted value from *Rm*.

PKHTB

Combines bits[31:16] of *Rn* with bits[15:0] of the shifted value from *Rm*.

cond

is an optional condition code.

Rd

is the destination register.

Rn

is the register holding the first operand.

Rm

is the register holding the first operand.

leftshift

is in the range 0 to 31.

rightshift

is in the range 1 to 32.

Register restrictions

You cannot use PC for any register.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

Condition flags

These instructions do not change the flags.

Architectures

These instructions are available in A32.

These 32-bit instructions are available T32. For the Armv7-M architecture, they are only available in an Armv7E-M implementation.

There are no 16-bit versions of these instructions in T32.

Correct examples

```
PKHBT   r0, r3, r5          ; combine the bottom halfword of R3
                            ; with the top halfword of R5
PKHBT   r0, r3, r5, LSL #16 ; combine the bottom halfword of R3
                            ; with the bottom halfword of R5
```

```
    PKHTB   r0, r3, r5, ASR #16 ; combine the top halfword of R3
                                ; with the top halfword of R5
```

You can also scale the second operand by using different values of shift.

Incorrect example

```
    PKHBTEQ r4, r5, r1, ASR #8  ; ASR not permitted with PKHBT
```

Related references

C1.9 Condition code suffixes on page C1-92

C2.72 PLD, PLDW, and PLI

Preload Data and Preload Instruction allow the processor to signal the memory system that a data or instruction load from an address is likely in the near future.

Syntax

`PL`*`type`*`{`*`cond`*`} [`*`Rn`* `{, #`*`offset`*`}]`

`PL`*`type`*`{`*`cond`*`} [`*`Rn`*`, ±`*`Rm`* `{,` *`shift`*`}]`

`PL`*`type`*`{`*`cond`*`}` *`label`*

where:

`type`

can be one of:

`D`

Data address.

`DW`

Data address with intention to write.

`I`

Instruction address.

`type` cannot be `DW` if the syntax specifies *`label`*.

`cond`

is an optional condition code.

Note

`cond` is permitted only in T32 code, using a preceding `IT` instruction, but this is deprecated in the Armv8 architecture. This is an unconditional instruction in A32 code and you must not use *`cond`*.

`Rn`

is the register on which the memory address is based.

`offset`

is an immediate offset. If offset is omitted, the address is the value in *`Rn`*.

`Rm`

is a register containing a value to be used as the offset.

`shift`

is an optional shift.

`label`

is a PC-relative expression.

Range of offsets

The offset is applied to the value in *`Rn`* before the preload takes place. The result is used as the memory address for the preload. The range of offsets permitted is:

- -4095 to +4095 for A32 instructions.
- -255 to +4095 for T32 instructions, when *`Rn`* is not PC.
- -4095 to +4095 for T32 instructions, when *`Rn`* is PC.

The assembler calculates the offset from the PC for you. The assembler generates an error if *label* is out of range.

Register or shifted register offset

In A32 code, the value in *Rm* is added to or subtracted from the value in *Rn*. In T32 code, the value in *Rm* can only be added to the value in *Rn*. The result is used as the memory address for the preload.

The range of shifts permitted is:

- LSL #0 to #3 for T32 instructions.
- Any one of the following for A32 instructions:
  - LSL #0 to #31.
  - LSR #1 to #32.
  - ASR #1 to #32.
  - ROR #1 to #31.
  - RRX.

Address alignment for preloads

No alignment checking is performed for preload instructions.

Register restrictions

Rm must not be PC. For T32 instructions *Rm* must also not be SP.

Rn must not be PC for T32 instructions of the syntax PL*type*{*cond*} [R*n*, ±R*m*{, #*shift*}].

Architectures

The PLD instruction is available in A32.

The 32-bit encoding of PLD is available in T32.

PLDW is available only in the Armv7 architecture and above that implement the Multiprocessing Extensions.

PLI is available only in the Armv7 architecture and above.

There are no 16-bit encodings of these instructions in T32.

These are hint instructions, and their implementation is optional. If they are not implemented, they execute as NOPs.

Related references

C1.9 Condition code suffixes on page C1-92

C2.73 POP

Pop registers off a full descending stack.

Syntax

`POP{`*`cond`*`} `*`reglist`*

where:

`cond`

is an optional condition code.

`reglist`

is a non-empty list of registers, enclosed in braces. It can contain register ranges. It must be comma separated if it contains more than one register or register range.

Operation

`POP` is a synonym for `LDMIA sp! reglist`. `POP` is the preferred mnemonic.

Note

`LDM` and `LDMFD` are synonyms of `LDMIA`.

Registers are stored on the stack in numerical order, with the lowest numbered register at the lowest address.

POP, with reglist including the PC

This instruction causes a branch to the address popped off the stack into the PC. This is usually a return from a subroutine, where the LR was pushed onto the stack at the start of the subroutine.

Also:

- Bits[1:0] must not be 0b10.
- If bit[0] is 1, execution continues in T32 state.
- If bit[0] is 0, execution continues in A32 state.

T32 instructions

A subset of this instruction is available in the T32 instruction set.

The following restriction applies to the 16-bit `POP` instruction:

- *`reglist`* can only include the Lo registers and the PC.

The following restrictions apply to the 32-bit `POP` instruction:

- *`reglist`* must not include the SP.
- *`reglist`* can include either the LR or the PC, but not both.

Restrictions on reglist in A32 instructions

The A32 `POP` instruction cannot have SP but can have PC in the *`reglist`*. The instruction that includes both PC and LR in the *`reglist`* is deprecated.

Example

```
        POP     {r0,r10,pc} ; no 16-bit version available
```

Related references

C2.45 LDM on page C2-177
C2.74 PUSH on page C2-222
C1.9 Condition code suffixes on page C1-92

C2.74 PUSH

Push registers onto a full descending stack.

Syntax

`PUSH{`*`cond`*`} `*`reglist`*

where:

`cond`

is an optional condition code.

`reglist`

is a non-empty list of registers, enclosed in braces. It can contain register ranges. It must be comma separated if it contains more than one register or register range.

Operation

`PUSH` is a synonym for `STMDB sp!, reglist`. `PUSH` is the preferred mnemonic.

Note

`STMFD` is a synonym of `STMDB`.

Registers are stored on the stack in numerical order, with the lowest numbered register at the lowest address.

T32 instructions

The following restriction applies to the 16-bit `PUSH` instruction:

- *`reglist`* can only include the Lo registers and the LR.

The following restrictions apply to the 32-bit `PUSH` instruction:

- *`reglist`* must not include the SP.
- *`reglist`* must not include the PC.

Restrictions on reglist in A32 instructions

The A32 `PUSH` instruction can have SP and PC in the *`reglist`* but the instruction that includes SP or PC in the *`reglist`* is deprecated.

Examples

```
    PUSH    {r0,r4-r7}
    PUSH    {r2,lr}
```

Related references

C2.45 LDM on page C2-177
C2.73 POP on page C2-221
C1.9 Condition code suffixes on page C1-92

C2.75 QADD

Signed saturating addition.

Syntax

`QADD{cond} {Rd}, Rm, Rn`

where:

cond

is an optional condition code.

Rd

is the destination register.

Rm, Rn

are the registers holding the operands.

Operation

The `QADD` instruction adds the values in *Rm* and *Rn*. It saturates the result to the signed range $-2^{31} \leq x \leq 2^{31}-1$.

Note

All values are treated as two's complement signed integers by this instruction.

Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

Q flag

If saturation occurs, this instruction sets the Q flag. To read the state of the Q flag, use an `MRS` instruction.

Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

Example

```
QADD    r0, r1, r9
```

Related references

C2.62 MRS (PSR to general-purpose register) on page C2-204

A1.11 The Q flag in AArch32 state on page A1-37

C1.9 Condition code suffixes on page C1-92

C2.76 QADD8

Signed saturating parallel byte-wise addition.

Syntax

`QADD8{cond} {Rd}, Rn, Rm`

where:

cond

is an optional condition code.

Rd

is the destination register.

Rm, Rn

are the general-purpose registers holding the operands.

Operation

This instruction performs four signed integer additions on the corresponding bytes of the operands and writes the results into the corresponding bytes of the destination. It saturates the results to the signed range $-2^7 \leq x \leq 2^7 - 1$. The Q flag is not affected even if this operation saturates.

Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

Condition flags

This instruction does not affect the N, Z, C, V, Q, or GE flags.

Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

Related references

A1.11 The Q flag in AArch32 state on page A1-37
C1.9 Condition code suffixes on page C1-92

C2.77 QADD16

Signed saturating parallel halfword-wise addition.

Syntax

`QADD16{cond} {Rd}, Rn, Rm`

where:

cond

is an optional condition code.

Rd

is the destination register.

Rm, Rn

are the general-purpose registers holding the operands.

Operation

This instruction performs two signed integer additions on the corresponding halfwords of the operands and writes the results into the corresponding halfwords of the destination. It saturates the results to the signed range $-2^{15} \leq x \leq 2^{15}-1$. The Q flag is not affected even if this operation saturates.

Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

Condition flags

This instruction does not affect the N, Z, C, V, Q, or GE flags.

Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

Related references

A1.11 The Q flag in AArch32 state on page A1-37
C1.9 Condition code suffixes on page C1-92

C2.78 QASX

Signed saturating parallel add and subtract halfwords with exchange.

Syntax

`QASX{cond} {Rd}, Rn, Rm`

where:

cond

is an optional condition code.

Rd

is the destination register.

Rm, Rn

are the general-purpose registers holding the operands.

Operation

This instruction exchanges the two halfwords of the second operand, then performs an addition on the two top halfwords of the operands and a subtraction on the bottom two halfwords. It writes the results into the corresponding halfwords of the destination. It saturates the results to the signed range $-2^{15} \leq x \leq 2^{15}-1$. The Q flag is not affected even if this operation saturates.

Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

Condition flags

This instruction does not affect the N, Z, C, V, Q, or GE flags.

Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

Related references

A1.11 The Q flag in AArch32 state on page A1-37

C1.9 Condition code suffixes on page C1-92

C2.79 QDADD

Signed saturating Double and Add.

Syntax

`QDADD{cond} {Rd}, Rm, Rn`

where:

cond

is an optional condition code.

Rd

is the destination register.

Rm, Rn

are the general-purpose registers holding the operands.

Operation

`QDADD` calculates `SAT(Rm + SAT(Rn * 2))`. It saturates the result to the signed range $-2^{31} \leq x \leq 2^{31}-1$. Saturation can occur on the doubling operation, on the addition, or on both. If saturation occurs on the doubling but not on the addition, the Q flag is set but the final result is unsaturated.

Note

All values are treated as two's complement signed integers by this instruction.

Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

Q flag

If saturation occurs, this instruction sets the Q flag. To read the state of the Q flag, use an `MRS` instruction.

Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

Related references

A1.11 The Q flag in AArch32 state on page A1-37
C2.62 MRS (PSR to general-purpose register) on page C2-204
C1.9 Condition code suffixes on page C1-92

C2.80 QDSUB

Signed saturating Double and Subtract.

Syntax

`QDSUB{cond} {Rd}, Rm, Rn`

where:

cond

is an optional condition code.

Rd

is the destination register.

Rm, Rn

are the general-purpose registers holding the operands.

Operation

`QDSUB` calculates `SAT(Rm - SAT(Rn * 2))`. It saturates the result to the signed range $-2^{31} \leq x \leq 2^{31}-1$. Saturation can occur on the doubling operation, on the subtraction, or on both. If saturation occurs on the doubling but not on the subtraction, the Q flag is set but the final result is unsaturated.

Note

All values are treated as two's complement signed integers by this instruction.

Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

Q flag

If saturation occurs, this instruction sets the Q flag. To read the state of the Q flag, use an `MRS` instruction.

Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

Example

```
QDSUBLT r9, r0, r1
```

Related references

A1.11 The Q flag in AArch32 state on page A1-37

C2.62 MRS (PSR to general-purpose register) on page C2-204

C1.9 Condition code suffixes on page C1-92

C2.81 QSAX

Signed saturating parallel subtract and add halfwords with exchange.

Syntax

`QSAX{cond} {Rd}, Rn, Rm`

where:

cond

is an optional condition code.

Rd

is the destination register.

Rm, Rn

are the general-purpose registers holding the operands.

Operation

This instruction exchanges the two halfwords of the second operand, then performs a subtraction on the two top halfwords of the operands and an addition on the bottom two halfwords. It writes the results into the corresponding halfwords of the destination. It saturates the results to the signed range $-2^{15} \leq x \leq 2^{15}-1$. The Q flag is not affected even if this operation saturates.

Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

Condition flags

This instruction does not affect the N, Z, C, V, Q, or GE flags.

Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

Related references

A1.11 The Q flag in AArch32 state on page A1-37
C1.9 Condition code suffixes on page C1-92

C2.82 QSUB

Signed saturating Subtract.

Syntax

`QSUB{cond} {Rd}, Rm, Rn`

where:

cond

is an optional condition code.

Rd

is the destination register.

Rm, Rn

are the general-purpose registers holding the operands.

Operation

The `QSUB` instruction subtracts the value in *Rn* from the value in *Rm*. It saturates the result to the signed range $-2^{31} \leq x \leq 2^{31}-1$.

Note

All values are treated as two's complement signed integers by this instruction.

Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

Q flag

If saturation occurs, this instruction sets the Q flag. To read the state of the Q flag, use an `MRS` instruction.

Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

Related references

A1.11 The Q flag in AArch32 state on page A1-37
C2.62 MRS (PSR to general-purpose register) on page C2-204
C1.9 Condition code suffixes on page C1-92

C2.83 QSUB8

Signed saturating parallel byte-wise subtraction.

Syntax

`QSUB8{cond} {Rd}, Rn, Rm`

where:

cond

is an optional condition code.

Rd

is the destination register.

Rm, Rn

are the general-purpose registers holding the operands.

Operation

This instruction subtracts each byte of the second operand from the corresponding byte of the first operand and writes the results into the corresponding bytes of the destination. It saturates the results to the signed range $-2^7 \leq x \leq 2^7 - 1$. The Q flag is not affected even if this operation saturates.

Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

Condition flags

This instruction does not affect the N, Z, C, V, Q, or GE flags.

Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

Related references

A1.11 The Q flag in AArch32 state on page A1-37
C1.9 Condition code suffixes on page C1-92

C2.84 QSUB16

Signed saturating parallel halfword-wise subtraction.

Syntax

`QSUB16{cond} {Rd}, Rn, Rm`

where:

cond

is an optional condition code.

Rd

is the destination register.

Rm, Rn

are the general-purpose registers holding the operands.

Operation

This instruction subtracts each halfword of the second operand from the corresponding halfword of the first operand and writes the results into the corresponding halfwords of the destination. It saturates the results to the signed range $-2^{15} \leq x \leq 2^{15}-1$. The Q flag is not affected even if this operation saturates.

Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

Condition flags

This instruction does not affect the N, Z, C, V, Q, or GE flags.

Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

Related references

A1.11 The Q flag in AArch32 state on page A1-37
C1.9 Condition code suffixes on page C1-92

C2.85 RBIT

Reverse the bit order in a 32-bit word.

Syntax

`RBIT{cond} Rd, Rn`

where:

cond

is an optional condition code.

Rd

is the destination register.

Rn

is the register holding the operand.

Register restrictions

You cannot use PC for any register.

You can use SP in the A32 instruction but this is deprecated. You cannot use SP in the T32 instruction.

Condition flags

This instruction does not change the flags.

Architectures

This instruction is available in A32 and T32.

There is no 16-bit version of this instruction in T32.

Example

```
    RBIT    r7, r8
```

Related references

C1.9 Condition code suffixes on page C1-92

C2.86 REV

Reverse the byte order in a word.

Syntax

`REV{cond} Rd, Rn`

where:

cond

is an optional condition code.

Rd

is the destination register.

Rn

is the register holding the operand.

Usage

You can use this instruction to change endianness. `REV` converts 32-bit big-endian data into little-endian data or 32-bit little-endian data into big-endian data.

Register restrictions

You cannot use PC for any register.

You can use SP in the A32 instruction but this is deprecated. You cannot use SP in the T32 instruction.

Condition flags

This instruction does not change the flags.

16-bit instructions

The following form of this instruction is available in T32 code, and is a 16-bit instruction:

`REV` ***`Rd, Rm`***

Rd and *Rm* must both be Lo registers.

Architectures

This instruction is available in A32 and T32.

Example

```
    REV     r3, r7
```

Related references

C1.9 Condition code suffixes on page C1-92

C2.87 REV16

Reverse the byte order in each halfword independently.

Syntax

`REV16{cond} Rd, Rn`

where:

cond

is an optional condition code.

Rd

is the destination register.

Rn

is the register holding the operand.

Usage

You can use this instruction to change endianness. `REV16` converts 16-bit big-endian data into little-endian data or 16-bit little-endian data into big-endian data.

Register restrictions

You cannot use PC for any register.

You can use SP in the A32 instruction but this is deprecated. You cannot use SP in the T32 instruction.

Condition flags

This instruction does not change the flags.

16-bit instructions

The following form of this instruction is available in T32 code, and is a 16-bit instruction:

`REV16 Rd, Rm`

Rd and *Rm* must both be Lo registers.

Architectures

This instruction is available in A32 and T32.

Example

```
    REV16   r0, r0
```

Related references

C1.9 Condition code suffixes on page C1-92

C2.88 REVSH

Reverse the byte order in the bottom halfword, and sign extend to 32 bits.

Syntax

`REVSH{cond} Rd, Rn`

where:

cond

is an optional condition code.

Rd

is the destination register.

Rn

is the register holding the operand.

Usage

You can use this instruction to change endianness. `REVSH` converts either:

- 16-bit signed big-endian data into 32-bit signed little-endian data.
- 16-bit signed little-endian data into 32-bit signed big-endian data.

Register restrictions

You cannot use PC for any register.

You can use SP in the A32 instruction but this is deprecated. You cannot use SP in the T32 instruction.

Condition flags

This instruction does not change the flags.

16-bit instructions

The following form of this instruction is available in T32 code, and is a 16-bit instruction:

`REVSH` ***`Rd, Rm`***

Rd and *Rm* must both be Lo registers.

Architectures

This instruction is available in A32 and T32.

Example

```
    REVSH   r0, r5       ; Reverse Signed Halfword
```

Related references

C1.9 Condition code suffixes on page C1-92

C2.89 RFE

Return From Exception.

Syntax

`RFE{`*`addr_mode`*`}{`*`cond`*`} `*`Rn`*`{!}`

where:

`addr_mode`

is any one of the following:

`IA`
Increment address After each transfer (Full Descending stack)

`IB`
Increment address Before each transfer (A32 only)

`DA`
Decrement address After each transfer (A32 only)

`DB`
Decrement address Before each transfer.

If *`addr_mode`* is omitted, it defaults to Increment After.

`cond`

is an optional condition code.

Note

`cond` is permitted only in T32 code, using a preceding `IT` instruction, but this is deprecated in Armv8. This is an unconditional instruction in A32 code.

`Rn`

specifies the base register. *`Rn`* must not be PC.

`!`

is an optional suffix. If `!` is present, the final address is written back into *`Rn`*.

Usage

You can use `RFE` to return from an exception if you previously saved the return state using the `SRS` instruction. *`Rn`* is usually the `SP` where the return state information was saved.

Operation

Loads the PC and the CPSR from the address contained in *`Rn`*, and the following address. Optionally updates *`Rn`*.

Notes

`RFE` writes an address to the PC. The alignment of this address must be correct for the instruction set in use after the exception return:

- For a return to A32, the address written to the PC must be word-aligned.
- For a return to T32, the address written to the PC must be halfword-aligned.
- For a return to Jazelle, there are no alignment restrictions on the address written to the PC.

No special precautions are required in software to follow these rules, if you use the instruction to return after a valid exception entry mechanism.

Where addresses are not word-aligned, `RFE` ignores the least significant two bits of *`Rn`*.

The time order of the accesses to individual words of memory generated by `RFE` is not architecturally defined. Do not use this instruction on memory-mapped I/O locations where access order matters.

Do not use `RFE` in unprivileged software execution.

Architectures

This instruction is available in A32.

This 32-bit T32 instruction is available, except in the Armv7-M and Armv8-M Mainline architectures.

There is no 16-bit version of this instruction.

Example

```
    RFE sp!
```

Related concepts

A1.3 Processor modes, and privileged and unprivileged software execution on page A1-28

Related references

C2.129 SRS on page C2-289

C1.9 Condition code suffixes on page C1-92

C2.90 ROR

Rotate Right. This instruction is a preferred synonym for `MOV` instructions with shifted register operands.

Syntax

`ROR{S}{`*cond*`} Rd, Rm, Rs`

`ROR{S}{`*cond*`} Rd, Rm, #sh`

where:

`S`

is an optional suffix. If `S` is specified, the condition flags are updated on the result of the operation.

`Rd`

is the destination register.

`Rm`

is the register holding the first operand. This operand is shifted right.

`Rs`

is a register holding a shift value to apply to the value in *`Rm`*. Only the least significant byte is used.

`sh`

is a constant shift. The range of values is 1-31.

Operation

`ROR` provides the value of the contents of a register rotated by a value. The bits that are rotated off the right end are inserted into the vacated bit positions on the left.

Restrictions in T32 code

T32 instructions must not use PC or SP.

Use of SP and PC in A32 instructions

You can use SP in these A32 instructions but this is deprecated.

You cannot use PC in instructions with the `ROR{S}{cond} Rd, Rm, Rs` syntax. You can use PC for `R`*`d`* and `R`*`m`* in the other syntax, but this is deprecated.

If you use PC as *`Rm`*, the value used is the address of the instruction plus 8.

If you use PC as *`Rd`*:

- Execution branches to the address corresponding to the result.
- If you use the `S` suffix, the SPSR of the current mode is copied to the CPSR. You can use this to return from exceptions.

  Note

  The A32 instruction `RORS{`*cond*`} pc,Rm,#sh` always disassembles to the preferred form `MOVS{`*cond*`} pc,Rm{,shift}`.

Caution

Do not use the `S` suffix when using PC as *`Rd`* in User mode or System mode. The assembler cannot warn you about this because it has no information about what the processor mode is likely to be at execution time.

You cannot use PC for *`Rd`* or any operand in this instruction if it has a register-controlled shift.

Condition flags

If `S` is specified, the instruction updates the N and Z flags according to the result.

The C flag is unaffected if the shift value is 0. Otherwise, the C flag is updated to the last bit shifted out.

16-bit instructions

The following forms of this instruction are available in T32 code, and are 16-bit instructions:

`RORS` ***`Rd, Rd, Rs`***

`Rd` and *`Rs`* must both be Lo registers. This form can only be used outside an IT block.

`ROR{`*`cond`***`}`** ***`Rd, Rd, Rs`***

`Rd` and *`Rs`* must both be Lo registers. This form can only be used inside an IT block.

Architectures

This instruction is available in A32 and T32.

Example

```
    ROR     r4, r5, r6
```

Related references

C2.58 MOV on page C2-199

C1.9 Condition code suffixes on page C1-92

C2.91 RRX

Rotate Right with Extend. This instruction is a preferred synonym for `MOV` instructions with shifted register operands.

Syntax

`RRX{S}{`*`cond`*`} `*`Rd, Rm`*

where:

`S`

is an optional suffix. If `S` is specified, the condition flags are updated on the result of the operation.

`Rd`

is the destination register.

`Rm`

is the register holding the first operand. This operand is shifted right.

Operation

`RRX` provides the value of the contents of a register shifted right one bit. The old carry flag is shifted into bit[31]. If the `S` suffix is present, the old bit[0] is placed in the carry flag.

Restrictions in T32 code

T32 instructions must not use PC or SP.

Use of SP and PC in A32 instructions

You can use SP in this A32 instruction but this is deprecated.

If you use PC as *`Rm`*, the value used is the address of the instruction plus 8.

If you use PC as *`Rd`*:

- Execution branches to the address corresponding to the result.
- If you use the `S` suffix, the SPSR of the current mode is copied to the CPSR. You can use this to return from exceptions.

Note

The A32 instruction `RRXS{cond} pc,Rm` always disassembles to the preferred form `MOVS{`*`cond`*`} pc,Rm{,shift}`.

Caution

Do not use the `S` suffix when using PC as *`Rd`* in User mode or System mode. The assembler cannot warn you about this because it has no information about what the processor mode is likely to be at execution time.

You cannot use PC for *`Rd`* or any operand in this instruction if it has a register-controlled shift.

Condition flags

If `S` is specified, the instruction updates the N and Z flags according to the result.

The C flag is unaffected if the shift value is 0. Otherwise, the C flag is updated to the last bit shifted out.

Architectures

The 32-bit instruction is available in A32 and T32.

There is no 16-bit instruction in T32.

Related references

C2.58 MOV on page C2-199
C1.9 Condition code suffixes on page C1-92

C2.92 RSB

Reverse Subtract without carry.

Syntax

`RSB{S}{cond} {Rd}, Rn, Operand2`

where:

`S`

is an optional suffix. If `S` is specified, the condition flags are updated on the result of the operation.

`cond`

is an optional condition code.

`Rd`

is the destination register.

`Rn`

is the register holding the first operand.

`Operand2`

is a flexible second operand.

Operation

The `RSB` instruction subtracts the value in *`Rn`* from the value of *`Operand2`*. This is useful because of the wide range of options for *`Operand2`*.

In certain circumstances, the assembler can substitute one instruction for another. Be aware of this when reading disassembly listings.

Use of PC and SP in T32 instructions

You cannot use PC (`R15`) for *`Rd`* or any operand.

You cannot use SP (`R13`) for *`Rd`* or any operand.

Use of PC and SP in A32 instructions

You cannot use PC for *`Rd`* or any operand in an `RSB` instruction that has a register-controlled shift.

Use of PC for any operand, in instructions without register-controlled shift, is deprecated.

If you use PC (`R15`) as *`Rn`* or *`Rm`*, the value used is the address of the instruction plus 8.

If you use PC as *`Rd`*:

- Execution branches to the address corresponding to the result.
- If you use the `S` suffix, see the `SUBS pc,lr` instruction.

Use of SP and PC in A32 instructions is deprecated.

Condition flags

If `S` is specified, the `RSB` instruction updates the N, Z, C and V flags according to the result.

16-bit instructions

The following forms of this instruction are available in T32 code, and are 16-bit instructions:

RSBS ***Rd, Rn, #0***

Rd and *Rn* must both be Lo registers. This form can only be used outside an IT block.

RSB{*cond***}** ***Rd, Rn, #0***

Rd and *Rn* must both be Lo registers. This form can only be used inside an IT block.

Example

```
    RSB     r4, r4, #1280       ; subtracts contents of R4 from 1280
```

Related references

C2.3 Flexible second operand (Operand2) on page C2-112

C1.9 Condition code suffixes on page C1-92

C2.93 RSC

Reverse Subtract with Carry.

Syntax

`RSC{S}{cond} {Rd}, Rn, Operand2`

where:

S

is an optional suffix. If S is specified, the condition flags are updated on the result of the operation.

cond

is an optional condition code.

Rd

is the destination register.

Rn

is the register holding the first operand.

Operand2

is a flexible second operand.

Usage

The RSC instruction subtracts the value in *Rn* from the value of *Operand2*. If the carry flag is clear, the result is reduced by one.

You can use RSC to synthesize multiword arithmetic.

In certain circumstances, the assembler can substitute one instruction for another. Be aware of this when reading disassembly listings.

RSC is not available in T32 code.

Use of PC and SP

Use of PC and SP is deprecated.

You cannot use PC for *Rd* or any operand in an RSC instruction that has a register-controlled shift.

If you use PC (R15) as *Rn* or *Rm*, the value used is the address of the instruction plus 8.

If you use PC as *Rd*:

- Execution branches to the address corresponding to the result.
- If you use the S suffix, see the `SUBS pc,lr` instruction.

Condition flags

If S is specified, the RSC instruction updates the N, Z, C and V flags according to the result.

Correct example

```
RSCSLE  r0,r5,r0,LSL r4     ; conditional, flags set
```

Incorrect example

```
RSCSLE  r0,pc,r0,LSL r4    ; PC not permitted with register
                           ; controlled shift
```

Related references

C2.3 Flexible second operand (Operand2) on page C2-112

C1.9 Condition code suffixes on page C1-92

C2.94 SADD8

Signed parallel byte-wise addition.

Syntax

`SADD8{cond} {Rd}, Rn, Rm`

where:

cond

is an optional condition code.

Rd

is the destination register.

Rm, Rn

are the general-purpose registers holding the operands.

Operation

This instruction performs four signed integer additions on the corresponding bytes of the operands and writes the results into the corresponding bytes of the destination. The results are modulo 2^8. It sets the APSR GE flags.

Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

GE flags

This instruction does not affect the N, Z, C, V, or Q flags.

It sets the GE flags in the APSR as follows:

GE[0]

for bits[7:0] of the result.

GE[1]

for bits[15:8] of the result.

GE[2]

for bits[23:16] of the result.

GE[3]

for bits[31:24] of the result.

It sets a GE flag to 1 to indicate that the corresponding result is greater than or equal to zero. This is equivalent to an `ADDS` instruction setting the N and V condition flags to the same value, so that the GE condition passes.

You can use these flags to control a following `SEL` instruction.

Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

Related references

C2.100 SEL on page C2-257

C1.9 Condition code suffixes on page C1-92

C2.95 SADD16

Signed parallel halfword-wise addition.

Syntax

`SADD16{cond} {Rd}, Rn, Rm`

where:

cond

is an optional condition code.

Rd

is the destination register.

Rm, Rn

are the general-purpose registers holding the operands.

Operation

This instruction performs two signed integer additions on the corresponding halfwords of the operands and writes the results into the corresponding halfwords of the destination. The results are modulo 2^{16}. It sets the APSR GE flags.

Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

GE flags

This instruction does not affect the N, Z, C, V, or Q flags.

It sets the GE flags in the APSR as follows:

GE[1:0]

for bits[15:0] of the result.

GE[3:2]

for bits[31:16] of the result.

It sets a pair of GE flags to 1 to indicate that the corresponding result is greater than or equal to zero. This is equivalent to an `ADDS` instruction setting the N and V condition flags to the same value, so that the GE condition passes.

You can use these flags to control a following `SEL` instruction.

Note

GE[1:0] are set or cleared together, and GE[3:2] are set or cleared together.

Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

Related references

C2.100 SEL on page C2-257
C1.9 Condition code suffixes on page C1-92

C2.96 SASX

Signed parallel add and subtract halfwords with exchange.

Syntax

`SASX{cond} {Rd}, Rn, Rm`

where:

cond

is an optional condition code.

Rd

is the destination register.

Rm, Rn

are the general-purpose registers holding the operands.

Operation

This instruction exchanges the two halfwords of the second operand, then performs an addition on the two top halfwords of the operands and a subtraction on the bottom two halfwords. It writes the results into the corresponding halfwords of the destination. The results are modulo 2^{16}. It sets the APSR GE flags.

Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

GE flags

This instruction does not affect the N, Z, C, V, or Q flags.

It sets the GE flags in the APSR as follows:

GE[1:0]

for bits[15:0] of the result.

GE[3:2]

for bits[31:16] of the result.

It sets a pair of GE flags to 1 to indicate that the corresponding result is greater than or equal to zero. This is equivalent to an `ADDS` or `SUBS` instruction setting the N and V condition flags to the same value, so that the GE condition passes.

You can use these flags to control a following `SEL` instruction.

Note

GE[1:0] are set or cleared together, and GE[3:2] are set or cleared together.

Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

Related references

C2.100 SEL on page C2-257
C1.9 Condition code suffixes on page C1-92

C2.97 SBC

Subtract with Carry.

Syntax

`SBC{S}{cond} {Rd}, Rn, Operand2`

where:

`S`

is an optional suffix. If `S` is specified, the condition flags are updated on the result of the operation.

`cond`

is an optional condition code.

`Rd`

is the destination register.

`Rn`

is the register holding the first operand.

`Operand2`

is a flexible second operand.

Usage

The `SBC` (Subtract with Carry) instruction subtracts the value of *`Operand2`* from the value in *`Rn`*. If the carry flag is clear, the result is reduced by one.

You can use `SBC` to synthesize multiword arithmetic.

In certain circumstances, the assembler can substitute one instruction for another. Be aware of this when reading disassembly listings.

Use of PC and SP in T32 instructions

You cannot use PC (`R15`) for *`Rd`*, or any operand.

You cannot use SP (`R13`) for *`Rd`*, or any operand.

Use of PC and SP in A32 instructions

You cannot use PC for *`Rd`* or any operand in an `SBC` instruction that has a register-controlled shift.

Use of PC for any operand in instructions without register-controlled shift, is deprecated.

If you use PC (`R15`) as *`Rn`* or *`Rm`*, the value used is the address of the instruction plus 8.

If you use PC as *`Rd`*:

- Execution branches to the address corresponding to the result.
- If you use the `S` suffix, see the `SUBS pc,lr` instruction.

Use of SP and PC in `SBC` A32 instructions is deprecated.

Condition flags

If `S` is specified, the `SBC` instruction updates the N, Z, C and V flags according to the result.

16-bit instructions

The following forms of this instruction are available in T32 code, and are 16-bit instructions:

SBCS ***Rd, Rd, Rm***

Rd and *Rm* must both be Lo registers. This form can only be used outside an IT block.

SBC{*cond***}** ***Rd, Rd, Rm***

Rd and *Rm* must both be Lo registers. This form can only be used inside an IT block.

Multiword arithmetic examples

These instructions subtract one 96-bit integer contained in `R9`, `R10`, and `R11` from another 96-bit integer contained in `R6`, `R7`, and `R8`, and place the result in `R3`, `R4`, and `R5`:

```
SUBS    r3, r6, r9
SBCS    r4, r7, r10
SBC     r5, r8, r11
```

For clarity, the above examples use consecutive registers for multiword values. There is no requirement to do this. The following, for example, is perfectly valid:

```
SUBS    r6, r6, r9
SBCS    r9, r2, r1
SBC     r2, r8, r11
```

Related references

C2.3 Flexible second operand (Operand2) on page C2-112

C1.9 Condition code suffixes on page C1-92

C2.98 SBFX

Signed Bit Field Extract.

Syntax

`SBFX{cond} Rd, Rn, #lsb, #width`

where:

cond

is an optional condition code.

Rd

is the destination register.

Rn

is the source register.

lsb

is the bit number of the least significant bit in the bitfield, in the range 0 to 31.

width

is the width of the bitfield, in the range 1 to (32–*lsb*).

Operation

Copies adjacent bits from one register into the least significant bits of a second register, and sign extends to 32 bits.

Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

Condition flags

This instruction does not alter any flags.

Architectures

This 32-bit instruction is available in A32 and T32.

There is no 16-bit version of this instruction in T32.

Related references

C1.9 Condition code suffixes on page C1-92

C2.99 SDIV

Signed Divide.

Syntax

`SDIV{cond} {Rd}, Rn, Rm`

where:

cond
: is an optional condition code.

Rd
: is the destination register.

Rn
: is the register holding the value to be divided.

Rm
: is a register holding the divisor.

Register restrictions

PC or SP cannot be used for `Rd`, `Rn`, or `Rm`.

Architectures

This 32-bit T32 instruction is available in Armv7-R, Armv7-M, and Armv8-M Mainline.

This 32-bit A32 instruction is optional in Armv7-R.

This 32-bit A32 and T32 instruction is available in Armv7-A if Virtualization Extensions are implemented, and optional if not.

There is no 16-bit T32 `SDIV` instruction.

Related references

C1.9 Condition code suffixes on page C1-92

C2.100 SEL

Select bytes from each operand according to the state of the APSR GE flags.

Syntax

`SEL{cond} {Rd}, Rn, Rm`

where:

cond

is an optional condition code.

Rd

is the destination register.

Rn

is the register holding the first operand.

Rm

is the register holding the second operand.

Operation

The `SEL` instruction selects bytes from *Rn* or *Rm* according to the APSR GE flags:

- If GE[0] is set, `Rd[7:0]` come from `Rn[7:0]`, otherwise from `Rm[7:0]`.
- If GE[1] is set, `Rd[15:8]` come from `Rn[15:8]`, otherwise from `Rm[15:8]`.
- If GE[2] is set, `Rd[23:16]` come from `Rn[23:16]`, otherwise from `Rm[23:16]`.
- If GE[3] is set, `Rd[31:24]` come from `Rn[31:24]`, otherwise from `Rm[31:24]`.

Usage

Use the `SEL` instruction after one of the signed parallel instructions. You can use this to select maximum or minimum values in multiple byte or halfword data.

Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

Condition flags

This instruction does not change the flags.

Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

Examples

```
SEL     r0, r4, r5
SELLT   r4, r0, r4
```

The following instruction sequence sets each byte in `R4` equal to the unsigned minimum of the corresponding bytes of `R1` and `R2`:

```
USUB8    r4, r1, r2
SEL      r4, r2, r1
```

Related concepts

A1.12 Application Program Status Register on page A1-38

Related references

C1.9 Condition code suffixes on page C1-92

C2.101 SETEND

Set the endianness bit in the CPSR, without affecting any other bits in the CPSR.

——— **Note** ———

This instruction is deprecated in Armv8.

———————————

Syntax

`SETEND` *specifier*

where:

specifier

is one of:

BE

Big-endian.

LE

Little-endian.

Usage

Use `SETEND` to access data of different endianness, for example, to access several big-endian DMA-formatted data fields from an otherwise little-endian application.

`SETEND` cannot be conditional, and is not permitted in an IT block.

Architectures

This instruction is available in A32 and 16-bit T32.

This 16-bit instruction is available in T32, except in the Armv6-M and Armv7-M architectures.

There is no 32-bit version of this instruction in T32.

Example

```
SETEND  BE       ; Set the CPSR E bit for big-endian accesses
LDR     r0, [r2, #header]
LDR     r1, [r2, #CRC32]
SETEND  le       ; Set the CPSR E bit for little-endian accesses
                 ; for the rest of the application
```

C2.102 SETPAN

Set Privileged Access Never.

Syntax

```
SETPAN{q} #imm ; A1 general registers (A32)
SETPAN{q} #imm ; T1 general registers (T32)
```

Where:

q
Is an optional instruction width specifier. See *C2.2 Instruction width specifiers* on page C2-111.

imm
Is the unsigned immediate 0 or 1.

Architectures supported

Supported in Armv8.1 and later.

Usage

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.

Related references
C2.1 A32 and T32 instruction summary on page C2-106

C2.103 SEV

Set Event.

Syntax

`SEV{`*cond*`}`

where:

cond

is an optional condition code.

Operation

This is a hint instruction. It is optional whether it is implemented or not. If it is not implemented, it executes as a `NOP`. The assembler produces a diagnostic message if the instruction executes as a `NOP` on the target.

`SEV` causes an event to be signaled to all cores within a multiprocessor system. If `SEV` is implemented, `WFE` must also be implemented.

Availability

This instruction is available in A32 and T32.

Related references

C2.104 SEVL on page C2-262
C2.68 NOP on page C2-213
C1.9 Condition code suffixes on page C1-92

C2.104 SEVL

Set Event Locally.

Note

This instruction is supported only in Armv8.

Syntax

`SEVL{`*cond*`}`

where:

cond

is an optional condition code.

Operation

This is a hint instruction. It is optional whether it is implemented or not. If it is not implemented, it executes as a `NOP`. `armasm` produces a diagnostic message if the instruction executes as a `NOP` on the target.

`SEVL` causes an event to be signaled to all cores the current processor. `SEVL` is not required to affect other processors although it is permitted to do so.

Availability

This instruction is available in A32 and T32.

Related references

C2.103 SEV on page C2-261
C2.68 NOP on page C2-213
C1.9 Condition code suffixes on page C1-92

C2.105 SG

Secure Gateway.

Syntax

SG

Usage

Secure Gateway marks a valid branch target for branches from Non-secure code that wants to call Secure code.

C2.106 SHADD8

Signed halving parallel byte-wise addition.

Syntax

`SHADD8{cond} {Rd}, Rn, Rm`

where:

cond

is an optional condition code.

Rd

is the destination register.

Rm, Rn

are the general-purpose registers holding the operands.

Operation

This instruction performs four signed integer additions on the corresponding bytes of the operands, halves the results, and writes the results into the corresponding bytes of the destination. This cannot cause overflow.

Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

Condition flags

This instruction does not affect the N, Z, C, V, Q, or GE flags.

Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

Related references

C1.9 Condition code suffixes on page C1-92

C2.107 SHADD16

Signed halving parallel halfword-wise addition.

Syntax

`SHADD16{cond} {Rd}, Rn, Rm`

where:

cond

is an optional condition code.

Rd

is the destination register.

Rm, Rn

are the general-purpose registers holding the operands.

Operation

This instruction performs two signed integer additions on the corresponding halfwords of the operands, halves the results, and writes the results into the corresponding halfwords of the destination. This cannot cause overflow.

Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

Condition flags

This instruction does not affect the N, Z, C, V, Q, or GE flags.

Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

Related references

C1.9 Condition code suffixes on page C1-92

C2.108 SHASX

Signed halving parallel add and subtract halfwords with exchange.

Syntax

`SHASX{cond} {Rd}, Rn, Rm`

where:

cond

is an optional condition code.

Rd

is the destination register.

Rm, Rn

are the general-purpose registers holding the operands.

Operation

This instruction exchanges the two halfwords of the second operand, then performs an addition on the two top halfwords of the operands and a subtraction on the bottom two halfwords. It halves the results and writes them into the corresponding halfwords of the destination. This cannot cause overflow.

Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

Condition flags

This instruction does not affect the N, Z, C, V, Q, or GE flags.

Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

Related references

C1.9 Condition code suffixes on page C1-92

C2.109 SHSAX

Signed halving parallel subtract and add halfwords with exchange.

Syntax

`SHSAX{cond} {Rd}, Rn, Rm`

where:

cond

is an optional condition code.

Rd

is the destination register.

Rm, Rn

are the general-purpose registers holding the operands.

Operation

This instruction exchanges the two halfwords of the second operand, then performs a subtraction on the two top halfwords of the operands and an addition on the bottom two halfwords. It halves the results and writes them into the corresponding halfwords of the destination. This cannot cause overflow.

Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

Condition flags

This instruction does not affect the N, Z, C, V, Q, or GE flags.

Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

Related references

C1.9 Condition code suffixes on page C1-92

C2.110 SHSUB8

Signed halving parallel byte-wise subtraction.

Syntax

`SHSUB8{cond} {Rd}, Rn, Rm`

where:

cond

is an optional condition code.

Rd

is the destination register.

Rm, Rn

are the general-purpose registers holding the operands.

Operation

This instruction subtracts each byte of the second operand from the corresponding byte of the first operand, halves the results, and writes the results into the corresponding bytes of the destination. This cannot cause overflow.

Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

Condition flags

This instruction does not affect the N, Z, C, V, Q, or GE flags.

Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

Related references

C1.9 Condition code suffixes on page C1-92

C2.111 SHSUB16

Signed halving parallel halfword-wise subtraction.

Syntax

`SHSUB16{cond} {Rd}, Rn, Rm`

where:

cond

is an optional condition code.

Rd

is the destination register.

Rm, Rn

are the general-purpose registers holding the operands.

Operation

This instruction subtracts each halfword of the second operand from the corresponding halfword of the first operand, halves the results, and writes the results into the corresponding halfwords of the destination. This cannot cause overflow.

Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

Condition flags

This instruction does not affect the N, Z, C, V, Q, or GE flags.

Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

Related references

C1.9 Condition code suffixes on page C1-92

C2.112 SMC

Secure Monitor Call.

Syntax

`SMC{cond} #imm4`

where:

cond

is an optional condition code.

imm4

is a 4-bit immediate value. This is ignored by the Arm processor, but can be used by the SMC exception handler to determine what service is being requested.

Note

`SMC` was called `SMI` in earlier versions of the A32 assembly language. `SMI` instructions disassemble to `SMC`, with a comment to say that this was formerly `SMI`.

Architectures

This 32-bit instruction is available in A32 and T32, if the Arm architecture has the Security Extensions.

There is no 16-bit version of this instruction in T32.

Related references

C1.9 Condition code suffixes on page C1-92

Related information

Arm Architecture Reference Manual

C2.113 SMLAxy

Signed Multiply Accumulate, with 16-bit operands and a 32-bit result and accumulator.

Syntax

`SMLA<x><y>{cond} Rd, Rn, Rm, Ra`

where:

`<x>`

is either `B` or `T`. `B` means use the bottom half (bits [15:0]) of `Rn`, `T` means use the top half (bits [31:16]) of `Rn`.

`<y>`

is either `B` or `T`. `B` means use the bottom half (bits [15:0]) of `Rm`, `T` means use the top half (bits [31:16]) of `Rm`.

`cond`

is an optional condition code.

`Rd`

is the destination register.

`Rn, Rm`

are the registers holding the values to be multiplied.

`Ra`

is the register holding the value to be added.

Operation

`SMLAxy` multiplies the 16-bit signed integers from the selected halves of `Rn` and `Rm`, adds the 32-bit result to the 32-bit value in `Ra`, and places the result in `Rd`.

Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

Condition flags

This instruction does not affect the N, Z, C, or V flags.

If overflow occurs in the accumulation, `SMLAxy` sets the Q flag. To read the state of the Q flag, use an `MRS` instruction.

Note

`SMLAxy` never clears the Q flag. To clear the Q flag, use an `MSR` instruction.

Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

Examples

```
SMLABBNE    r0, r2, r1, r10
SMLABT      r0, r0, r3, r5
```

Related references

C2.62 MRS (PSR to general-purpose register) on page C2-204
C2.65 MSR (general-purpose register to PSR) on page C2-208
C1.9 Condition code suffixes on page C1-92

C2.114 SMLAD

Dual 16-bit Signed Multiply with Addition of products and 32-bit accumulation.

Syntax

`SMLAD{X}{cond} Rd, Rn, Rm, Ra`

where:

cond

is an optional condition code.

X

is an optional parameter. If `X` is present, the most and least significant halfwords of the second operand are exchanged before the multiplications occur.

Rd

is the destination register.

Rn, Rm

are the registers holding the operands.

Ra

is the register holding the accumulate operand.

Operation

`SMLAD` multiplies the bottom halfword of *Rn* with the bottom halfword of *Rm*, and the top halfword of *Rn* with the top halfword of *Rm*. It then adds both products to the value in *Ra* and stores the sum to *Rd*.

Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

Condition flags

This instruction does not change the flags.

Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

Example

```
        SMLADLT     r1, r2, r4, r1
```

Related references

C1.9 Condition code suffixes on page C1-92

C2.115 SMLAL

Signed Long Multiply, with optional Accumulate, with 32-bit operands, and 64-bit result and accumulator.

Syntax

`SMLAL{S}{cond} RdLo, RdHi, Rn, Rm`

where:

`S`

is an optional suffix available in A32 state only. If `S` is specified, the condition flags are updated on the result of the operation.

`cond`

is an optional condition code.

`RdLo, RdHi`

are the destination registers. They also hold the accumulating value. *`RdLo`* and *`RdHi`* must be different registers

`Rn, Rm`

are general-purpose registers holding the operands.

Operation

The `SMLAL` instruction interprets the values from *`Rn`* and *`Rm`* as two's complement signed integers. It multiplies these integers, and adds the 64-bit result to the 64-bit signed integer contained in *`RdHi`* and *`RdLo`*.

Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

Condition flags

If `S` is specified, this instruction:

- Updates the N and Z flags according to the result.
- Does not affect the C or V flags.

Architectures

This instruction is available in A32 and T32.

There is no 16-bit version of this instruction in T32.

Related references

C1.9 Condition code suffixes on page C1-92

C2.116 SMLALD

Dual 16-bit Signed Multiply with Addition of products and 64-bit Accumulation.

Syntax

`SMLALD{X}{cond} RdLo, RdHi, Rn, Rm`

where:

X

is an optional parameter. If `X` is present, the most and least significant halfwords of the second operand are exchanged before the multiplications occur.

cond

is an optional condition code.

RdLo, RdHi

are the destination registers for the 64-bit result. They also hold the 64-bit accumulate operand. *RdHi* and *RdLo* must be different registers.

Rn, Rm

are the general-purpose registers holding the operands.

Operation

`SMLALD` multiplies the bottom halfword of *Rn* with the bottom halfword of *Rm*, and the top halfword of *Rn* with the top halfword of *Rm*. It then adds both products to the value in *RdLo*, *RdHi* and stores the sum to *RdLo*, *RdHi*.

Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

Condition flags

This instruction does not change the flags.

Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

Example

```
    SMLALD      r10, r11, r5, r1
```

Related references

C1.9 Condition code suffixes on page C1-92

C2.117 SMLALxy

Signed Multiply-Accumulate with 16-bit operands and a 64-bit accumulator.

Syntax

`SMLAL<x><y>{cond} RdLo, RdHi, Rn, Rm`

where:

`<x>`

is either `B` or `T`. `B` means use the bottom half (bits [15:0]) of `Rn`, `T` means use the top half (bits [31:16]) of `Rn`.

`<y>`

is either `B` or `T`. `B` means use the bottom half (bits [15:0]) of `Rm`, `T` means use the top half (bits [31:16]) of `Rm`.

`cond`

is an optional condition code.

`RdLo, RdHi`

are the destination registers. They also hold the accumulate value. `RdHi` and `RdLo` must be different registers.

`Rn, Rm`

are the general-purpose registers holding the values to be multiplied.

Operation

`SMLALxy` multiplies the signed integer from the selected half of `Rm` by the signed integer from the selected half of `Rn`, and adds the 32-bit result to the 64-bit value in `RdHi` and `RdLo`.

Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

Condition flags

This instruction does not change the flags.

Note

`SMLALxy` cannot raise an exception. If overflow occurs on this instruction, the result wraps round without any warning.

Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

Examples

```
SMLALTB     r2, r3, r7, r1
SMLALBTVS   r0, r1, r9, r2
```

Related references

C1.9 Condition code suffixes on page C1-92

C2.118 SMLAWy

Signed Multiply-Accumulate Wide, with one 32-bit and one 16-bit operand, and a 32-bit accumulate value, providing the top 32 bits of the result.

Syntax

`SMLAW<y>{cond} Rd, Rn, Rm, Ra`

where:

`<y>`

is either `B` or `T`. `B` means use the bottom half (bits [15:0]) of `Rm`, `T` means use the top half (bits [31:16]) of `Rm`.

`cond`

is an optional condition code.

`Rd`

is the destination register.

`Rn, Rm`

are the registers holding the values to be multiplied.

`Ra`

is the register holding the value to be added.

Operation

`SMLAWy` multiplies the signed 16-bit integer from the selected half of `Rm` by the signed 32-bit integer from `Rn`, adds the top 32 bits of the 48-bit result to the 32-bit value in `Ra`, and places the result in `Rd`.

Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

Condition flags

This instruction does not affect the N, Z, C, or V flags.

If overflow occurs in the accumulation, `SMLAWy` sets the Q flag.

Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

Related references

C2.62 MRS (PSR to general-purpose register) on page C2-204
C1.9 Condition code suffixes on page C1-92

C2.119 SMLSD

Dual 16-bit Signed Multiply with Subtraction of products and 32-bit accumulation.

Syntax

`SMLSD{X}{cond} Rd, Rn, Rm, Ra`

where:

cond

is an optional condition code.

X

is an optional parameter. If `X` is present, the most and least significant halfwords of the second operand are exchanged before the multiplications occur.

Rd

is the destination register.

Rn, Rm

are the registers holding the operands.

Ra

is the register holding the accumulate operand.

Operation

`SMLSD` multiplies the bottom halfword of *Rn* with the bottom halfword of *Rm*, and the top halfword of *Rn* with the top halfword of *Rm*. It then subtracts the second product from the first, adds the difference to the value in *Ra*, and stores the result to *Rd*.

Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

Condition flags

This instruction does not change the flags.

Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

Examples

```
SMLSD       r1, r2, r0, r7
SMLSDX      r11, r10, r2, r3
```

Related references

C1.9 Condition code suffixes on page C1-92

C2.120 SMLSLD

Dual 16-bit Signed Multiply with Subtraction of products and 64-bit accumulation.

Syntax

`SMLSD{X}{cond} RdLo, RdHi, Rn, Rm`

where:

X

is an optional parameter. If `X` is present, the most and least significant halfwords of the second operand are exchanged before the multiplications occur.

cond

is an optional condition code.

RdLo, RdHi

are the destination registers for the 64-bit result. They also hold the 64-bit accumulate operand. *RdHi* and *RdLo* must be different registers.

Rn, Rm

are the general-purpose registers holding the operands.

Operation

`SMLSLD` multiplies the bottom halfword of *Rn* with the bottom halfword of *Rm*, and the top halfword of *Rn* with the top halfword of *Rm*. It then subtracts the second product from the first, adds the difference to the value in *RdLo*, *RdHi*, and stores the result to *RdLo*, *RdHi*.

Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

Condition flags

This instruction does not change the flags.

Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

Example

```
    SMLSLD      r3, r0, r5, r1
```

Related references

C1.9 Condition code suffixes on page C1-92

C2.121 SMMLA

Signed Most significant word Multiply with Accumulation.

Syntax

`SMMLA{R}{cond} Rd, Rn, Rm, Ra`

where:

R

is an optional parameter. If `R` is present, the result is rounded, otherwise it is truncated.

cond

is an optional condition code.

Rd

is the destination register.

Rn, Rm

are the registers holding the operands.

Ra

is a register holding the value to be added or subtracted from.

Operation

`SMMLA` multiplies the values from *Rn* and *Rm*, adds the value in *Ra* to the most significant 32 bits of the product, and stores the result in *Rd*.

If the optional `R` parameter is specified, `0x80000000` is added before extracting the most significant 32 bits. This has the effect of rounding the result.

Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

Condition flags

This instruction does not change the flags.

Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

Related references

C1.9 Condition code suffixes on page C1-92

C2.122 SMMLS

Signed Most significant word Multiply with Subtraction.

Syntax

`SMMLS{R}{cond} Rd, Rn, Rm, Ra`

where:

R

is an optional parameter. If `R` is present, the result is rounded, otherwise it is truncated.

cond

is an optional condition code.

Rd

is the destination register.

Rn, Rm

are the registers holding the operands.

Ra

is a register holding the value to be added or subtracted from.

Operation

`SMMLS` multiplies the values from *Rn* and *Rm*, subtracts the product from the value in *Ra* shifted left by 32 bits, and stores the most significant 32 bits of the result in *Rd*.

If the optional `R` parameter is specified, `0x80000000` is added before extracting the most significant 32 bits. This has the effect of rounding the result.

Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

Condition flags

This instruction does not change the flags.

Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

Related references

C1.9 Condition code suffixes on page C1-92

C2.123 SMMUL

Signed Most significant word Multiply.

Syntax

`SMMUL{R}{cond} {Rd}, Rn, Rm`

where:

R

is an optional parameter. If `R` is present, the result is rounded, otherwise it is truncated.

cond

is an optional condition code.

Rd

is the destination register.

Rn, Rm

are the registers holding the operands.

Ra

is a register holding the value to be added or subtracted from.

Operation

`SMMUL` multiplies the 32-bit values from *Rn* and *Rm*, and stores the most significant 32 bits of the 64-bit result to *Rd*.

If the optional `R` parameter is specified, `0x80000000` is added before extracting the most significant 32 bits. This has the effect of rounding the result.

Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

Condition flags

This instruction does not change the flags.

Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

Examples

```
    SMMULGE     r6, r4, r3
    SMMULR      r2, r2, r2
```

Related references

C1.9 Condition code suffixes on page C1-92

C2.124 SMUAD

Dual 16-bit Signed Multiply with Addition of products, and optional exchange of operand halves.

Syntax

`SMUAD{X}{cond} {Rd}, Rn, Rm`

where:

X

is an optional parameter. If `X` is present, the most and least significant halfwords of the second operand are exchanged before the multiplications occur.

cond

is an optional condition code.

Rd

is the destination register.

Rn, Rm

are the registers holding the operands.

Operation

`SMUAD` multiplies the bottom halfword of *Rn* with the bottom halfword of *Rm*, and the top halfword of *Rn* with the top halfword of *Rm*. It then adds the products and stores the sum to *Rd*.

Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

Q flag

The `SMUAD` instruction sets the Q flag if the addition overflows.

Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

Examples

```
SMUAD       r2, r3, r2
```

Related references

C1.9 Condition code suffixes on page C1-92

C2.125 SMULxy

Signed Multiply, with 16-bit operands and a 32-bit result.

Syntax

`SMUL<x><y>{cond} {Rd}, Rn, Rm`

where:

`<x>`

is either `B` or `T`. `B` means use the bottom half (bits [15:0]) of `Rn`, `T` means use the top half (bits [31:16]) of `Rn`.

`<y>`

is either `B` or `T`. `B` means use the bottom half (bits [15:0]) of `Rm`, `T` means use the top half (bits [31:16]) of `Rm`.

`cond`

is an optional condition code.

`Rd`

is the destination register.

`Rn, Rm`

are the registers holding the values to be multiplied.

Operation

`SMULxy` multiplies the 16-bit signed integers from the selected halves of `Rn` and `Rm`, and places the 32-bit result in `Rd`.

Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

Condition flags

These instructions do not affect the N, Z, C, or V flags.

Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

Examples

```
    SMULTBEQ    r8, r7, r9
```

Related references

C2.62 MRS (PSR to general-purpose register) on page C2-204
C2.65 MSR (general-purpose register to PSR) on page C2-208
C1.9 Condition code suffixes on page C1-92

C2.126 SMULL

Signed Long Multiply, with 32-bit operands and 64-bit result.

Syntax

`SMULL{S}{cond} RdLo, RdHi, Rn, Rm`

where:

`S`

is an optional suffix available in A32 state only. If `S` is specified, the condition flags are updated on the result of the operation.

`cond`

is an optional condition code.

`RdLo, RdHi`

are the destination registers. *`RdLo`* and *`RdHi`* must be different registers

`Rn, Rm`

are general-purpose registers holding the operands.

Operation

The `SMULL` instruction interprets the values from *`Rn`* and *`Rm`* as two's complement signed integers. It multiplies these integers and places the least significant 32 bits of the result in *`RdLo`*, and the most significant 32 bits of the result in *`RdHi`*.

Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

Condition flags

If `S` is specified, this instruction:

- Updates the N and Z flags according to the result.
- Does not affect the C or V flags.

Architectures

This instruction is available in A32 and T32.

There is no 16-bit version of this instruction in T32.

Related references

C1.9 Condition code suffixes on page C1-92

C2.127 SMULWy

Signed Multiply Wide, with one 32-bit and one 16-bit operand, providing the top 32 bits of the result.

Syntax

`SMULW<y>{cond} {Rd}, Rn, Rm`

where:

`<y>`

is either `B` or `T`. `B` means use the bottom half (bits [15:0]) of `Rm`, `T` means use the top half (bits [31:16]) of `Rm`.

`cond`

is an optional condition code.

`Rd`

is the destination register.

`Rn, Rm`

are the registers holding the values to be multiplied.

Operation

`SMULWy` multiplies the signed integer from the selected half of `Rm` by the signed integer from `Rn`, and places the upper 32-bits of the 48-bit result in `Rd`.

Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

Condition flags

This instruction does not affect the N, Z, C, or V flags.

Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

Related references

C2.62 MRS (PSR to general-purpose register) on page C2-204
C1.9 Condition code suffixes on page C1-92

C2.128 SMUSD

Dual 16-bit Signed Multiply with Subtraction of products, and optional exchange of operand halves.

Syntax

`SMUSD{X}{cond} {Rd}, Rn, Rm`

where:

X

is an optional parameter. If `X` is present, the most and least significant halfwords of the second operand are exchanged before the multiplications occur.

cond

is an optional condition code.

Rd

is the destination register.

Rn, Rm

are the registers holding the operands.

Operation

`SMUSD` multiplies the bottom halfword of *Rn* with the bottom halfword of *Rm*, and the top halfword of *Rn* with the top halfword of *Rm*. It then subtracts the second product from the first, and stores the difference to *Rd*.

Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

Example

```
    SMUSDXNE    r0, r1, r2
```

Related references

C1.9 Condition code suffixes on page C1-92

C2.129 SRS

Store Return State onto a stack.

Syntax

`SRS{`*`addr_mode`*`}{`*`cond`*`} sp{!}, #`*`modenum`*

`SRS{`*`addr_mode`*`}{`*`cond`*`} #`*`modenum`*`{!} ; This is pre-UAL syntax`

where:

`addr_mode`

is any one of the following:

`IA`

Increment address After each transfer

`IB`

Increment address Before each transfer (A32 only)

`DA`

Decrement address After each transfer (A32 only)

`DB`

Decrement address Before each transfer (Full Descending stack).

If *`addr_mode`* is omitted, it defaults to Increment After. You can also use stack oriented addressing mode suffixes, for example, when implementing stacks.

`cond`

is an optional condition code.

Note

`cond` is permitted only in T32 code, using a preceding `IT` instruction, but this is deprecated in the Armv8 architecture. This is an unconditional instruction in A32.

`!`

is an optional suffix. If `!` is present, the final address is written back into the SP of the mode specified by *`modenum`*.

`modenum`

specifies the number of the mode whose banked SP is used as the base register. You must use only the defined mode numbers.

Operation

`SRS` stores the LR and the SPSR of the current mode, at the address contained in SP of the mode specified by *`modenum`*, and the following word respectively. Optionally updates SP of the mode specified by *`modenum`*. This is compatible with the normal use of the `STM` instruction for stack accesses.

Note

For full descending stack, you must use `SRSFD` or `SRSDB`.

Usage

You can use `SRS` to store return state for an exception handler on a different stack from the one automatically selected.

Notes

Where addresses are not word-aligned, `SRS` ignores the least significant two bits of the specified address.

The time order of the accesses to individual words of memory generated by `SRS` is not architecturally defined. Do not use this instruction on memory-mapped I/O locations where access order matters.

Do not use `SRS` in User and System modes because these modes do not have a SPSR.

`SRS` is not permitted in a non-secure state if *`modenum`* specifies monitor mode.

Availability

This 32-bit instruction is available in A32 and T32.

The 32-bit T32 instruction is not available in the Armv7-M architecture.

There is no 16-bit version of this instruction in T32.

Example

```
R13_usr  EQU     16
         SRSFD   sp,#R13_usr
```

Related concepts

A1.3 Processor modes, and privileged and unprivileged software execution on page A1-28

Related references

C2.45 LDM on page C2-177

C1.9 Condition code suffixes on page C1-92

C2.130 SSAT

Signed Saturate to any bit position, with optional shift before saturating.

Syntax

`SSAT{cond} Rd, #sat, Rm{, shift}`

where:

cond

is an optional condition code.

Rd

is the destination register.

sat

specifies the bit position to saturate to, in the range 1 to 32.

Rm

is the register containing the operand.

shift

is an optional shift. It must be one of the following:

ASR #*n*

where *n* is in the range 1-32 (A32) or 1-31 (T32)

LSL #*n*

where *n* is in the range 0-31.

Operation

The `SSAT` instruction applies the specified shift, then saturates a signed value to the signed range $-2^{sat-1} \leq x \leq 2^{sat-1} - 1$.

Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

Q flag

If saturation occurs, this instruction sets the Q flag. To read the state of the Q flag, use an `MRS` instruction.

Architectures

This instruction is available in A32 and T32.

There is no 16-bit version of this instruction in T32.

Example

```
    SSAT    r7, #16, r7, LSL #4
```

Related references

C2.131 SSAT16 on page C2-292
C2.62 MRS (PSR to general-purpose register) on page C2-204
C1.9 Condition code suffixes on page C1-92

C2.131 SSAT16

Parallel halfword Saturate.

Syntax

`SSAT16{cond} Rd, #sat, Rn`

where:

cond

is an optional condition code.

Rd

is the destination register.

sat

specifies the bit position to saturate to, in the range 1 to 16.

Rn

is the register holding the operand.

Operation

Halfword-wise signed saturation to any bit position.

The `SSAT16` instruction saturates each signed halfword to the signed range $-2^{sat-1} \leq x \leq 2^{sat-1} - 1$.

Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

Q flag

If saturation occurs on either halfword, this instruction sets the Q flag. To read the state of the Q flag, use an `MRS` instruction.

Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

Correct example

```
SSAT16  r7, #12, r7
```

Incorrect example

```
SSAT16  r1, #16, r2, LSL #4 ; shifts not permitted with halfword
                            ; saturations
```

Related references

C2.62 MRS (PSR to general-purpose register) on page C2-204
C1.9 Condition code suffixes on page C1-92

C2.132 SSAX

Signed parallel subtract and add halfwords with exchange.

Syntax

`SSAX{cond} {Rd}, Rn, Rm`

where:

cond

is an optional condition code.

Rd

is the destination register.

Rm, Rn

are the general-purpose registers holding the operands.

Operation

This instruction exchanges the two halfwords of the second operand, then performs a subtraction on the two top halfwords of the operands and an addition on the bottom two halfwords. It writes the results into the corresponding halfwords of the destination. The results are modulo 2^{16}. It sets the APSR GE flags.

Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

GE flags

This instruction does not affect the N, Z, C, V, or Q flags.

It sets the GE flags in the APSR as follows:

GE[1:0]

for bits[15:0] of the result.

GE[3:2]

for bits[31:16] of the result.

It sets a pair of GE flags to 1 to indicate that the corresponding result is greater than or equal to zero. This is equivalent to an `ADDS` or `SUBS` instruction setting the N and V condition flags to the same value, so that the GE condition passes.

You can use these flags to control a following `SEL` instruction.

Note

GE[1:0] are set or cleared together, and GE[3:2] are set or cleared together.

Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

Related references

C2.100 SEL on page C2-257
C1.9 Condition code suffixes on page C1-92

C2.133 SSUB8

Signed parallel byte-wise subtraction.

Syntax

`SSUB8{cond} {Rd}, Rn, Rm`

where:

cond

is an optional condition code.

Rd

is the destination register.

Rm, Rn

are the general-purpose registers holding the operands.

Operation

This instruction subtracts each byte of the second operand from the corresponding byte of the first operand and writes the results into the corresponding bytes of the destination. The results are modulo 2^8. It sets the APSR GE flags.

Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

GE flags

This instruction does not affect the N, Z, C, V, or Q flags.

It sets the GE flags in the APSR as follows:

GE[0]

for bits[7:0] of the result.

GE[1]

for bits[15:8] of the result.

GE[2]

for bits[23:16] of the result.

GE[3]

for bits[31:24] of the result.

It sets a GE flag to 1 to indicate that the corresponding result is greater than or equal to zero. This is equivalent to a `SUBS` instruction setting the N and V condition flags to the same value, so that the GE condition passes.

You can use these flags to control a following `SEL` instruction.

Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

Related references

C2.100 SEL on page C2-257

C1.9 Condition code suffixes on page C1-92

C2.134 SSUB16

Signed parallel halfword-wise subtraction.

Syntax

`SSUB16{cond} {Rd}, Rn, Rm`

where:

cond

is an optional condition code.

Rd

is the destination register.

Rm, Rn

are the general-purpose registers holding the operands.

Operation

This instruction subtracts each halfword of the second operand from the corresponding halfword of the first operand and writes the results into the corresponding halfwords of the destination. The results are modulo 2^{16}. It sets the APSR GE flags.

Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

GE flags

This instruction does not affect the N, Z, C, V, or Q flags.

It sets the GE flags in the APSR as follows:

GE[1:0]

for bits[15:0] of the result.

GE[3:2]

for bits[31:16] of the result.

It sets a pair of GE flags to 1 to indicate that the corresponding result is greater than or equal to zero. This is equivalent to a `SUBS` instruction setting the N and V condition flags to the same value, so that the GE condition passes.

You can use these flags to control a following `SEL` instruction.

——— **Note** ———

GE[1:0] are set or cleared together, and GE[3:2] are set or cleared together.

———————————

Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

Related references

C2.100 SEL on page C2-257
C1.9 Condition code suffixes on page C1-92

C2.135 STC and STC2

Transfer Data between memory and Coprocessor.

——— **Note** ———

STC2 is not supported in Armv8.

———————————

Syntax

op{L}{*cond*} *coproc*, *CRd*, [*Rn*]

op{L}{*cond*} *coproc*, *CRd*, [*Rn*, #{-}*offset*] ; offset addressing

op{L}{*cond*} *coproc*, *CRd*, [*Rn*, #{-}*offset*]! ; pre-index addressing

op{L}{*cond*} *coproc*, *CRd*, [*Rn*], #{-}*offset* ; post-index addressing

op{L}{*cond*} *coproc*, *CRd*, [*Rn*], {*option*}

where:

op

is one of STC or STC2.

cond

is an optional condition code.

In A32 code, *cond* is not permitted for STC2.

L

is an optional suffix specifying a long transfer.

coproc

is the name of the coprocessor the instruction is for. The standard name is p*n*, where *n* is an integer whose value must be:

- In the range 0-15 in Armv7 and earlier.
- 14 in Armv8.

CRd

is the coprocessor register to store.

Rn

is the register on which the memory address is based. If PC is specified, the value used is the address of the current instruction plus eight.

-

is an optional minus sign. If - is present, the offset is subtracted from *Rn*. Otherwise, the offset is added to *Rn*.

offset

is an expression evaluating to a multiple of 4, in the range 0 to 1020.

!

is an optional suffix. If ! is present, the address including the offset is written back into *Rn*.

option

is a coprocessor option in the range 0-255, enclosed in braces.

Usage

The use of these instructions depends on the coprocessor. See the coprocessor documentation for details.

Architectures

These 32-bit instructions are available in A32 and T32.

There are no 16-bit versions of these instructions in T32.

Register restrictions

You cannot use PC for `Rn` in the pre-index and post-index instructions. These are the forms that write back to `Rn`.

You cannot use PC for `Rn` in T32 `STC` and `STC2` instructions.

A32 `STC` and `STC2` instructions where `Rn` is PC, are deprecated.

Related references

C1.9 Condition code suffixes on page C1-92

C2.136 STL

Store-Release Register.

Note

This instruction is supported only in Armv8.

Syntax

`STL{cond} Rt, [Rn]`

`STLB{cond} Rt, [Rn]`

`STLH{cond} Rt, [Rn]`

where:

cond
: is an optional condition code.

Rt
: is the register to store.

Rn
: is the register on which the memory address is based.

Operation

`STL` stores data to memory. If any loads or stores appear before a store-release in program order, then all observers are guaranteed to observe the loads and stores before observing the store-release. Loads and stores appearing after a store-release are unaffected.

If a store-release follows a load-acquire, each observer is guaranteed to observe them in program order.

There is no requirement that a store-release be paired with a load-acquire.

All store-release operations are multi-copy atomic, meaning that in a multiprocessing system, if one observer observes a write to memory because of a store-release operation, then all observers observe it. Also, all observers observe all such writes to the same location in the same order.

Restrictions

The address specified must be naturally aligned, or an alignment fault is generated.

The PC must not be used for *Rt* or *Rn*.

Availability

This 32-bit instruction is available in A32 and T32.

There is no 16-bit version of this instruction.

Related references

C2.43 LDAEX on page C2-173
C2.42 LDA on page C2-172
C2.137 STLEX on page C2-302
C1.9 Condition code suffixes on page C1-92

C2.137 STLEX

Store-Release Register Exclusive.

Note

This instruction is supported only in Armv8.

Syntax

`STLEX{cond} Rd, Rt, [Rn]`

`STLEXB{cond} Rd, Rt, [Rn]`

`STLEXH{cond} Rd, Rt, [Rn]`

`STLEXD{cond} Rd, Rt, Rt2, [Rn]`

where:

cond
: is an optional condition code.

Rd
: is the destination register for the returned status.

Rt
: is the register to load or store.

Rt2
: is the second register for doubleword loads or stores.

Rn
: is the register on which the memory address is based.

Operation

`STLEX` performs a conditional store to memory. The conditions are as follows:

- If the physical address does not have the Shared TLB attribute, and the executing processor has an outstanding tagged physical address, the store takes place, the tag is cleared, and the value 0 is returned in *Rd*.
- If the physical address does not have the Shared TLB attribute, and the executing processor does not have an outstanding tagged physical address, the store does not take place, and the value 1 is returned in *Rd*.
- If the physical address has the Shared TLB attribute, and the physical address is tagged as exclusive access for the executing processor, the store takes place, the tag is cleared, and the value 0 is returned in *Rd*.
- If the physical address has the Shared TLB attribute, and the physical address is not tagged as exclusive access for the executing processor, the store does not take place, and the value 1 is returned in *Rd*.

If any loads or stores appear before `STLEX` in program order, then all observers are guaranteed to observe the loads and stores before observing the store-release. Loads and stores appearing after `STLEX` are unaffected.

All store-release operations are multi-copy atomic.

Restrictions

The PC must not be used for any of *Rd*, *Rt*, *Rt2*, or *Rn*.

For `STLEX`, *Rd* must not be the same register as *Rt*, *Rt2*, or *Rn*.

For A32 instructions:

- SP can be used but use of SP for any of *Rd*, *Rt*, or *Rt2* is deprecated.
- For STLEXD, *Rt* must be an even numbered register, and not LR.
- *Rt2* must be *R(t+1)*.

For T32 instructions, SP can be used for R*n*, but must not be used for any of *Rd*, *Rt*, or *Rt2*.

Usage

Use LDAEX and STLEX to implement interprocess communication in multiple-processor and shared-memory systems.

For reasons of performance, keep the number of instructions between corresponding LDAEX and STLEX instructions to a minimum.

Note

The address used in a STLEX instruction must be the same as the address in the most recently executed LDAEX instruction.

Availability

These 32-bit instructions are available in A32 and T32.

There are no 16-bit versions of these instructions.

Related references

C2.43 LDAEX on page C2-173
C2.136 STL on page C2-301
C2.42 LDA on page C2-172
C1.9 Condition code suffixes on page C1-92

C2.138 STM

Store Multiple registers.

Syntax

`STM{addr_mode}{cond} Rn{!}, reglist{^}`

where:

addr_mode

is any one of the following:

IA

Increment address After each transfer. This is the default, and can be omitted.

IB

Increment address Before each transfer (A32 only).

DA

Decrement address After each transfer (A32 only).

DB

Decrement address Before each transfer.

You can also use the stack-oriented addressing mode suffixes, for example when implementing stacks.

cond

is an optional condition code.

Rn

is the *base register*, the general-purpose register holding the initial address for the transfer. *Rn* must not be PC.

!

is an optional suffix. If ! is present, the final address is written back into *Rn*.

reglist

is a list of one or more registers to be stored, enclosed in braces. It can contain register ranges. It must be comma-separated if it contains more than one register or register range. Any combination of registers R0 to R15 (PC) can be transferred in A32 state, but there are some restrictions in T32 state.

^

is an optional suffix, available in A32 state only. You must not use it in User mode or System mode. Data is transferred into or out of the User mode registers instead of the current mode registers.

Restrictions on reglist in 32-bit T32 instructions

In 32-bit T32 instructions:

- The SP cannot be in the list.
- The PC cannot be in the list.
- There must be two or more registers in the list.

If you write an STM instruction with only one register in *reglist*, the assembler automatically substitutes the equivalent STR instruction. Be aware of this when comparing disassembly listings with source code.

Restrictions on reglist in A32 instructions

A32 store instructions can have SP and PC in the *reglist* but these instructions that include SP or PC in the *reglist* are deprecated.

16-bit instruction

A 16-bit version of this instruction is available in T32 code.

The following restrictions apply to the 16-bit instruction:

- All registers in *reglist* must be Lo registers.
- *Rn* must be a Lo register.
- *addr_mode* must be omitted (or IA), meaning increment address after each transfer.
- Writeback must be specified for STM instructions.

Note

16-bit T32 STM instructions with writeback that specify *Rn* as the lowest register in the *reglist* are deprecated.

In addition, the PUSH and POP instructions are subsets of the STM and LDM instructions and can therefore be expressed using the STM and LDM instructions. Some forms of PUSH and POP are also 16-bit instructions.

Storing the base register, with writeback

In A32 or 16-bit T32 instructions, if *Rn* is in *reglist*, and writeback is specified with the ! suffix:

- If the instruction is STM{*addr_mode*}{*cond*} and *Rn* is the lowest-numbered register in *reglist*, the initial value of *Rn* is stored. These instructions are deprecated.
- Otherwise, the stored value of R*n* cannot be relied on, so these instructions are not permitted.

32-bit T32 instructions are not permitted if *Rn* is in *reglist*, and writeback is specified with the ! suffix.

Correct example

```
STMDB   r1!,{r3-r6,r11,r12}
```

Incorrect example

```
STM     r5!,{r5,r4,r9} ; value stored for R5 unknown
```

Related references

C2.73 POP on page C2-221
C1.9 Condition code suffixes on page C1-92

C2.139 STR (immediate offset)

Store with immediate offset, pre-indexed immediate offset, or post-indexed immediate offset.

Syntax

```
STR{type}{cond} Rt, [Rn {, #offset}] ; immediate offset
STR{type}{cond} Rt, [Rn, #offset]! ; pre-indexed
STR{type}{cond} Rt, [Rn], #offset ; post-indexed
STRD{cond} Rt, Rt2, [Rn {, #offset}] ; immediate offset, doubleword
STRD{cond} Rt, Rt2, [Rn, #offset]! ; pre-indexed, doubleword
STRD{cond} Rt, Rt2, [Rn], #offset ; post-indexed, doubleword
```

where:

type

can be any one of:

B

Byte

H

Halfword

-

omitted, for Word.

cond

is an optional condition code.

Rt

is the general-purpose register to store.

Rn

is the general-purpose register on which the memory address is based.

offset

is an offset. If *offset* is omitted, the address is the contents of *Rn*.

Rt2

is the additional register to store for doubleword operations.

Not all options are available in every instruction set and architecture.

Offset ranges and architectures

The following table shows the ranges of offsets and availability of this instruction:

Table C2-15 Offsets and architectures, STR, word, halfword, and byte

Instruction	Immediate offset	Pre-indexed	Post-indexed
A32, word or byte	-4095 to 4095	-4095 to 4095	-4095 to 4095
A32, halfword	-255 to 255	-255 to 255	-255 to 255

Table C2-15 Offsets and architectures, STR, word, halfword, and byte (continued)

Instruction	Immediate offset	Pre-indexed	Post-indexed
A32, doubleword	-255 to 255	-255 to 255	-255 to 255
T32 32-bit encoding, word, halfword, or byte	-255 to 4095	-255 to 255	-255 to 255
T32 32-bit encoding, doubleword	-1020 to 1020 [z]	-1020 to 1020 [z]	-1020 to 1020 [z]
T32 16-bit encoding, word [aa]	0 to 124 [z]	Not available	Not available
T32 16-bit encoding, halfword [aa]	0 to 62 [ac]	Not available	Not available
T32 16-bit encoding, byte [aa]	0 to 31	Not available	Not available
T32 16-bit encoding, word, Rn is SP [ab]	0 to 1020 [z]	Not available	Not available

Register restrictions

R*n* must be different from R*t* in the pre-index and post-index forms.

Doubleword register restrictions

R*n* must be different from R*t2* in the pre-index and post-index forms.

For T32 instructions, you must not specify SP or PC for either *Rt* or *Rt2*.

For A32 instructions:

- *Rt* must be an even-numbered register.
- *Rt* must not be LR.
- Arm strongly recommends that you do not use R12 for *Rt*.
- *Rt2* must be R(*t* + 1).

Use of PC

In A32 instructions you can use PC for R*t* in STR word instructions and PC for R*n* in STR instructions with immediate offset syntax (that is the forms that do not writeback to the R*n*). However, this is deprecated.

Other uses of PC are not permitted in these A32 instructions.

In T32 code, using PC in STR instructions is not permitted.

Use of SP

You can use SP for R*n*.

In A32 code, you can use SP for R*t* in word instructions. You can use SP for R*t* in non-word instructions in A32 code but this is deprecated.

In T32 code, you can use SP for R*t* in word instructions only. All other use of SP for R*t* in this instruction is not permitted in T32 code.

Example

```
    STR     r2,[r9,#consta-struc]   ; consta-struc is an expression
                                    ; evaluating to a constant in
                                    ; the range 0-4095.
```

Related references

C1.9 Condition code suffixes on page C1-92

[z] Must be divisible by 4.
[aa] Rt and Rn must be in the range R0-R7.
[ab] Rt must be in the range R0-R7.
[ac] Must be divisible by 2.

C2.140 STR (register offset)

Store with register offset, pre-indexed register offset, or post-indexed register offset.

Syntax

```
STR{type}{cond} Rt, [Rn, ±Rm {, shift}] ; register offset
STR{type}{cond} Rt, [Rn, ±Rm {, shift}]! ; pre-indexed ; A32 only
STR{type}{cond} Rt, [Rn], ±Rm {, shift} ; post-indexed ; A32 only
STRD{cond} Rt, Rt2, [Rn, ±Rm] ; register offset, doubleword ; A32 only
STRD{cond} Rt, Rt2, [Rn, ±Rm]! ; pre-indexed, doubleword ; A32 only
STRD{cond} Rt, Rt2, [Rn], ±Rm ; post-indexed, doubleword ; A32 only
```

where:

type

can be any one of:

B
: Byte

H
: Halfword

-
: omitted, for Word.

cond
: is an optional condition code.

Rt
: is the general-purpose register to store.

Rn
: is the general-purpose register on which the memory address is based.

Rm
: is a general-purpose register containing a value to be used as the offset. *–Rm* is not permitted in T32 code.

shift
: is an optional shift.

Rt2
: is the additional register to store for doubleword operations.

Not all options are available in every instruction set and architecture.

Offset register and shift options

The following table shows the ranges of offsets and availability of this instruction:

Table C2-16 Options and architectures, STR (register offsets)

Instruction	±*Rm* [ad]	shift		
A32, word or byte	±*Rm*	LSL #0-31	LSR #1-32	
		ASR #1-32	ROR #1-31	RRX
A32, halfword	±*Rm*	Not available		
A32, doubleword	±*Rm*	Not available		

[ad] Where ±Rm is shown, you can use –Rm, +Rm, or Rm. Where +Rm is shown, you cannot use –Rm.
[ae] Rt, Rn, and Rm must all be in the range R0-R7.

Table C2-16 Options and architectures, STR (register offsets) (continued)

Instruction	±*Rm* [ad]	shift		
T32 32-bit encoding, word, halfword, or byte	+*Rm*	`LSL #0-3`		
T32 16-bit encoding, all except doubleword [ae]	+*Rm*	Not available		

Register restrictions

In the pre-index and post-index forms, `Rn` must be different from `Rt`.

Doubleword register restrictions

For A32 instructions:

- *Rt* must be an even-numbered register.
- *Rt* must not be LR.
- Arm strongly recommends that you do not use `R12` for *Rt*.
- *Rt2* must be `R(t + 1)`.
- `Rn` must be different from *Rt2* in the pre-index and post-index forms.

Use of PC

In A32 instructions you can use PC for `Rt` in `STR` word instructions, and you can use PC for `Rn` in `STR` instructions with register offset syntax (that is, the forms that do not writeback to the `Rn`). However, this is deprecated.

Other uses of PC are not permitted in A32 instructions.

Use of PC in `STR` T32 instructions is not permitted.

Use of SP

You can use SP for `Rn`.

In A32 code, you can use SP for `Rt` in word instructions. You can use SP for `Rt` in non-word A32 instructions but this is deprecated.

You can use SP for `Rm` in A32 instructions but this is deprecated.

In T32 code, you can use SP for `Rt` in word instructions only. All other use of SP for `Rt` in this instruction is not permitted in T32 code.

Use of SP for `Rm` is not permitted in T32 state.

Related references

C1.9 Condition code suffixes on page C1-92

C2.141 STR, unprivileged

Unprivileged Store, byte, halfword, or word.

Syntax

`STR{type}T{cond} Rt, [Rn {, #offset}] ; immediate offset (T32, 32-bit encoding only)`

`STR{type}T{cond} Rt, [Rn] {, #offset} ; post-indexed (A32 only)`

`STR{type}T{cond} Rt, [Rn], ±Rm {, shift} ; post-indexed (register) (A32 only)`

where:

type

can be any one of:

B

Byte

H

Halfword

-

omitted, for Word.

cond

is an optional condition code.

Rt

is the register to load or store.

Rn

is the register on which the memory address is based.

offset

is an offset. If offset is omitted, the address is the value in *Rn*.

Rm

is a register containing a value to be used as the offset. *Rm* must not be PC.

shift

is an optional shift.

Operation

When these instructions are executed by privileged software, they access memory with the same restrictions as they would have if they were executed by unprivileged software.

When executed by unprivileged software, these instructions behave in exactly the same way as the corresponding store instruction, for example `STRBT` behaves in the same way as `STRB`.

Offset ranges and architectures

The following table shows the ranges of offsets and availability of this instruction:

Table C2-17 Offsets and architectures, STR (User mode)

Instruction	Immediate offset	Post-indexed	+/–*Rm* [af]	shift
A32, word or byte	Not available	-4095 to 4095	+/–*Rm*	`LSL #0-31`
				`LSR #1-32`
				`ASR #1-32`
				`ROR #1-31`
				`RRX`
A32, halfword	Not available	-255 to 255	+/–*Rm*	Not available
T32 32-bit encoding, word, halfword, or byte	0 to 255	Not available	Not available	

Related references

C1.9 Condition code suffixes on page C1-92

[af] You can use –Rm, +Rm, or Rm.

C2.142 STREX

Store Register Exclusive.

Syntax

`STREX{cond} Rd, Rt, [Rn {, #offset}]`

`STREXB{cond} Rd, Rt, [Rn]`

`STREXH{cond} Rd, Rt, [Rn]`

`STREXD{cond} Rd, Rt, Rt2, [Rn]`

where:

cond

is an optional condition code.

Rd

is the destination register for the returned status.

Rt

is the register to store.

Rt2

is the second register for doubleword stores.

Rn

is the register on which the memory address is based.

offset

is an optional offset applied to the value in *Rn*. *offset* is permitted only in T32 instructions. If *offset* is omitted, an offset of 0 is assumed.

Operation

STREX performs a conditional store to memory. The conditions are as follows:

- If the physical address does not have the Shared TLB attribute, and the executing processor has an outstanding tagged physical address, the store takes place, the tag is cleared, and the value 0 is returned in *Rd*.
- If the physical address does not have the Shared TLB attribute, and the executing processor does not have an outstanding tagged physical address, the store does not take place, and the value 1 is returned in *Rd*.
- If the physical address has the Shared TLB attribute, and the physical address is tagged as exclusive access for the executing processor, the store takes place, the tag is cleared, and the value 0 is returned in *Rd*.
- If the physical address has the Shared TLB attribute, and the physical address is not tagged as exclusive access for the executing processor, the store does not take place, and the value 1 is returned in *Rd*.

Restrictions

PC must not be used for any of *Rd*, *Rt*, *Rt2*, or *Rn*.

For STREX, *Rd* must not be the same register as *Rt*, *Rt2*, or *Rn*.

For A32 instructions:

- SP can be used but use of SP for any of *Rd*, *Rt*, or *Rt2* is deprecated.
- For STREXD, *Rt* must be an even numbered register, and not LR.

- *Rt2* must be *R(t+1)*.
- *offset* is not permitted.

For T32 instructions:

- SP can be used for Rn, but must not be used for any of *Rd*, *Rt*, or *Rt2*.
- The value of *offset* can be any multiple of four in the range 0-1020.

Usage

Use LDREX and STREX to implement interprocess communication in multiple-processor and shared-memory systems.

For reasons of performance, keep the number of instructions between corresponding LDREX and STREX instructions to a minimum.

Note

The address used in a STREX instruction must be the same as the address in the most recently executed LDREX instruction.

Availability

All these 32-bit instructions are available in A32 and T32.

There are no 16-bit versions of these instructions.

Examples

```
    MOV r1, #0x1                ; load the 'lock taken' value
try
    LDREX r0, [LockAddr]        ; load the lock value
    CMP r0, #0                  ; is the lock free?
    STREXEQ r0, r1, [LockAddr]  ; try and claim the lock
    CMPEQ r0, #0                ; did this succeed?
    BNE try                     ; no - try again
    ....                        ; yes - we have the lock
```

Related references

C1.9 Condition code suffixes on page C1-92

C2.143 SUB

Subtract without carry.

Syntax

`SUB{S}{cond} {Rd}, Rn, Operand2`

`SUB{cond} {Rd}, Rn, #imm12 ; T32, 32-bit encoding only`

where:

`S`

is an optional suffix. If `S` is specified, the condition flags are updated on the result of the operation.

`cond`

is an optional condition code.

`Rd`

is the destination register.

`Rn`

is the register holding the first operand.

`Operand2`

is a flexible second operand.

`imm12`

is any value in the range 0-4095.

Operation

The `SUB` instruction subtracts the value of *`Operand2`* or *`imm12`* from the value in *`Rn`*.

In certain circumstances, the assembler can substitute one instruction for another. Be aware of this when reading disassembly listings.

Use of PC and SP in T32 instructions

In general, you cannot use PC (`R15`) for *`Rd`*, or any operand. The exception is you can use PC for *`Rn`* in 32-bit T32 `SUB` instructions, with a constant *`Operand2`* value in the range 0-4095, and no `S` suffix. These instructions are useful for generating PC-relative addresses. Bit[1] of the PC value reads as 0 in this case, so that the base address for the calculation is always word-aligned.

Generally, you cannot use SP (`R13`) for *`Rd`*, or any operand, except that you can use SP for *`Rn`*.

Use of PC and SP in A32 instructions

You cannot use PC for *`Rd`* or any operand in a `SUB` instruction that has a register-controlled shift.

In `SUB` instructions without register-controlled shift, use of PC is deprecated except for the following cases:

- Use of PC for `Rd`.
- Use of PC for `Rn` in the instruction `SUB{cond} Rd, Rn, #Constant`.

If you use PC (`R15`) as *`Rn`* or *`Rm`*, the value used is the address of the instruction plus 8.

If you use PC as *`Rd`*:

- Execution branches to the address corresponding to the result.
- If you use the `S` suffix, see the `SUBS pc,lr` instruction.

You can use SP for *Rn* in `SUB` instructions, however, `SUBS PC, SP, #Constant` is deprecated.

You can use SP in `SUB` (register) if *Rn* is SP and *shift* is omitted or `LSL #1`, `LSL #2`, or `LSL #3`.

Other uses of SP in A32 `SUB` instructions are deprecated.

Note

Use of SP and PC is deprecated in A32 instructions.

Condition flags

If `S` is specified, the `SUB` instruction updates the N, Z, C and V flags according to the result.

16-bit instructions

The following forms of this instruction are available in T32 code, and are 16-bit instructions:

`SUBS` *`Rd, Rn, Rm`*

Rd, *Rn* and *Rm* must all be Lo registers. This form can only be used outside an IT block.

`SUB{`*`cond`*`} Rd, Rn, Rm`

Rd, *Rn* and *Rm* must all be Lo registers. This form can only be used inside an IT block.

`SUBS` *`Rd, Rn, #imm`*

imm range 0-7. *Rd* and *Rn* must both be Lo registers. This form can only be used outside an IT block.

`SUB{`*`cond`*`} Rd, Rn, #imm`

imm range 0-7. *Rd* and *Rn* must both be Lo registers. This form can only be used inside an IT block.

`SUBS` *`Rd, Rd, #imm`*

imm range 0-255. *Rd* must be a Lo register. This form can only be used outside an IT block.

`SUB{`*`cond`*`} Rd, Rd, #imm`

imm range 0-255. *Rd* must be a Lo register. This form can only be used inside an IT block.

`SUB{`*`cond`*`} SP, SP, #imm`

imm range 0-508, word aligned.

Example

```
    SUBS    r8, r6, #240        ; sets the flags based on the result
```

Multiword arithmetic examples

These instructions subtract one 96-bit integer contained in `R9`, `R10`, and `R11` from another 96-bit integer contained in `R6`, `R7`, and `R8`, and place the result in `R3`, `R4`, and `R5`:

```
    SUBS    r3, r6, r9
    SBCS    r4, r7, r10
    SBC     r5, r8, r11
```

For clarity, the above examples use consecutive registers for multiword values. There is no requirement to do this. The following, for example, is perfectly valid:

```
    SUBS    r6, r6, r9
    SBCS    r9, r2, r1
    SBC     r2, r8, r11
```

Related references

C2.3 Flexible second operand (Operand2) on page C2-112

C2.144 SUBS pc, lr on page C2-317

C1.9 Condition code suffixes on page C1-92

C2.144 SUBS pc, lr

Exception return, without popping anything from the stack.

Syntax

`SUBS{cond} pc, lr, #imm ; A32 and T32 code`

`MOVS{cond} pc, lr ; A32 and T32 code`

`op1S{cond} pc, Rn, #imm ; A32 code only and is deprecated`

`op1S{cond} pc, Rn, Rm {, shift} ; A32 code only and is deprecated`

`op2S{cond} pc, #imm ; A32 code only and is deprecated`

`op2S{cond} pc, Rm {, shift} ; A32 code only and is deprecated`

where:

op1

is one of `ADC`, `ADD`, `AND`, `BIC`, `EOR`, `ORN`, `ORR`, `RSB`, `RSC`, `SBC`, and `SUB`.

op2

is one of `MOV` and `MVN`.

cond

is an optional condition code.

imm

is an immediate value. In T32 code, it is limited to the range 0-255. In A32 code, it is a flexible second operand.

R*n*

is the first general-purpose source register. Arm deprecates the use of any register except `LR`.

R*m*

is the optionally shifted second or only general-purpose register.

shift

is an optional condition code.

Usage

`SUBS pc, lr, #imm` subtracts a value from the link register and loads the PC with the result, then copies the SPSR to the CPSR.

You can use `SUBS pc, lr, #imm` to return from an exception if there is no return state on the stack. The value of `#imm` depends on the exception to return from.

Notes

`SUBS pc, lr, #imm` writes an address to the PC. The alignment of this address must be correct for the instruction set in use after the exception return:

- For a return to A32, the address written to the PC must be word-aligned.
- For a return to T32, the address written to the PC must be halfword-aligned.
- For a return to Jazelle, there are no alignment restrictions on the address written to the PC.

No special precautions are required in software to follow these rules, if you use the instruction to return after a valid exception entry mechanism.

In T32, only `SUBS{cond} pc, lr, #imm` is a valid instruction. `MOVS pc, lr` is a synonym of `SUBS pc, lr, #0`. Other instructions are undefined.

In A32, only `SUBS{cond} pc, lr, #imm` and `MOVS{cond} pc, lr` are valid instructions. Other instructions are deprecated.

Caution

Do not use these instructions in User mode or System mode. The assembler cannot warn you about this.

Availability

This 32-bit instruction is available in A32 and T32.

The 32-bit T32 instruction is not available in the Armv7-M architecture.

There is no 16-bit version of this instruction in T32.

Related references

C2.12 AND on page C2-128
C2.58 MOV on page C2-199
C2.3 Flexible second operand (Operand2) on page C2-112
C2.9 ADD on page C2-121
C1.9 Condition code suffixes on page C1-92

C2.145 SVC

SuperVisor Call.

Syntax

`SVC{cond} #imm`

where:

cond

is an optional condition code.

imm

is an expression evaluating to an integer in the range:

- 0 to 2^{24}-1 (a 24-bit value) in an A32 instruction.
- 0-255 (an 8-bit value) in a T32 instruction.

Operation

The `SVC` instruction causes an exception. This means that the processor mode changes to Supervisor, the CPSR is saved to the Supervisor mode SPSR, and execution branches to the SVC vector.

imm is ignored by the processor. However, it can be retrieved by the exception handler to determine what service is being requested.

Note

`SVC` was called `SWI` in earlier versions of the A32 assembly language. `SWI` instructions disassemble to `SVC`, with a comment to say that this was formerly `SWI`.

Condition flags

This instruction does not change the flags.

Availability

This instruction is available in A32 and 16-bit T32 and in the Armv7 architectures.

There is no 32-bit version of this instruction in T32.

Related references

C1.9 Condition code suffixes on page C1-92

C2.146 SWP and SWPB

Swap data between registers and memory.

Note

These instruction are not supported in Armv8.

Syntax

`SWP{B}{cond} Rt, Rt2, [Rn]`

where:

cond

is an optional condition code.

B

is an optional suffix. If `B` is present, a byte is swapped. Otherwise, a 32-bit word is swapped.

Rt

is the destination register. *Rt* must not be PC.

Rt2

is the source register. *Rt2* can be the same register as *Rt*. *Rt2* must not be PC.

Rn

contains the address in memory. *Rn* must be a different register from both *Rt* and *Rt2*. *Rn* must not be PC.

Usage

You can use `SWP` and `SWPB` to implement semaphores:

- Data from memory is loaded into *Rt*.
- The contents of *Rt2* are saved to memory.
- If *Rt2* is the same register as *Rt*, the contents of the register are swapped with the contents of the memory location.

Note

The use of `SWP` and `SWPB` is deprecated. You can use `LDREX` and `STREX` instructions to implement more sophisticated semaphores.

Availability

These instructions are available in A32.

There are no T32 `SWP` or `SWPB` instructions.

Related references

C2.51 LDREX on page C2-189
C1.9 Condition code suffixes on page C1-92

C2.147 SXTAB

Sign extend Byte with Add, to extend an 8-bit value to a 32-bit value.

Syntax

`SXTAB{cond} {Rd}, Rn, Rm {,rotation}`

where:

cond

is an optional condition code.

Rd

is the destination register.

Rn

is the register holding the number to add.

Rm

is the register holding the value to extend.

rotation

is one of:

ROR #8

Value from *Rm* is rotated right 8 bits.

ROR #16

Value from *Rm* is rotated right 16 bits.

ROR #24

Value from *Rm* is rotated right 24 bits.

If *rotation* is omitted, no rotation is performed.

Operation

This instruction does the following:

1. Rotate the value from *Rm* right by 0, 8, 16 or 24 bits.
2. Extract bits[7:0] from the value obtained.
3. Sign extend to 32 bits.
4. Add the value from *Rn*.

Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

Condition flags

This instruction does not change the flags.

Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

Related references

C1.9 Condition code suffixes on page C1-92

C2.148 SXTAB16

Sign extend two Bytes with Add, to extend two 8-bit values to two 16-bit values.

Syntax

`SXTAB16{cond} {Rd}, Rn, Rm {,rotation}`

where:

cond

is an optional condition code.

Rd

is the destination register.

Rn

is the register holding the number to add.

Rm

is the register holding the value to extend.

rotation

is one of:

ROR #8

Value from *Rm* is rotated right 8 bits.

ROR #16

Value from *Rm* is rotated right 16 bits.

ROR #24

Value from *Rm* is rotated right 24 bits.

If *rotation* is omitted, no rotation is performed.

Operation

This instruction does the following:

1. Rotate the value from *Rm* right by 0, 8, 16 or 24 bits.
2. Extract bits[23:16] and bits[7:0] from the value obtained.
3. Sign extend to 16 bits.
4. Add them to bits[31:16] and bits[15:0] respectively of *Rn* to form bits[31:16] and bits[15:0] of the result.

Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

Condition flags

This instruction does not change the flags.

Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

Related references

C1.9 Condition code suffixes on page C1-92

C2.149 SXTAH

Sign extend Halfword with Add, to extend a 16-bit value to a 32-bit value.

Syntax

`SXTAH{cond} {Rd}, Rn, Rm {,rotation}`

where:

cond

is an optional condition code.

Rd

is the destination register.

Rn

is the register holding the number to add.

Rm

is the register holding the value to extend.

rotation

is one of:

ROR #8

Value from *Rm* is rotated right 8 bits.

ROR #16

Value from *Rm* is rotated right 16 bits.

ROR #24

Value from *Rm* is rotated right 24 bits.

If *rotation* is omitted, no rotation is performed.

Operation

This instruction does the following:

1. Rotate the value from *Rm* right by 0, 8, 16 or 24 bits.
2. Extract bits[15:0] from the value obtained.
3. Sign extend to 32 bits.
4. Add the value from *Rn*.

Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

Condition flags

This instruction does not change the flags.

Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

Related references

C1.9 Condition code suffixes on page C1-92

C2.150 SXTB

Sign extend Byte, to extend an 8-bit value to a 32-bit value.

Syntax

`SXTB{cond} {Rd}, Rm {,rotation}`

where:

cond

is an optional condition code.

Rd

is the destination register.

Rm

is the register holding the value to extend.

rotation

is one of:

ROR #8

Value from *Rm* is rotated right 8 bits.

ROR #16

Value from *Rm* is rotated right 16 bits.

ROR #24

Value from *Rm* is rotated right 24 bits.

If *rotation* is omitted, no rotation is performed.

Operation

This instruction does the following:

1. Rotates the value from *Rm* right by 0, 8, 16 or 24 bits.
2. Extracts bits[7:0] from the value obtained.
3. Sign extends to 32 bits.

Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

Condition flags

This instruction does not change the flags.

16-bit instructions

The following form of this instruction is available in T32 code, and is a 16-bit instruction:

SXTB *Rd, Rm*

Rd and *Rm* must both be Lo registers.

Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

The 16-bit instruction is available in T32.

Related references

C1.9 Condition code suffixes on page C1-92

C2.151 SXTB16

Sign extend two bytes.

Syntax

`SXTB16{cond} {Rd}, Rm {,rotation}`

where:

cond

is an optional condition code.

Rd

is the destination register.

Rm

is the register holding the value to extend.

rotation

is one of:

ROR #8

Value from *Rm* is rotated right 8 bits.

ROR #16

Value from *Rm* is rotated right 16 bits.

ROR #24

Value from *Rm* is rotated right 24 bits.

If *rotation* is omitted, no rotation is performed.

Operation

`SXTB16` extends two 8-bit values to two 16-bit values. It does this by:

1. Rotating the value from *Rm* right by 0, 8, 16 or 24 bits.
2. Extracting bits[23:16] and bits[7:0] from the value obtained.
3. Sign extending to 16 bits each.

Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

Condition flags

This instruction does not change the flags.

Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

Related references

C1.9 Condition code suffixes on page C1-92

C2.152 SXTH

Sign extend Halfword.

Syntax

`SXTH{cond} {Rd}, Rm {,rotation}`

where:

cond

is an optional condition code.

Rd

is the destination register.

Rm

is the register holding the value to extend.

rotation

is one of:

ROR #8

Value from *Rm* is rotated right 8 bits.

ROR #16

Value from *Rm* is rotated right 16 bits.

ROR #24

Value from *Rm* is rotated right 24 bits.

If *rotation* is omitted, no rotation is performed.

Operation

SXTH extends a 16-bit value to a 32-bit value. It does this by:

1. Rotating the value from *Rm* right by 0, 8, 16 or 24 bits.
2. Extracting bits[15:0] from the value obtained.
3. Sign extending to 32 bits.

Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

Condition flags

This instruction does not change the flags.

16-bit instructions

The following form of this instruction is available in T32 code, and is a 16-bit instruction:

SXTH ***Rd, Rm***

Rd and *Rm* must both be Lo registers.

Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

The 16-bit instruction is available in T32.

Example

```
SXTH        r3, r9
```

Incorrect example

```
SXTH    r3, r9, ROR #12 ; rotation must be 0, 8, 16, or 24.
```

Related references

C1.9 Condition code suffixes on page C1-92

C2.153 SYS

Execute system coprocessor instruction.

Syntax

`SYS{cond} instruction{, Rn}`

where:

cond

is an optional condition code.

instruction

is the coprocessor instruction to execute.

Rn

is an operand to the instruction. For instructions that take an argument, *Rn* is compulsory. For instructions that do not take an argument, *Rn* is optional and if it is not specified, `R0` is used. *Rn* must not be PC.

Usage

You can use this pseudo-instruction to execute special coprocessor instructions such as cache, branch predictor, and TLB operations. The instructions operate by writing to special write-only coprocessor registers. The instruction names are the same as the write-only coprocessor register names and are listed in the *Arm® Architecture Reference Manual*. For example:

```
SYS ICIALLUIS ; invalidates all instruction caches Inner Shareable
              ; to Point of Unification and also flushes branch
              ; target cache.
```

Availability

This 32-bit instruction is available in A32 and T32.

The 32-bit T32 instruction is not available in the Armv7-M architecture.

There is no 16-bit version of this instruction in T32.

Related references

C1.9 Condition code suffixes on page C1-92

Related information

Arm Architecture Reference Manual

C2.154 TBB and TBH

Table Branch Byte and Table Branch Halfword.

Syntax

`TBB [Rn, Rm]`

`TBH [Rn, Rm, LSL #1]`

where:

Rn

is the base register. This contains the address of the table of branch lengths. *Rn* must not be SP.

If PC is specified for *Rn*, the value used is the address of the instruction plus 4.

Rm

is the index register. This contains an index into the table.

Rm must not be PC or SP.

Operation

These instructions cause a PC-relative forward branch using a table of single byte offsets (`TBB`) or halfword offsets (`TBH`). *Rn* provides a pointer to the table, and *Rm* supplies an index into the table. The branch length is twice the value of the byte (`TBB`) or the halfword (`TBH`) returned from the table. The target of the branch table must be in the same execution state.

Architectures

These 32-bit T32 instructions are available.

There are no versions of these instructions in A32 or in 16-bit T32 encodings.

C2.155 TEQ

Test Equivalence.

Syntax

`TEQ{cond} Rn, Operand2`

where:

cond

is an optional condition code.

Rn

is the general-purpose register holding the first operand.

Operand2

is a flexible second operand.

Usage

This instruction tests the value in a register against *Operand2*. It updates the condition flags on the result, but does not place the result in any register.

The `TEQ` instruction performs a bitwise Exclusive OR operation on the value in *Rn* and the value of *Operand2*. This is the same as an `EORS` instruction, except that the result is discarded.

Use the `TEQ` instruction to test if two values are equal, without affecting the V or C flags (as `CMP` does).

`TEQ` is also useful for testing the sign of a value. After the comparison, the N flag is the logical Exclusive OR of the sign bits of the two operands.

Register restrictions

In this T32 instruction, you cannot use SP or PC for *Rn* or *Operand2*.

In this A32 instruction, use of SP or PC is deprecated.

For A32 instructions:

- If you use PC (`R15`) as *Rn*, the value used is the address of the instruction plus 8.
- You cannot use PC for any operand in any data processing instruction that has a register-controlled shift.

Condition flags

This instruction:

- Updates the N and Z flags according to the result.
- Can update the C flag during the calculation of *Operand2*.
- Does not affect the V flag.

Architectures

This instruction is available in A32 and T32.

Correct example

```
TEQEQ   r10, r9
```

Incorrect example

```
TEQ     pc, r1, ROR r0      ; PC not permitted with register
                            ; controlled shift
```

Related references

C2.3 Flexible second operand (Operand2) on page C2-112

C1.9 Condition code suffixes on page C1-92

C2.156 TST

Test bits.

Syntax

`TST{cond} Rn, Operand2`

where:

cond

is an optional condition code.

Rn

is the general-purpose register holding the first operand.

Operand2

is a flexible second operand.

Operation

This instruction tests the value in a register against *Operand2*. It updates the condition flags on the result, but does not place the result in any register.

The `TST` instruction performs a bitwise AND operation on the value in *Rn* and the value of *Operand2*. This is the same as an `ANDS` instruction, except that the result is discarded.

Register restrictions

In this T32 instruction, you cannot use SP or PC for *Rn* or *Operand2*.

In this A32 instruction, use of SP or PC is deprecated.

For A32 instructions:

- If you use PC (`R15`) as *Rn*, the value used is the address of the instruction plus 8.
- You cannot use PC for any operand in any data processing instruction that has a register-controlled shift.

Condition flags

This instruction:

- Updates the N and Z flags according to the result.
- Can update the C flag during the calculation of *Operand2*.
- Does not affect the V flag.

16-bit instructions

The following form of the `TST` instruction is available in T32 code, and is a 16-bit instruction:

`TST` ***`Rn, Rm`***

Rn and *Rm* must both be Lo registers.

Architectures

This instruction is available A32 and T32.

Examples

```
    TST     r0, #0x3F8
    TSTNE   r1, r5, ASR r1
```

Related references

C2.3 Flexible second operand (Operand2) on page C2-112

C1.9 Condition code suffixes on page C1-92

C2.157 TT, TTT, TTA, TTAT

Test Target (Alternate Domain, Unprivileged).

Syntax

```
TT{cond}{q} Rd, Rn ; T1 TT general registers (T32)
TTA{cond}{q} Rd, Rn ; T1 TTA general registers (T32)
TTAT{cond}{q} Rd, Rn ; T1 TTAT general registers (T32)
TTT{cond}{q} Rd, Rn ; T1 TTT general registers (T32)
```

Where:

cond
Is an optional condition code. It specifies the condition under which the instruction is executed. If `cond` is omitted, it defaults to *always* (`AL`). See *Chapter C1 Condition Codes* on page C1-83.

q
Is an optional instruction width specifier. See *C2.2 Instruction width specifiers* on page C2-111.

Rd
Is the destination general-purpose register into which the status result of the target test is written.

Rn
Is the general-purpose base register.

Usage

Test Target (TT) queries the security state and access permissions of a memory location.

Test Target Unprivileged (TTT) queries the security state and access permissions of a memory location for an unprivileged access to that location.

Test Target Alternate Domain (TTA) and Test Target Alternate Domain Unprivileged (TTAT) query the security state and access permissions of a memory location for a Non-secure access to that location. These instructions are only valid when executing in Secure state, and are UNDEFINED if used from Non-secure state.

These instructions return the security state and access permissions in the destination register, the contents of which are as follows:

Bits	Name	Description
[7:0]	MREGION	The MPU region that the address maps to. This field is `0` if MRVALID is `0`.
[15:8]	SREGION	The SAU region that the address maps to. This field is only valid if the instruction is executed from Secure state. This field is `0` if SRVALID is `0`.
[16]	MRVALID	Set to `1` if the MREGION content is valid. Set to `0` if the MREGION content is invalid.
[17]	SRVALID	Set to `1` if the SREGION content is valid. Set to `0` if the SREGION content is invalid.
[18]	R	Read accessibility. Set to `1` if the memory location can be read according to the permissions of the selected MPU when operating in the current mode. For `TTT` and `TTAT`, this bit returns the permissions for unprivileged access, regardless of whether the current mode is privileged or unprivileged.
[19]	RW	Read/write accessibility. Set to `1` if the memory location can be read and written according to the permissions of the selected MPU when operating in the current mode. For `TTT` and `TTAT`, this bit returns the permissions for unprivileged access, regardless of whether the current mode is privileged or unprivileged.
[20]	NSR	Equal to R AND NOT S. Can be used in combination with the `LSLS` (immediate) instruction to check both the MPU and SAU/IDAU permissions. This bit is only valid if the instruction is executed from Secure state and the R field is valid.

(continued)

Bits	Name	Description
[21]	NSRW	Equal to RW AND NOT S. Can be used in combination with the `LSLS` (immediate) instruction to check both the MPU and SAU/IDAU permissions. This bit is only valid if the instruction is executed from Secure state and the RW field is valid.
[22]	S	Security. A value of `1` indicates the memory location is Secure, and a value of `0` indicates the memory location is Non-secure. This bit is only valid if the instruction is executed from Secure state.
[23]	IRVALID	IREGION valid flag. For a Secure request, indicates the validity of the IREGION field. Set to `1` if the IREGION content is valid. Set to `0` if the IREGION content is invalid. This bit is always `0` if the IDAU cannot provide a region number, the address is exempt from security attribution, or if the requesting `TT` instruction is executed from the Non-secure state.
[31:24]	IREGION	IDAU region number. Indicates the IDAU region number containing the target address. This field is `0` if IRVALID is`0`.

Invalid fields are `0`.

The MREGION field is invalid and `0` if any of the following conditions are true:

- The MPU is not present or MPU_CTRL.ENABLE is `0`.
- The address did not match any enabled MPU regions.
- The address matched multiple MPU regions.
- `TT` or `TTT` was executed from an unprivileged mode.

The SREGION field is invalid and `0` if any of the following conditions are true:

- SAU_CTRL.ENABLE is set to `0`.
- The address did not match any enabled SAU regions.
- The address matched multiple SAU regions.
- The SAU attributes were overridden by the IDAU.
- The instruction is executed from Non-secure state, or is executed on a processor that does not implement the Armv8-M Security Extensions.

The R and RW bits are invalid and `0` if any of the following conditions are true:

- The address matched multiple MPU regions.
- `TT` or `TTT` is executed from an unprivileged mode.

Related references

C1.9 Condition code suffixes on page C1-92
C2.2 Instruction width specifiers on page C2-111

C2.158 UADD8

Unsigned parallel byte-wise addition.

Syntax

`UADD8{cond} {Rd}, Rn, Rm`

where:

cond

is an optional condition code.

Rd

is the destination general-purpose register.

Rm, Rn

are the general-purpose registers holding the operands.

Operation

This instruction performs four unsigned integer additions on the corresponding bytes of the operands and writes the results into the corresponding bytes of the destination. The results are modulo 2^8. It sets the APSR GE flags.

Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

GE flags

This instruction does not affect the N, Z, C, V, or Q flags.

It sets the GE flags in the APSR as follows:

GE[0]

for bits[7:0] of the result.

GE[1]

for bits[15:8] of the result.

GE[2]

for bits[23:16] of the result.

GE[3]

for bits[31:24] of the result.

It sets a GE flag to 1 to indicate that the corresponding result overflowed, generating a carry. This is equivalent to an `ADDS` instruction setting the C condition flag to 1.

You can use these flags to control a following `SEL` instruction.

Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

Related references

C2.100 SEL on page C2-257
C1.9 Condition code suffixes on page C1-92

C2.159 UADD16

Unsigned parallel halfword-wise addition.

Syntax

`UADD16{cond} {Rd}, Rn, Rm`

where:

cond

is an optional condition code.

Rd

is the destination general-purpose register.

Rm, Rn

are the general-purpose registers holding the operands.

Operation

This instruction performs two unsigned integer additions on the corresponding halfwords of the operands and writes the results into the corresponding halfwords of the destination. The results are modulo 2^{16}. It sets the APSR GE flags.

Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

GE flags

This instruction does not affect the N, Z, C, V, or Q flags.

It sets the GE flags in the APSR as follows:

GE[1:0]

for bits[15:0] of the result.

GE[3:2]

for bits[31:16] of the result.

It sets a pair of GE flags to 1 to indicate that the corresponding result overflowed, generating a carry. This is equivalent to an `ADDS` instruction setting the C condition flag to 1.

You can use these flags to control a following `SEL` instruction.

——— **Note** ———

GE[1:0] are set or cleared together, and GE[3:2] are set or cleared together.

———————————

Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

Related references

C2.100 SEL on page C2-257
C1.9 Condition code suffixes on page C1-92

C2.160 UASX

Unsigned parallel add and subtract halfwords with exchange.

Syntax

`UASX{cond} {Rd}, Rn, Rm`

where:

cond

is an optional condition code.

Rd

is the destination general-purpose register.

Rm, Rn

are the general-purpose registers holding the operands.

Operation

This instruction exchanges the two halfwords of the second operand, then performs an addition on the two top halfwords of the operands and a subtraction on the bottom two halfwords. It writes the results into the corresponding halfwords of the destination. The results are modulo 2^{16}. It sets the APSR GE flags.

Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

GE flags

This instruction does not affect the N, Z, C, V, or Q flags.

It sets the GE flags in the APSR as follows:

GE[1:0]

for bits[15:0] of the result.

GE[3:2]

for bits[31:16] of the result.

It sets GE[1:0] to 1 to indicate that the subtraction gave a result greater than or equal to zero, meaning a borrow did not occur. This is equivalent to a `SUBS` instruction setting the C condition flag to 1.

It sets GE[3:2] to 1 to indicate that the addition overflowed, generating a carry. This is equivalent to an `ADDS` instruction setting the C condition flag to 1.

You can use these flags to control a following `SEL` instruction.

Note

GE[1:0] are set or cleared together, and GE[3:2] are set or cleared together.

Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

Related references

C2.100 SEL on page C2-257
C1.9 Condition code suffixes on page C1-92

C2.161 UBFX

Unsigned Bit Field Extract.

Syntax

`UBFX{cond} Rd, Rn, #lsb, #width`

where:

cond

is an optional condition code.

Rd

is the destination register.

Rn

is the source register.

lsb

is the bit number of the least significant bit in the bitfield, in the range 0 to 31.

width

is the width of the bitfield, in the range 1 to (32–*lsb*).

Operation

Copies adjacent bits from one register into the least significant bits of a second register, and zero extends to 32 bits.

Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

Condition flags

This instruction does not alter any flags.

Architectures

This instruction is available in A32 and T32.

There is no 16-bit version of this instruction in T32.

Related references

C1.9 Condition code suffixes on page C1-92

C2.162 UDF

Permanently Undefined.

Syntax

`UDF{c}{q} {#}imm ; A1 general registers (A32)`

`UDF{c}{q} {#}imm ; T1 general registers (T32)`

`UDF{c}.W {#}imm ; T2 general registers (T32)`

Where:

imm

The value depends on the instruction variant:

A1 general registers
For A32, a 16-bit unsigned immediate, in the range 0 to 65535.

T1 general registers
For T32, an 8-bit unsigned immediate, in the range 0 to 255.

T2 general registers
For T32, a 16-bit unsigned immediate, in the range 0 to 65535.

Note

The PE ignores the value of this constant.

c

Is an optional condition code. See *Chapter C1 Condition Codes* on page C1-83. Arm deprecates using any *c* value other than `AL`.

q

Is an optional instruction width specifier. See *C2.2 Instruction width specifiers* on page C2-111.

Usage

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.

Related references

C2.1 A32 and T32 instruction summary on page C2-106

C2.163 UDIV

Unsigned Divide.

Syntax

`UDIV{cond} {Rd}, Rn, Rm`

where:

cond
: is an optional condition code.

Rd
: is the destination register.

Rn
: is the register holding the value to be divided.

Rm
: is a register holding the divisor.

Register restrictions

PC or SP cannot be used for `Rd`, `Rn`, or `Rm`.

Architectures

This 32-bit T32 instruction is available in Armv7-R, Armv7-M and Armv8-M Mainline.

This 32-bit A32 instruction is optional in Armv7-R.

This 32-bit A32 and T32 instruction is available in Armv7-A if Virtualization Extensions are implemented, and optional if not.

There is no 16-bit T32 `UDIV` instruction.

Related references

C1.9 Condition code suffixes on page C1-92

C2.164 UHADD8

Unsigned halving parallel byte-wise addition.

Syntax

`UHADD8{cond} {Rd}, Rn, Rm`

where:

cond

is an optional condition code.

Rd

is the destination general-purpose register.

Rm, Rn

are the general-purpose registers holding the operands.

Operation

This instruction performs four unsigned integer additions on the corresponding bytes of the operands, halves the results, and writes the results into the corresponding bytes of the destination. This cannot cause overflow.

Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

Condition flags

This instruction does not affect the N, Z, C, V, Q, or GE flags.

Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

Related references

C1.9 Condition code suffixes on page C1-92

C2.165 UHADD16

Unsigned halving parallel halfword-wise addition.

Syntax

`UHADD16{cond} {Rd}, Rn, Rm`

where:

cond

is an optional condition code.

Rd

is the destination general-purpose register.

Rm, Rn

are the general-purpose registers holding the operands.

Operation

This instruction performs two unsigned integer additions on the corresponding halfwords of the operands, halves the results, and writes the results into the corresponding halfwords of the destination. This cannot cause overflow.

Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

Condition flags

This instruction does not affect the N, Z, C, V, Q, or GE flags.

Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

Related references

C1.9 Condition code suffixes on page C1-92

C2.166 UHASX

Unsigned halving parallel add and subtract halfwords with exchange.

Syntax

`UHASX{cond} {Rd}, Rn, Rm`

where:

cond

is an optional condition code.

Rd

is the destination general-purpose register.

Rm, Rn

are the general-purpose registers holding the operands.

Operation

This instruction exchanges the two halfwords of the second operand, then performs an addition on the two top halfwords of the operands and a subtraction on the bottom two halfwords. It halves the results and writes them into the corresponding halfwords of the destination. This cannot cause overflow.

Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

Condition flags

This instruction does not affect the N, Z, C, V, Q, or GE flags.

Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

Related references

C1.9 Condition code suffixes on page C1-92

C2.167 UHSAX

Unsigned halving parallel subtract and add halfwords with exchange.

Syntax

`UHSAX{cond} {Rd}, Rn, Rm`

where:

cond

is an optional condition code.

Rd

is the destination general-purpose register.

Rm, Rn

are the general-purpose registers holding the operands.

Operation

This instruction exchanges the two halfwords of the second operand, then performs a subtraction on the two top halfwords of the operands and an addition on the bottom two halfwords. It halves the results and writes them into the corresponding halfwords of the destination. This cannot cause overflow.

Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

Condition flags

This instruction does not affect the N, Z, C, V, Q, or GE flags.

Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

Related references

C1.9 Condition code suffixes on page C1-92

C2.168 UHSUB8

Unsigned halving parallel byte-wise subtraction.

Syntax

`UHSUB8{cond} {Rd}, Rn, Rm`

where:

cond

is an optional condition code.

Rd

is the destination general-purpose register.

Rm, Rn

are the general-purpose registers holding the operands.

Operation

This instruction subtracts each byte of the second operand from the corresponding byte of the first operand, halves the results, and writes the results into the corresponding bytes of the destination. This cannot cause overflow.

Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

Condition flags

This instruction does not affect the N, Z, C, V, Q, or GE flags.

Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

Related references

C1.9 Condition code suffixes on page C1-92

C2.169 UHSUB16

Unsigned halving parallel halfword-wise subtraction.

Syntax

`UHSUB16{cond} {Rd}, Rn, Rm`

where:

cond

is an optional condition code.

Rd

is the destination general-purpose register.

Rm, Rn

are the general-purpose registers holding the operands.

Operation

This instruction subtracts each halfword of the second operand from the corresponding halfword of the first operand, halves the results, and writes the results into the corresponding halfwords of the destination. This cannot cause overflow.

Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

Condition flags

This instruction does not affect the N, Z, C, V, Q, or GE flags.

Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

Related references

C1.9 Condition code suffixes on page C1-92

C2.170 UMAAL

Unsigned Multiply Accumulate Accumulate Long.

Syntax

`UMAAL{cond} RdLo, RdHi, Rn, Rm`

where:

cond

is an optional condition code.

RdLo, RdHi

are the destination registers for the 64-bit result. They also hold the two 32-bit accumulate operands. *RdLo* and *RdHi* must be different registers.

Rn, Rm

are the general-purpose registers holding the multiply operands.

Operation

The `UMAAL` instruction multiplies the 32-bit values in *Rn* and *Rm*, adds the two 32-bit values in *RdHi* and *RdLo*, and stores the 64-bit result to *RdLo, RdHi*.

Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

Condition flags

This instruction does not change the flags.

Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

Examples

```
UMAAL       r8, r9, r2, r3
UMAALGE     r2, r0, r5, r3
```

Related references

C1.9 Condition code suffixes on page C1-92

C2.171 UMLAL

Unsigned Long Multiply, with optional Accumulate, with 32-bit operands and 64-bit result and accumulator.

Syntax

`UMLAL{S}{cond} RdLo, RdHi, Rn, Rm`

where:

`S`

is an optional suffix available in A32 state only. If `S` is specified, the condition flags are updated based on the result of the operation.

`cond`

is an optional condition code.

`RdLo, RdHi`

are the destination registers. They also hold the accumulating value. *`RdLo`* and *`RdHi`* must be different registers.

`Rn, Rm`

are general-purpose registers holding the operands.

Operation

The `UMLAL` instruction interprets the values from *`Rn`* and *`Rm`* as unsigned integers. It multiplies these integers, and adds the 64-bit result to the 64-bit unsigned integer contained in *`RdHi`* and *`RdLo`*.

Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

Condition flags

If `S` is specified, this instruction:

- Updates the N and Z flags according to the result.
- Does not affect the C or V flags.

Architectures

This 32-bit instruction is available in A32 and T32.

There is no 16-bit version of this instruction in T32.

Example

```
    UMLALS      r4, r5, r3, r8
```

Related references

C1.9 Condition code suffixes on page C1-92

C2.172 UMULL

Unsigned Long Multiply, with 32-bit operands, and 64-bit result.

Syntax

`UMULL{S}{cond} RdLo, RdHi, Rn, Rm`

where:

`S`

is an optional suffix available in A32 state only. If `S` is specified, the condition flags are updated based on the result of the operation.

`cond`

is an optional condition code.

`RdLo, RdHi`

are the destination general-purpose registers. *`RdLo`* and *`RdHi`* must be different registers.

`Rn, Rm`

are general-purpose registers holding the operands.

Operation

The `UMULL` instruction interprets the values from *`Rn`* and *`Rm`* as unsigned integers. It multiplies these integers and places the least significant 32 bits of the result in *`RdLo`*, and the most significant 32 bits of the result in *`RdHi`*.

Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

Condition flags

If `S` is specified, this instruction:

- Updates the N and Z flags according to the result.
- Does not affect the C or V flags.

Architectures

This 32-bit instruction is available in A32 and T32.

There is no 16-bit version of this instruction in T32.

Example

```
    UMULL       r0, r4, r5, r6
```

Related references

C1.9 Condition code suffixes on page C1-92

C2.173 UQADD8

Unsigned saturating parallel byte-wise addition.

Syntax

`UQADD8{cond} {Rd}, Rn, Rm`

where:

cond

is an optional condition code.

Rd

is the destination general-purpose register.

Rm, Rn

are the general-purpose registers holding the operands.

Operation

This instruction performs four unsigned integer additions on the corresponding bytes of the operands and writes the results into the corresponding bytes of the destination. It saturates the results to the unsigned range $0 \leq x \leq 2^8$ -1. The Q flag is not affected even if this operation saturates.

Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

Condition flags

This instruction does not affect the N, Z, C, V, Q, or GE flags.

Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

Related references

C1.9 Condition code suffixes on page C1-92

C2.174 UQADD16

Unsigned saturating parallel halfword-wise addition.

Syntax

`UQADD16{cond} {Rd}, Rn, Rm`

where:

cond

is an optional condition code.

Rd

is the destination general-purpose register.

Rm, Rn

are the general-purpose registers holding the operands.

Operation

This instruction performs two unsigned integer additions on the corresponding halfwords of the operands and writes the results into the corresponding halfwords of the destination. It saturates the results to the unsigned range $0 \le x \le 2^{16}$ -1. The Q flag is not affected even if this operation saturates.

Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

Condition flags

This instruction does not affect the N, Z, C, V, Q, or GE flags.

Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

Related references

C1.9 Condition code suffixes on page C1-92

C2.175 UQASX

Unsigned saturating parallel add and subtract halfwords with exchange.

Syntax

`UQASX{cond} {Rd}, Rn, Rm`

where:

cond

is an optional condition code.

Rd

is the destination general-purpose register.

Rm, Rn

are the general-purpose registers holding the operands.

Operation

This instruction exchanges the two halfwords of the second operand, then performs an addition on the two top halfwords of the operands and a subtraction on the bottom two halfwords. It writes the results into the corresponding halfwords of the destination. It saturates the results to the unsigned range $0 \leq x \leq 2^{16} - 1$. The Q flag is not affected even if this operation saturates.

Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

Condition flags

This instruction does not affect the N, Z, C, V, Q, or GE flags.

Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

Related references

C1.9 Condition code suffixes on page C1-92

C2.176 UQSAX

Unsigned saturating parallel subtract and add halfwords with exchange.

Syntax

`UQSAX{cond} {Rd}, Rn, Rm`

where:

cond

is an optional condition code.

Rd

is the destination general-purpose register.

Rm, Rn

are the general-purpose registers holding the operands.

Operation

This instruction exchanges the two halfwords of the second operand, then performs a subtraction on the two top halfwords of the operands and an addition on the bottom two halfwords. It writes the results into the corresponding halfwords of the destination. It saturates the results to the unsigned range $0 \leq x \leq 2^{16}-1$. The Q flag is not affected even if this operation saturates.

Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

Condition flags

This instruction does not affect the N, Z, C, V, Q, or GE flags.

Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

Related references

C1.9 Condition code suffixes on page C1-92

C2.177 UQSUB8

Unsigned saturating parallel byte-wise subtraction.

Syntax

`UQSUB8{cond} {Rd}, Rn, Rm`

where:

cond

is an optional condition code.

Rd

is the destination general-purpose register.

Rm, Rn

are the general-purpose registers holding the operands.

Operation

This instruction subtracts each byte of the second operand from the corresponding byte of the first operand and writes the results into the corresponding bytes of the destination. It saturates the results to the unsigned range $0 \leq x \leq 2^8 - 1$. The Q flag is not affected even if this operation saturates.

Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

Condition flags

This instruction does not affect the N, Z, C, V, Q, or GE flags.

Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

Related references

C1.9 Condition code suffixes on page C1-92

C2.178 UQSUB16

Unsigned saturating parallel halfword-wise subtraction.

Syntax

`UQSUB16{cond} {Rd}, Rn, Rm`

where:

cond

is an optional condition code.

Rd

is the destination general-purpose register.

Rm, Rn

are the general-purpose registers holding the operands.

Operation

This instruction subtracts each halfword of the second operand from the corresponding halfword of the first operand and writes the results into the corresponding halfwords of the destination. It saturates the results to the unsigned range $0 \le x \le 2^{16}$ -1. The Q flag is not affected even if this operation saturates.

Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

Condition flags

This instruction does not affect the N, Z, C, V, Q, or GE flags.

Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

Related references

C1.9 Condition code suffixes on page C1-92

C2.179 USAD8

Unsigned Sum of Absolute Differences.

Syntax

`USAD8{cond} {Rd}, Rn, Rm`

where:

cond

is an optional condition code.

Rd

is the destination register.

Rn

is the register holding the first operand.

Rm

is the register holding the second operand.

Operation

The `USAD8` instruction finds the four differences between the unsigned values in corresponding bytes of *Rn* and *Rm*. It adds the absolute values of the four differences, and saves the result to *Rd*.

Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

Condition flags

This instruction does not alter any flags.

Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

Example

```
    USAD8       r2, r4, r6
```

Related references

C1.9 Condition code suffixes on page C1-92

C2.180 USADA8

Unsigned Sum of Absolute Differences and Accumulate.

Syntax

`USADA8{cond} Rd, Rn, Rm, Ra`

where:

cond

is an optional condition code.

Rd

is the destination register.

Rn

is the register holding the first operand.

Rm

is the register holding the second operand.

Ra

is the register holding the accumulate operand.

Operation

The `USADA8` instruction adds the absolute values of the four differences to the value in *Ra*, and saves the result to *Rd*.

Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

Condition flags

This instruction does not alter any flags.

Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

Correct examples

```
USADA8      r0, r3, r5, r2
USADA8VS    r0, r4, r0, r1
```

Incorrect examples

```
USADA8      r2, r4, r6      ; USADA8 requires four registers
USADA16     r0, r4, r0, r1  ; no such instruction
```

Related references

C1.9 Condition code suffixes on page C1-92

C2.181 USAT

Unsigned Saturate to any bit position, with optional shift before saturating.

Syntax

`USAT{cond} Rd, #sat, Rm{, shift}`

where:

cond

is an optional condition code.

Rd

is the destination register.

sat

specifies the bit position to saturate to, in the range 0 to 31.

Rm

is the register containing the operand.

shift

is an optional shift. It must be one of the following:

ASR #*n*

where *n* is in the range 1-32 (A32) or 1-31 (T32).

LSL #*n*

where *n* is in the range 0-31.

Operation

The `USAT` instruction applies the specified shift to a signed value, then saturates to the unsigned range $0 \leq x \leq 2^{sat} - 1$.

Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

Q flag

If saturation occurs, this instruction sets the Q flag. To read the state of the Q flag, use an `MRS` instruction.

Architectures

This instruction is available in A32 and T32.

There is no 16-bit version of this instruction in T32.

Example

```
USATNE  r0, #7, r5
```

Related references

C2.131 SSAT16 on page C2-292
C2.62 MRS (PSR to general-purpose register) on page C2-204
C1.9 Condition code suffixes on page C1-92

C2.182 USAT16

Parallel halfword Saturate.

Syntax

`USAT16{cond} Rd, #sat, Rn`

where:

cond

is an optional condition code.

Rd

is the destination register.

sat

specifies the bit position to saturate to, in the range 0 to 15.

Rn

is the register holding the operand.

Operation

Halfword-wise unsigned saturation to any bit position.

The `USAT16` instruction saturates each signed halfword to the unsigned range $0 \leq x \leq 2^{sat}-1$.

Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

Q flag

If saturation occurs on either halfword, this instruction sets the Q flag. To read the state of the Q flag, use an `MRS` instruction.

Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

Example

```
    USAT16  r0, #7, r5
```

Related references

C2.62 MRS (PSR to general-purpose register) on page C2-204
C1.9 Condition code suffixes on page C1-92

C2.183 USAX

Unsigned parallel subtract and add halfwords with exchange.

Syntax

`USAX{cond} {Rd}, Rn, Rm`

where:

cond

is an optional condition code.

Rd

is the destination general-purpose register.

Rm, Rn

are the general-purpose registers holding the operands.

Operation

This instruction exchanges the two halfwords of the second operand, then performs a subtraction on the two top halfwords of the operands and an addition on the bottom two halfwords. It writes the results into the corresponding halfwords of the destination. The results are modulo 2^{16}. It sets the APSR GE flags.

Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

GE flags

This instruction does not affect the N, Z, C, V, or Q flags.

It sets the GE flags in the APSR as follows:

GE[1:0]

for bits[15:0] of the result.

GE[3:2]

for bits[31:16] of the result.

It sets GE[1:0] to 1 to indicate that the addition overflowed, generating a carry. This is equivalent to an `ADDS` instruction setting the C condition flag to 1.

It sets GE[3:2] to 1 to indicate that the subtraction gave a result greater than or equal to zero, meaning a borrow did not occur. This is equivalent to a `SUBS` instruction setting the C condition flag to 1.

You can use these flags to control a following `SEL` instruction.

Note

GE[1:0] are set or cleared together, and GE[3:2] are set or cleared together.

Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

Related references

C2.100 SEL on page C2-257
C1.9 Condition code suffixes on page C1-92

C2.184 USUB8

Unsigned parallel byte-wise subtraction.

Syntax

`USUB8{cond} {Rd}, Rn, Rm`

where:

cond

is an optional condition code.

Rd

is the destination general-purpose register.

Rm, Rn

are the general-purpose registers holding the operands.

Operation

This instruction subtracts each byte of the second operand from the corresponding byte of the first operand and writes the results into the corresponding bytes of the destination. The results are modulo 2^8. It sets the APSR GE flags.

Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

GE flags

This instruction does not affect the N, Z, C, V, or Q flags.

It sets the GE flags in the APSR as follows:

GE[0]

for bits[7:0] of the result.

GE[1]

for bits[15:8] of the result.

GE[2]

for bits[23:16] of the result.

GE[3]

for bits[31:24] of the result.

It sets a GE flag to 1 to indicate that the corresponding result is greater than or equal to zero, meaning a borrow did not occur. This is equivalent to a `SUBS` instruction setting the C condition flag to 1.

You can use these flags to control a following `SEL` instruction.

Availability

This 32-bit instruction is available in A32 and T32.

There is no 16-bit version of this instruction in T32.

Related references

C2.100 SEL on page C2-257

C1.9 Condition code suffixes on page C1-92

C2.185 USUB16

Unsigned parallel halfword-wise subtraction.

Syntax

`USUB16{cond} {Rd}, Rn, Rm`

where:

cond

is an optional condition code.

Rd

is the destination general-purpose register.

Rm, Rn

are the general-purpose registers holding the operands.

Operation

This instruction subtracts each halfword of the second operand from the corresponding halfword of the first operand and writes the results into the corresponding halfwords of the destination. The results are modulo 2^{16}. It sets the APSR GE flags.

Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

Condition flags

This instruction does not affect the N, Z, C, V, or Q flags.

It sets the GE flags in the APSR as follows:

GE[1:0]

for bits[15:0] of the result.

GE[3:2]

for bits[31:16] of the result.

It sets a pair of GE flags to 1 to indicate that the corresponding result is greater than or equal to zero, meaning a borrow did not occur. This is equivalent to a `SUBS` instruction setting the C condition flag to 1.

You can use these flags to control a following `SEL` instruction.

Note

GE[1:0] are set or cleared together, and GE[3:2] are set or cleared together.

Availability

This 32-bit instruction is available in A32 and T32.

There is no 16-bit version of this instruction in T32.

Related references

C2.100 SEL on page C2-257
C1.9 Condition code suffixes on page C1-92

C2.186 UXTAB

Zero extend Byte and Add.

Syntax

`UXTAB{cond} {Rd}, Rn, Rm {,rotation}`

where:

cond

is an optional condition code.

Rd

is the destination register.

Rn

is the register holding the number to add.

Rm

is the register holding the value to extend.

rotation

is one of:

ROR #8

Value from *Rm* is rotated right 8 bits.

ROR #16

Value from *Rm* is rotated right 16 bits.

ROR #24

Value from *Rm* is rotated right 24 bits.

If *rotation* is omitted, no rotation is performed.

Operation

`UXTAB` extends an 8-bit value to a 32-bit value. It does this by:

1. Rotating the value from *Rm* right by 0, 8, 16 or 24 bits.
2. Extracting bits[7:0] from the value obtained.
3. Zero extending to 32 bits.
4. Adding the value from *Rn*.

Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

Condition flags

This instruction does not change the flags.

Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

Related references

C1.9 Condition code suffixes on page C1-92

C2.187 UXTAB16

Zero extend two Bytes and Add.

Syntax

`UXTAB16{cond} {Rd}, Rn, Rm {,rotation}`

where:

cond

is an optional condition code.

Rd

is the destination register.

Rn

is the register holding the number to add.

Rm

is the register holding the value to extend.

rotation

is one of:

ROR #8

Value from *Rm* is rotated right 8 bits.

ROR #16

Value from *Rm* is rotated right 16 bits.

ROR #24

Value from *Rm* is rotated right 24 bits.

If *rotation* is omitted, no rotation is performed.

Operation

`UXTAB16` extends two 8-bit values to two 16-bit values. It does this by:

1. Rotating the value from *Rm* right by 0, 8, 16 or 24 bits.
2. Extracting bits[23:16] and bits[7:0] from the value obtained.
3. Zero extending them to 16 bits.
4. Adding them to bits[31:16] and bits[15:0] respectively of *Rn* to form bits[31:16] and bits[15:0] of the result.

Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

Condition flags

This instruction does not change the flags.

Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

Example

```
    UXTAB16EQ    r0, r0, r4, ROR #16
```

Related references

C1.9 Condition code suffixes on page C1-92

C2.188 UXTAH

Zero extend Halfword and Add.

Syntax

`UXTAH{cond} {Rd}, Rn, Rm {,rotation}`

where:

cond

is an optional condition code.

Rd

is the destination register.

Rn

is the register holding the number to add.

Rm

is the register holding the value to extend.

rotation

is one of:

ROR #8

Value from *Rm* is rotated right 8 bits.

ROR #16

Value from *Rm* is rotated right 16 bits.

ROR #24

Value from *Rm* is rotated right 24 bits.

If *rotation* is omitted, no rotation is performed.

Operation

`UXTAH` extends a 16-bit value to a 32-bit value. It does this by:

1. Rotating the value from *Rm* right by 0, 8, 16 or 24 bits.
2. Extracting bits[15:0] from the value obtained.
3. Zero extending to 32 bits.
4. Adding the value from *Rn*.

Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

Condition flags

This instruction does not change the flags.

Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

Related references

C1.9 Condition code suffixes on page C1-92

C2.189 UXTB

Zero extend Byte.

Syntax

`UXTB{cond} {Rd}, Rm {,rotation}`

where:

cond

is an optional condition code.

Rd

is the destination register.

Rm

is the register holding the value to extend.

rotation

is one of:

ROR #8

Value from *Rm* is rotated right 8 bits.

ROR #16

Value from *Rm* is rotated right 16 bits.

ROR #24

Value from *Rm* is rotated right 24 bits.

If *rotation* is omitted, no rotation is performed.

Operation

`UXTB` extends an 8-bit value to a 32-bit value. It does this by:

1. Rotating the value from *Rm* right by 0, 8, 16, or 24 bits.
2. Extracting bits[7:0] from the value obtained.
3. Zero extending to 32 bits.

Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

Condition flags

This instruction does not change the flags.

16-bit instruction

The following form of this instruction is available in T32 code, and is a 16-bit instruction:

UXTB ***Rd, Rm***

Rd and *Rm* must both be Lo registers.

Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

The 16-bit instruction is available in T32.

Related references

C1.9 Condition code suffixes on page C1-92

C2.190 UXTB16

Zero extend two Bytes.

Syntax

`UXTB16{cond} {Rd}, Rm {,rotation}`

where:

cond

is an optional condition code.

Rd

is the destination register.

Rm

is the register holding the value to extend.

rotation

is one of:

ROR #8

Value from *Rm* is rotated right 8 bits.

ROR #16

Value from *Rm* is rotated right 16 bits.

ROR #24

Value from *Rm* is rotated right 24 bits.

If *rotation* is omitted, no rotation is performed.

Operation

`UXTB16` extends two 8-bit values to two 16-bit values. It does this by:

1. Rotating the value from *Rm* right by 0, 8, 16 or 24 bits.
2. Extracting bits[23:16] and bits[7:0] from the value obtained.
3. Zero extending each to 16 bits.

Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

Condition flags

This instruction does not change the flags.

Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

There is no 16-bit version of this instruction in T32.

Related references

C1.9 Condition code suffixes on page C1-92

C2.191 UXTH

Zero extend Halfword.

Syntax

`UXTH{cond} {Rd}, Rm {,rotation}`

where:

cond

is an optional condition code.

Rd

is the destination register.

Rm

is the register holding the value to extend.

rotation

is one of:

ROR #8

Value from *Rm* is rotated right 8 bits.

ROR #16

Value from *Rm* is rotated right 16 bits.

ROR #24

Value from *Rm* is rotated right 24 bits.

If *rotation* is omitted, no rotation is performed.

Operation

UXTH extends a 16-bit value to a 32-bit value. It does this by:

1. Rotating the value from *Rm* right by 0, 8, 16, or 24 bits.
2. Extracting bits[15:0] from the value obtained.
3. Zero extending to 32 bits.

Register restrictions

You cannot use PC for any operand.

You can use SP in A32 instructions but this is deprecated. You cannot use SP in T32 instructions.

Condition flags

This instruction does not change the flags.

16-bit instructions

The following form of this instruction is available in T32 code, and is a 16-bit instruction:

UXTH ***Rd, Rm***

Rd and *Rm* must both be Lo registers.

Availability

The 32-bit instruction is available in A32 and T32.

For the Armv7-M architecture, the 32-bit T32 instruction is only available in an Armv7E-M implementation.

The 16-bit instruction is available in T32.

Related references

C1.9 Condition code suffixes on page C1-92

C2.192 WFE

Wait For Event.

Syntax

`WFE{`*cond*`}`

where:

cond

is an optional condition code.

Operation

This is a hint instruction. It is optional whether this instruction is implemented or not. If this instruction is not implemented, it executes as a `NOP`. The assembler produces a diagnostic message if the instruction executes as a `NOP` on the target.

If the Event Register is not set, `WFE` suspends execution until one of the following events occurs:

- An IRQ interrupt, unless masked by the CPSR I-bit.
- An FIQ interrupt, unless masked by the CPSR F-bit.
- An Imprecise Data abort, unless masked by the CPSR A-bit.
- A Debug Entry request, if Debug is enabled.
- An Event signaled by another processor using the `SEV` instruction, or by the current processor using the `SEVL` instruction.

If the Event Register is set, `WFE` clears it and returns immediately.

If `WFE` is implemented, `SEV` must also be implemented.

Availability

This instruction is available in A32 and T32.

Related references

C2.68 NOP on page C2-213
C1.9 Condition code suffixes on page C1-92
C2.103 SEV on page C2-261
C2.104 SEVL on page C2-262
C2.193 WFI on page C2-385

C2.193 WFI

Wait for Interrupt.

Syntax

`WFI{`*cond*`}`

where:

cond

is an optional condition code.

Operation

This is a hint instruction. It is optional whether this instruction is implemented or not. If this instruction is not implemented, it executes as a `NOP`. The assembler produces a diagnostic message if the instruction executes as a `NOP` on the target.

`WFI` suspends execution until one of the following events occurs:

- An IRQ interrupt, regardless of the CPSR I-bit.
- An FIQ interrupt, regardless of the CPSR F-bit.
- An Imprecise Data abort, unless masked by the CPSR A-bit.
- A Debug Entry request, regardless of whether Debug is enabled.

Availability

This instruction is available in A32 and T32.

Related references

C2.68 NOP on page C2-213
C1.9 Condition code suffixes on page C1-92
C2.192 WFE on page C2-384

C2.194 YIELD

Yield.

Syntax

`YIELD{`*cond*`}`

where:

cond

is an optional condition code.

Operation

This is a hint instruction. It is optional whether this instruction is implemented or not. If this instruction is not implemented, it executes as a `NOP`. The assembler produces a diagnostic message if the instruction executes as a `NOP` on the target.

`YIELD` indicates to the hardware that the current thread is performing a task, for example a spinlock, that can be swapped out. Hardware can use this hint to suspend and resume threads in a multithreading system.

Availability

This instruction is available in A32 and T32.

Related references

C2.68 NOP on page C2-213
C1.9 Condition code suffixes on page C1-92

Chapter C3
Advanced SIMD Instructions (32-bit)

Describes Advanced SIMD assembly language instructions.

It contains the following sections:

C3.1 Summary of Advanced SIMD instructions

Most Advanced SIMD instructions are not available in floating-point.

The following table shows a summary of Advanced SIMD instructions that are not available as floating-point instructions:

Table C3-1 Summary of Advanced SIMD instructions

Mnemonic	Brief description
FLDMDBX, FLDMIAX	FLDMX
FSTMDBX, FSTMIAX	FSTMX
VABA, VABD	Absolute difference and Accumulate, Absolute Difference
VABS	Absolute value
VACGE, VACGT	Absolute Compare Greater than or Equal, Greater Than
VACLE, VACLT	Absolute Compare Less than or Equal, Less Than (pseudo-instructions)
VADD	Add
VADDHN	Add, select High half
VAND	Bitwise AND
VAND	Bitwise AND (pseudo-instruction)
VBIC	Bitwise Bit Clear (register)
VBIC	Bitwise Bit Clear (immediate)
VBIF, VBIT, VBSL	Bitwise Insert if False, Insert if True, Select
VCADD	Vector Complex Add
VCEQ, VCLE, VCLT	Compare Equal, Less than or Equal, Compare Less Than
VCGE, VCGT	Compare Greater than or Equal, Greater Than
VCLE, VCLT	Compare Less than or Equal, Compare Less Than (pseudo-instruction)
VCLS, VCLZ, VCNT	Count Leading Sign bits, Count Leading Zeros, and Count set bits
VCMLA	Vector Complex Multiply Accumulate
VCMLA (by element)	Vector Complex Multiply Accumulate (by element)
VCVT	Convert fixed-point or integer to floating-point, floating-point to integer or fixed-point
VCVT	Convert floating-point to integer with directed rounding modes
VCVT	Convert between half-precision and single-precision floating-point numbers
VDUP	Duplicate scalar to all lanes of vector
VEOR	Bitwise Exclusive OR
VEXT	Extract
VFMA, VFMS	Fused Multiply Accumulate, Fused Multiply Subtract
VFMAL, VFMSL	Vector Floating-point Multiply-Add Long to accumulator (by scalar)
VFMAL, VFMSL	Vector Floating-point Multiply-Add Long to accumulator (vector)
VHADD, VHSUB	Halving Add, Halving Subtract

Table C3-1 Summary of Advanced SIMD instructions (continued)

Mnemonic	Brief description
VLD	Vector Load
VMAX, VMIN	Maximum, Minimum
VMAXNM, VMINNM	Maximum, Minimum, consistent with IEEE 754-2008
VMLA, VMLS	Multiply Accumulate, Multiply Subtract (vector)
VMLA, VMLS	Multiply Accumulate, Multiply Subtract (by scalar)
VMOV	Move (immediate)
VMOV	Move (register)
VMOVL, VMOV{U}N	Move Long, Move Narrow (register)
VMUL	Multiply (vector)
VMUL	Multiply (by scalar)
VMVN	Move Negative (immediate)
VNEG	Negate
VORN	Bitwise OR NOT
VORN	Bitwise OR NOT (pseudo-instruction)
VORR	Bitwise OR (register)
VORR	Bitwise OR (immediate)
VPADD, VPADAL	Pairwise Add, Pairwise Add and Accumulate
VPMAX, VPMIN	Pairwise Maximum, Pairwise Minimum
VQABS	Absolute value, saturate
VQADD	Add, saturate
VQDMLAL, VQDMLSL	Saturating Doubling Multiply Accumulate, and Multiply Subtract
VQDMULL	Saturating Doubling Multiply
VQDMULH	Saturating Doubling Multiply returning High half
VQMOV{U}N	Saturating Move (register)
VQNEG	Negate, saturate
VQRDMULH	Saturating Doubling Multiply returning High half
VQRSHL	Shift Left, Round, saturate (by signed variable)
VQRSHR{U}N	Shift Right, Round, saturate (by immediate)
VQSHL	Shift Left, saturate (by immediate)
VQSHL	Shift Left, saturate (by signed variable)
VQSHR{U}N	Shift Right, saturate (by immediate)
VQSUB	Subtract, saturate
VRADDHN	Add, select High half, Round
VRECPE	Reciprocal Estimate

Table C3-1 Summary of Advanced SIMD instructions (continued)

Mnemonic	Brief description
VRECPS	Reciprocal Step
VREV	Reverse elements
VRHADD	Halving Add, Round
VRINT	Round to integer
VRSHR	Shift Right and Round (by immediate)
VRSHRN	Shift Right, Round, Narrow (by immediate)
VRSQRTE	Reciprocal Square Root Estimate
VRSQRTS	Reciprocal Square Root Step
VRSRA	Shift Right, Round, and Accumulate (by immediate)
VRSUBHN	Subtract, select High half, Round
VSDOT (vector)	Dot Product vector form with signed integers
VSDOT (by element)	Dot Product index form with signed integers
VSHL	Shift Left (by immediate)
VSHR	Shift Right (by immediate)
VSHRN	Shift Right, Narrow (by immediate)
VSLI	Shift Left and Insert
VSRA	Shift Right, Accumulate (by immediate)
VSRI	Shift Right and Insert
VST	Vector Store
VSUB	Subtract
VSUBHN	Subtract, select High half
VSWP	Swap vectors
VTBL, VTBX	Vector table look-up
VTRN	Vector transpose
VTST	Test bits
VUDOT (vector)	Dot Product vector form with unsigned integers
VUDOT (by element)	Dot Product index form with unsigned integers
VUZP, VZIP	Vector interleave and de-interleave
VZIP	Vector Zip

C3.2 Summary of shared Advanced SIMD and floating-point instructions

Some instructions are common to Advanced SIMD and floating-point.

The following table shows a summary of instructions that are common to the Advanced SIMD and floating-point instruction sets.

Table C3-2 Summary of shared Advanced SIMD and floating-point instructions

Mnemonic	Brief description
VLDM	Load multiple
VLDR	Load
	Load (post-increment and pre-decrement)
VMOV	Transfer from one general-purpose register to a scalar
	Transfer from two general-purpose registers to either one double-precision or two single-precision registers
	Transfer from a scalar to a general-purpose register
	Transfer from either one double-precision or two single-precision registers to two general-purpose registers
VMRS	Transfer from a SIMD and floating-point system register to a general-purpose register
VMSR	Transfer from a general-purpose register to a SIMD and floating-point system register
VPOP	Pop floating-point or SIMD registers from full-descending stack
VPUSH	Push floating-point or SIMD registers to full-descending stack
VSTM	Store multiple
VSTR	Store
	Store (post-increment and pre-decrement)

Related references

C3.54 VLDM on page C3-449
C3.55 VLDR on page C3-450
C3.56 VLDR (post-increment and pre-decrement) on page C3-451
C3.57 VLDR pseudo-instruction on page C3-452
C3.70 VMOV (between two general-purpose registers and a 64-bit extension register) on page C3-465
C3.71 VMOV (between a general-purpose register and an Advanced SIMD scalar) on page C3-466
C3.75 VMRS on page C3-470
C3.76 VMSR on page C3-471
C3.92 VPOP on page C3-487
C3.93 VPUSH on page C3-488
C3.131 VSTM on page C3-526
C3.134 VSTR on page C3-531
C3.135 VSTR (post-increment and pre-decrement) on page C3-532

C3.3 Interleaving provided by load and store element and structure instructions

Many instructions in this group provide interleaving when structures are stored to memory, and de-interleaving when structures are loaded from memory.

The following figure shows an example of de-interleaving. Interleaving is the inverse process.

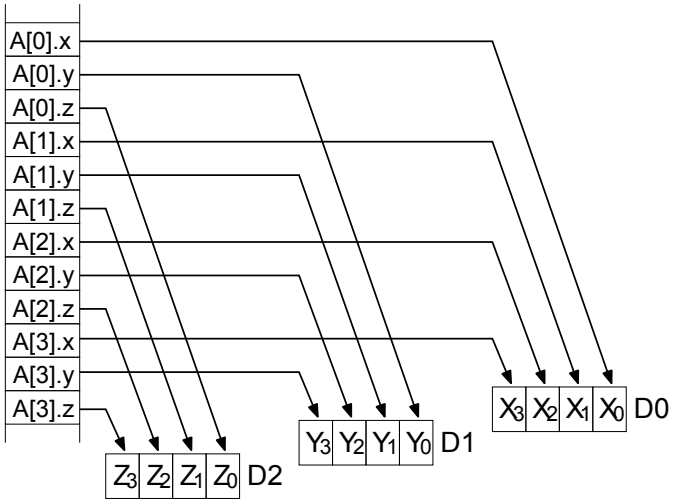


Figure C3-1 De-interleaving an array of 3-element structures

Related concepts

C3.4 Alignment restrictions in load and store element and structure instructions on page C3-396

Related references

C3.51 VLDn (single n-element structure to one lane) on page C3-443

C3.52 VLDn (single n-element structure to all lanes) on page C3-445

C3.53 VLDn (multiple n-element structures) on page C3-447

C3.132 VSTn (multiple n-element structures) on page C3-527

C3.133 VSTn (single n-element structure to one lane) on page C3-529

Related information

Arm Architecture Reference Manual

C3.4 Alignment restrictions in load and store element and structure instructions

Many of these instructions allow you to specify memory alignment restrictions.

When the alignment is not specified in the instruction, the alignment restriction is controlled by the A bit (SCTLR bit[1]):

- If the A bit is 0, there are no alignment restrictions (except for strongly-ordered or device memory, where accesses must be element-aligned).
- If the A bit is 1, accesses must be element-aligned.

If an address is not correctly aligned, an alignment fault occurs.

Related concepts

C3.3 Interleaving provided by load and store element and structure instructions on page C3-395

Related references

C3.51 VLDn (single n-element structure to one lane) on page C3-443

C3.52 VLDn (single n-element structure to all lanes) on page C3-445

C3.53 VLDn (multiple n-element structures) on page C3-447

C3.132 VSTn (multiple n-element structures) on page C3-527

C3.133 VSTn (single n-element structure to one lane) on page C3-529

Related information

Arm Architecture Reference Manual

C3.5 FLDMDBX, FLDMIAX

FLDMX.

Syntax

```
FLDMDBX{c}{q} Rn!, dreglist ; A1 Decrement Before FP/SIMD registers (A32)
FLDMIAX{c}{q} Rn{!}, dreglist ; A1 Increment After FP/SIMD registers (A32)
FLDMDBX{c}{q} Rn!, dreglist ; T1 Decrement Before FP/SIMD registers (T32)
FLDMIAX{c}{q} Rn{!}, dreglist ; T1 Increment After FP/SIMD registers (T32)
```

Where:

c
Is an optional condition code. See *Chapter C1 Condition Codes* on page C1-83.

q
Is an optional instruction width specifier. See *C2.2 Instruction width specifiers* on page C2-111.

Rn
Is the general-purpose base register. If writeback is not specified, the PC can be used.

!
Specifies base register writeback.

dreglist
Is the list of consecutively numbered 64-bit SIMD and FP registers to be transferred. 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.

Usage

FLDMX loads multiple SIMD and 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 mode in which the instruction is executed, an 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 the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Note

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

C3.1 Summary of Advanced SIMD instructions on page C3-391

C3.6 FSTMDBX, FSTMIAX

FSTMX.

Syntax

```
FSTMDBX{c}{q} Rn!, dreglist ; A1 Decrement Before FP/SIMD registers (A32)
FSTMIAX{c}{q} Rn{!}, dreglist ; A1 Increment After FP/SIMD registers (A32)
FSTMDBX{c}{q} Rn!, dreglist ; T1 Decrement Before FP/SIMD registers (T32)
FSTMIAX{c}{q} Rn{!}, dreglist ; T1 Increment After FP/SIMD registers (T32)
```

Where:

c
Is an optional condition code. See *Chapter C1 Condition Codes* on page C1-83.

q
Is an optional instruction width specifier. See *C2.2 Instruction width specifiers* on page C2-111.

Rn
Is the general-purpose base register. If writeback is not specified, the PC can be used. However, Arm deprecates use of the PC.

!
Specifies base register writeback.

dreglist
Is the list of consecutively numbered 64-bit SIMD and FP registers to be transferred. 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.

Usage

FSTMX stores multiple SIMD and 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 FLDMDBX and FLDMIAX, 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 mode in which the instruction is executed, an 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 the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Note

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

C3.1 Summary of Advanced SIMD instructions on page C3-391

C3.7 VABA and VABAL

Vector Absolute Difference and Accumulate.

Syntax

`VABA{cond}.datatype {Qd}, Qn, Qm`

`VABA{cond}.datatype {Dd}, Dn, Dm`

`VABAL{cond}.datatype Qd, Dn, Dm`

where:

cond

is an optional condition code.

datatype

must be one of `S8`, `S16`, `S32`, `U8`, `U16`, or `U32`.

Qd, Qn, Qm

are the destination vector, the first operand vector, and the second operand vector, for a quadword operation.

Dd, Dn, Dm

are the destination vector, the first operand vector, and the second operand vector, for a doubleword operation.

Qd, Dn, Dm

are the destination vector, the first operand vector, and the second operand vector, for a long operation.

Operation

`VABA` 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.

`VABAL` is the long version of the `VABA` instruction.

Related references

C1.9 Condition code suffixes on page C1-92

C3.8 VABD and VABDL

Vector Absolute Difference.

Syntax

`VABD{cond}.datatype {Qd}, Qn, Qm`

`VABD{cond}.datatype {Dd}, Dn, Dm`

`VABDL{cond}.datatype Qd, Dn, Dm`

where:

cond

is an optional condition code.

datatype

must be one of:

- `S8`, `S16`, `S32`, `U8`, `U16`, or `U32` for `VABDL`.
- `S8`, `S16`, `S32`, `U8`, `U16`, `U32` or `F32` for `VABD`.

Qd, Qn, Qm

are the destination vector, the first operand vector, and the second operand vector, for a quadword operation.

Dd, Dn, Dm

are the destination vector, the first operand vector, and the second operand vector, for a doubleword operation.

Qd, Dn, Dm

are the destination vector, the first operand vector, and the second operand vector, for a long operation.

Operation

`VABD` subtracts the elements of one vector from the corresponding elements of another vector, and places the absolute values of the results into the elements of the destination vector.

`VABDL` is the long version of the `VABD` instruction.

Related references

C1.9 Condition code suffixes on page C1-92

C3.9 VABS

Vector Absolute

Syntax

`VABS{cond}.datatype Qd, Qm`

`VABS{cond}.datatype Dd, Dm`

where:

cond

is an optional condition code.

datatype

must be one of `S8`, `S16`, `S32`, or `F32`.

Qd, Qm

are the destination vector and the operand vector, for a quadword operation.

Dd, Dm

are the destination vector and the operand vector, for a doubleword operation.

Operation

`VABS` 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.)

Related references

C3.94 VQABS on page C3-489
C1.9 Condition code suffixes on page C1-92

C3.10 VACLE, VACLT, VACGE and VACGT

Vector Absolute Compare.

Syntax

`VACop{cond}.F32 {Qd}, Qn, Qm`

`VACop{cond}.F32 {Dd}, Dn, Dm`

where:

op

must be one of:

GE

Absolute Greater than or Equal.

GT

Absolute Greater Than.

LE

Absolute Less than or Equal.

LT

Absolute Less Than.

cond

is an optional condition code.

Qd, Qn, Qm

specifies the destination register, the first operand register, and the second operand register, for a quadword operation.

Dd, Dn, Dm

specifies the destination register, the first operand register, and the second operand register, for a doubleword operation.

The result datatype is `I32`.

Operation

These instructions take the absolute value of each element in a vector, and compare it with the absolute value of the corresponding element of a second vector. If the condition is true, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

Note

On disassembly, the `VACLE` and `VACLT` pseudo-instructions are disassembled to the corresponding `VACGE` and `VACGT` instructions, with the operands reversed.

Related references

C1.9 Condition code suffixes on page C1-92

C3.11 VADD

Vector Add.

Syntax

`VADD{cond}.datatype {Qd}, Qn, Qm`

`VADD{cond}.datatype {Dd}, Dn, Dm`

where:

cond

is an optional condition code.

datatype

must be one of `I8`, `I16`, `I32`, `I64`, or `F32`

Qd, Qn, Qm

are the destination vector, the first operand vector, and the second operand vector, for a quadword operation.

Dd, Dn, Dm

are the destination vector, the first operand vector, and the second operand vector, for a doubleword operation.

Operation

`VADD` adds corresponding elements in two vectors, and places the results in the destination vector.

Related references

C3.13 VADDL and VADDW on page C3-405

C3.95 VQADD on page C3-490

C1.9 Condition code suffixes on page C1-92

C3.12 VADDHN

Vector Add and Narrow, selecting High half.

Syntax

`VADDHN{cond}.datatype Dd, Qn, Qm`

where:

cond

is an optional condition code.

datatype

must be one of `I16`, `I32`, or `I64`.

Dd, Qn, Qm

are the destination vector, the first operand vector, and the second operand vector.

Operation

`VADDHN` adds corresponding elements in two vectors, selects the most significant halves of the results, and places the final results in the destination vector. Results are truncated.

Related references

C3.108 VRADDHN on page C3-503

C1.9 Condition code suffixes on page C1-92

C3.13 VADDL and VADDW

Vector Add Long, Vector Add Wide.

Syntax

`VADDL{cond}.datatype Qd, Dn, Dm ; Long operation`

`VADDW{cond}.datatype {Qd,} Qn, Dm ; Wide operation`

where:

cond

is an optional condition code.

datatype

must be one of `S8`, `S16`, `S32`, `U8`, `U16`, or `U32`.

Qd, Dn, Dm

are the destination vector, the first operand vector, and the second operand vector, for a long operation.

Qd, Qn, Dm

are the destination vector, the first operand vector, and the second operand vector, for a wide operation.

Operation

`VADDL` adds corresponding elements in two doubleword vectors, and places the results in the destination quadword vector.

`VADDW` adds corresponding elements in one quadword and one doubleword vector, and places the results in the destination quadword vector.

Related references

C3.11 VADD on page C3-403
C1.9 Condition code suffixes on page C1-92

C3.14 VAND (immediate)

Vector bitwise AND immediate pseudo-instruction.

Syntax

`VAND{`*`cond`*`}.`*`datatype Qd, #imm`*

`VAND{`*`cond`*`}.`*`datatype Dd, #imm`*

where:

`cond`

is an optional condition code.

`datatype`

must be either `I8`, `I16`, `I32`, or `I64`.

`Qd` or *`Dd`*

is the Advanced SIMD register for the result.

`imm`

is the immediate value.

Operation

`VAND` takes each element of the destination vector, performs a bitwise AND with an immediate value, and returns the result into the destination vector.

Note

On disassembly, this pseudo-instruction is disassembled to a corresponding `VBIC` instruction, with the complementary immediate value.

Immediate values

If *`datatype`* is `I16`, the immediate value must have one of the following forms:

- `0xFFXY`.
- `0xXYFF`.

If *`datatype`* is `I32`, the immediate value must have one of the following forms:

- `0xFFFFFFXY`.
- `0xFFFFXYFF`.
- `0xFFXYFFFF`.
- `0xXYFFFFFF`.

Related references

C3.16 VBIC (immediate) on page C3-408
C1.9 Condition code suffixes on page C1-92

C3.15 VAND (register)

Vector bitwise AND.

Syntax

`VAND{cond}{.datatype} {Qd}, Qn, Qm`

`VAND{cond}{.datatype} {Dd}, Dn, Dm`

where:

cond

is an optional condition code.

datatype

is an optional data type. The assembler ignores *datatype*.

Qd, Qn, Qm

specifies the destination register, the first operand register, and the second operand register, for a quadword operation.

Dd, Dn, Dm

specifies the destination register, the first operand register, and the second operand register, for a doubleword operation.

Operation

`VAND` performs a bitwise logical AND between two registers, and places the result in the destination register.

Related references

C1.9 Condition code suffixes on page C1-92

C3.16 VBIC (immediate)

Vector Bit Clear immediate.

Syntax

`VBIC{cond}.datatype Qd, #imm`

`VBIC{cond}.datatype Dd, #imm`

where:

cond

is an optional condition code.

datatype

must be either `I8`, `I16`, `I32`, or `I64`.

Qd or *Dd*

is the Advanced SIMD register for the source and result.

imm

is the immediate value.

Operation

`VBIC` takes each element of the destination vector, performs a bitwise AND complement with an immediate value, and returns the result in the destination vector.

Immediate values

You can either specify *imm* as a pattern which the assembler repeats to fill the destination register, or you can directly specify the immediate value (that conforms to the pattern) in full. The pattern for *imm* depends on *datatype* as shown in the following table:

Table C3-3 Patterns for immediate value in VBIC (immediate)

I16	I32
0x00XY	0x000000XY
0xXY00	0x0000XY00
	0x00XY0000
	0xXY000000

If you use the `I8` or `I64` datatypes, the assembler converts it to either the `I16` or `I32` instruction to match the pattern of *imm*. If the immediate value does not match any of the patterns in the preceding table, the assembler generates an error.

Related references

C3.14 VAND (immediate) on page C3-406
C1.9 Condition code suffixes on page C1-92

C3.17 VBIC (register)

Vector Bit Clear.

Syntax

`VBIC{cond}{.datatype} {Qd}, Qn, Qm`

`VBIC{cond}{.datatype} {Dd}, Dn, Dm`

where:

cond

is an optional condition code.

datatype

is an optional data type. The assembler ignores *datatype*.

Qd, Qn, Qm

specifies the destination register, the first operand register, and the second operand register, for a quadword operation.

Dd, Dn, Dm

specifies the destination register, the first operand register, and the second operand register, for a doubleword operation.

Operation

`VBIC` performs a bitwise logical AND complement between two registers, and places the result in the destination register.

Related references

C1.9 Condition code suffixes on page C1-92

C3.18 VBIF

Vector Bitwise Insert if False.

Syntax

`VBIF{cond}{.datatype} {Qd}, Qn, Qm`

`VBIF{cond}{.datatype} {Dd}, Dn, Dm`

where:

cond

is an optional condition code.

datatype

is an optional datatype. The assembler ignores *datatype*.

Qd, Qn, Qm

specifies the destination register, the first operand register, and the second operand register, for a quadword operation.

Dd, Dn, Dm

specifies the destination register, the first operand register, and the second operand register, for a doubleword operation.

Operation

`VBIF` inserts each bit from the first operand into the destination if the corresponding bit of the second operand is 0, otherwise it leaves the destination bit unchanged.

Related references

C1.9 Condition code suffixes on page C1-92

C3.19 VBIT

Vector Bitwise Insert if True.

Syntax

`VBIT{cond}{.datatype} {Qd}, Qn, Qm`

`VBIT{cond}{.datatype} {Dd}, Dn, Dm`

where:

cond

is an optional condition code.

datatype

is an optional datatype. The assembler ignores *datatype*.

Qd, Qn, Qm

specifies the destination register, the first operand register, and the second operand register, for a quadword operation.

Dd, Dn, Dm

specifies the destination register, the first operand register, and the second operand register, for a doubleword operation.

Operation

`VBIT` inserts each bit from the first operand into the destination if the corresponding bit of the second operand is 1, otherwise it leaves the destination bit unchanged.

Related references

C1.9 Condition code suffixes on page C1-92

C3.20 VBSL

Vector Bitwise Select.

Syntax

`VBSL{cond}{.datatype} {Qd}, Qn, Qm`

`VBSL{cond}{.datatype} {Dd}, Dn, Dm`

where:

cond

is an optional condition code.

datatype

is an optional datatype. The assembler ignores *datatype*.

Qd, Qn, Qm

specifies the destination register, the first operand register, and the second operand register, for a quadword operation.

Dd, Dn, Dm

specifies the destination register, the first operand register, and the second operand register, for a doubleword operation.

Operation

`VBSL` selects each bit for the destination from the first operand if the corresponding bit of the destination is 1, or from the second operand if the corresponding bit of the destination is 0.

Related references

C1.9 Condition code suffixes on page C1-92

C3.21 VCADD

Vector Complex Add.

Syntax

```
VCADD{q}.dt {Dd,} Dn, Dm, #rotate ; A1 64-bit SIMD vector FP/SIMD registers (A32)
VCADD{q}.dt {Qd,} Qn, Qm, #rotate ; A1 128-bit SIMD vector FP/SIMD registers (A32)
```

Where:

Dd
Is the 64-bit name of the SIMD and FP destination register.

Dn
Is the 64-bit name of the first SIMD and FP source register.

Dm
Is the 64-bit name of the second SIMD and FP source register.

Qd
Is the 128-bit name of the SIMD and FP destination register.

Qn
Is the 128-bit name of the first SIMD and FP source register.

Qm
Is the 128-bit name of the second SIMD and FP source register.

q
Is an optional instruction width specifier. See *C2.2 Instruction width specifiers* on page C2-111.

dt
Is the data type for the elements of the vectors, and can be either `F16` or `F32`.

rotate
Is the rotation to be applied to elements in the second SIMD and FP source register, and can be either `90` or `270`.

Architectures supported

Supported in the Armv8.3-A architecture and later.

Usage

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the security state and mode in which the instruction is executed, an 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 the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references
C3.1 Summary of Advanced SIMD instructions on page C3-391

C3.22 VCEQ (immediate #0)

Vector Compare Equal to zero.

Syntax

`VCEQ{cond}.datatype {Qd}, Qn, #0`

`VCEQ{cond}.datatype {Dd}, Dn, #0`

where:

cond

is an optional condition code.

datatype

must be one of `I8`, `I16`, `I32`, or `F32`.

The result datatype is:

- `I32` for operand datatypes `I32` or `F32`.
- `I16` for operand datatype `I16`.
- `I8` for operand datatype `I8`.

Qd, Qn, Qm

specifies the destination register and the operand register, for a quadword operation.

Dd, Dn, Dm

specifies the destination register and the operand register, for a doubleword operation.

#0

specifies a comparison with zero.

Operation

`VCEQ` takes the value of each element in a vector, and compares it with zero. If the condition is true, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

Related references

C1.9 Condition code suffixes on page C1-92

C3.23 VCEQ (register)

Vector Compare Equal.

Syntax

`VCEQ{cond}.datatype {Qd}, Qn, Qm`

`VCEQ{cond}.datatype {Dd}, Dn, Dm`

where:

cond

is an optional condition code.

datatype

must be one of `I8`, `I16`, `I32`, or `F32`.

The result datatype is:

- `I32` for operand datatypes `I32` or `F32`.
- `I16` for operand datatype `I16`.
- `I8` for operand datatype `I8`.

Qd, Qn, Qm

specifies the destination register, the first operand register, and the second operand register, for a quadword operation.

Dd, Dn, Dm

specifies the destination register, the first operand register, and the second operand register, for a doubleword operation.

Operation

`VCEQ` takes the value of each element in a vector, and compares it with the value of the corresponding element of a second vector. If the condition is true, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

Related references

C1.9 Condition code suffixes on page C1-92

C3.24 VCGE (immediate #0)

Vector Compare Greater than or Equal to zero.

Syntax

`VCGE{cond}.datatype {Qd}, Qn, #0`

`VCGE{cond}.datatype {Dd}, Dn, #0`

where:

cond

is an optional condition code.

datatype

must be one of `S8`, `S16`, `S32`, or `F32`.

The result datatype is:

- `I32` for operand datatypes `S32` or `F32`.
- `I16` for operand datatype `S16`.
- `I8` for operand datatype `S8`.

Qd, Qn, Qm

specifies the destination register and the operand register, for a quadword operation.

Dd, Dn, Dm

specifies the destination register and the operand register, for a doubleword operation.

#0

specifies a comparison with zero.

Operation

`VCGE` takes the value of each element in a vector, and compares it with zero. If the condition is true, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

Related references

C1.9 Condition code suffixes on page C1-92

C3.25 VCGE (register)

Vector Compare Greater than or Equal.

Syntax

`VCGE{cond}.datatype {Qd}, Qn, Qm`

`VCGE{cond}.datatype {Dd}, Dn, Dm`

where:

cond

is an optional condition code.

datatype

must be one of `S8`, `S16`, `S32`, `U8`, `U16`, `U32`, or `F32`.

The result datatype is:

- `I32` for operand datatypes `S32`, `U32`, or `F32`.
- `I16` for operand datatypes `S16` or `U16`.
- `I8` for operand datatypes `S8` or `U8`.

Qd, Qn, Qm

specifies the destination register, the first operand register, and the second operand register, for a quadword operation.

Dd, Dn, Dm

specifies the destination register, the first operand register, and the second operand register, for a doubleword operation.

Operation

`VCGE` takes the value of each element in a vector, and compares it with the value of the corresponding element of a second vector. If the condition is true, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

Related references

C1.9 Condition code suffixes on page C1-92

C3.26 VCGT (immediate #0)

Vector Compare Greater Than zero.

Syntax

`VCGT{cond}.datatype {Qd}, Qn, #0`

`VCGT{cond}.datatype {Dd}, Dn, #0`

where:

cond

is an optional condition code.

datatype

must be one of `S8`, `S16`, `S32`, or `F32`.

The result datatype is:

- `I32` for operand datatypes `S32` or `F32`.
- `I16` for operand datatype `S16`.
- `I8` for operand datatype `S8`.

Qd, Qn, Qm

specifies the destination register and the operand register, for a quadword operation.

Dd, Dn, Dm

specifies the destination register and the operand register, for a doubleword operation.

Operation

`VCGT` takes the value of each element in a vector, and compares it with zero. If the condition is true, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

Related references

C1.9 Condition code suffixes on page C1-92

C3.27 VCGT (register)

Vector Compare Greater Than.

Syntax

`VCGT{cond}.datatype {Qd}, Qn, Qm`

`VCGT{cond}.datatype {Dd}, Dn, Dm`

where:

cond

is an optional condition code.

datatype

must be one of `S8`, `S16`, `S32`, `U8`, `U16`, `U32`, or `F32`.

The result datatype is:

- `I32` for operand datatypes `S32`, `U32`, or `F32`.
- `I16` for operand datatypes `S16` or `U16`.
- `I8` for operand datatypes `S8` or `U8`.

Qd, Qn, Qm

specifies the destination register, the first operand register, and the second operand register, for a quadword operation.

Dd, Dn, Dm

specifies the destination register, the first operand register, and the second operand register, for a doubleword operation.

Operation

`VCGT` takes the value of each element in a vector, and compares it with the value of the corresponding element of a second vector. If the condition is true, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

Related references

C1.9 Condition code suffixes on page C1-92

C3.28 VCLE (immediate #0)

Vector Compare Less than or Equal to zero.

Syntax

`VCLE{cond}.datatype {Qd}, Qn, #0`

`VCLE{cond}.datatype {Dd}, Dn, #0`

where:

cond

is an optional condition code.

datatype

must be one of `S8`, `S16`, `S32`, or `F32`.

The result datatype is:

- `I32` for operand datatypes `S32` or `F32`.
- `I16` for operand datatype `S16`.
- `I8` for operand datatype `S8`.

Qd, Qn, Qm

specifies the destination register and the operand register, for a quadword operation.

Dd, Dn, Dm

specifies the destination register and the operand register, for a doubleword operation.

#0

specifies a comparison with zero.

Operation

`VCLE` takes the value of each element in a vector, and compares it with zero. If the condition is true, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

Related references

C1.9 Condition code suffixes on page C1-92

C3.29 VCLS

Vector Count Leading Sign bits.

Syntax

`VCLS{cond}.datatype Qd, Qm`

`VCLS{cond}.datatype Dd, Dm`

where:

cond

is an optional condition code.

datatype

must be one of `S8`, `S16`, or `S32`.

Qd, Qm

are the destination vector and the operand vector, for a quadword operation.

Dd, Dm

are the destination vector and the operand vector, for a doubleword operation.

Operation

`VCLS` 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.

Related references

C1.9 Condition code suffixes on page C1-92

C3.30 VCLE (register)

Vector Compare Less than or Equal pseudo-instruction.

Syntax

`VCLE{cond}.datatype {Qd}, Qn, Qm`

`VCLE{cond}.datatype {Dd}, Dn, Dm`

where:

cond

is an optional condition code.

datatype

must be one of `S8`, `S16`, `S32`, `U8`, `U16`, `U32`, or `F32`.

The result datatype is:

- `I32` for operand datatypes `S32`, `U32`, or `F32`.
- `I16` for operand datatypes `S16` or `U16`.
- `I8` for operand datatypes `S8` or `U8`.

Qd, Qn, Qm

specifies the destination register, the first operand register, and the second operand register, for a quadword operation.

Dd, Dn, Dm

specifies the destination register, the first operand register, and the second operand register, for a doubleword operation.

Operation

`VCLE` takes the value of each element in a vector, and compares it with the value of the corresponding element of a second vector. If the condition is true, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

On disassembly, this pseudo-instruction is disassembled to the corresponding `VCGE` instruction, with the operands reversed.

Related references

C1.9 Condition code suffixes on page C1-92

C3.31 VCLT (immediate #0)

Vector Compare Less Than zero.

Syntax

`VCLT{cond}.datatype {Qd}, Qn, #0`

`VCLT{cond}.datatype {Dd}, Dn, #0`

where:

cond

is an optional condition code.

datatype

must be one of `S8`, `S16`, `S32`, or `F32`.

The result datatype is:

- `I32` for operand datatypes `S32` or `F32`.
- `I16` for operand datatype `S16`.
- `I8` for operand datatype `S8`.

Qd, Qn, Qm

specifies the destination register and the operand register, for a quadword operation.

Dd, Dn, Dm

specifies the destination register and the operand register, for a doubleword operation.

#0

specifies a comparison with zero.

Operation

`VCLT` takes the value of each element in a vector, and compares it with zero. If the condition is true, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

Related references

C1.9 Condition code suffixes on page C1-92

C3.32 VCLT (register)

Vector Compare Less Than.

Syntax

`VCLT{cond}.datatype {Qd}, Qn, Qm`

`VCLT{cond}.datatype {Dd}, Dn, Dm`

where:

cond

is an optional condition code.

datatype

must be one of `S8`, `S16`, `S32`, `U8`, `U16`, `U32`, or `F32`.

The result datatype is:

- `I32` for operand datatypes `S32`, `U32`, or `F32`.
- `I16` for operand datatypes `S16` or `U16`.
- `I8` for operand datatypes `S8` or `U8`.

Qd, Qn, Qm

specifies the destination register, the first operand register, and the second operand register, for a quadword operation.

Dd, Dn, Dm

specifies the destination register, the first operand register, and the second operand register, for a doubleword operation.

Operation

`VCLT` takes the value of each element in a vector, and compares it with the value of the corresponding element of a second vector. If the condition is true, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

Note

On disassembly, this pseudo-instruction is disassembled to the corresponding `VCGT` instruction, with the operands reversed.

Related references

C1.9 Condition code suffixes on page C1-92

C3.33 VCLZ

Vector Count Leading Zeros.

Syntax

`VCLZ{cond}.datatype Qd, Qm`

`VCLZ{cond}.datatype Dd, Dm`

where:

cond

is an optional condition code.

datatype

must be one of `I8`, `I16`, or `I32`.

Qd, Qm

are the destination vector and the operand vector, for a quadword operation.

Dd, Dm

are the destination vector and the operand vector, for a doubleword operation.

Operation

`VCLZ` counts the number of consecutive zeros, starting from the top bit, in each element in a vector, and places the results in a second vector.

Related references

C1.9 Condition code suffixes on page C1-92

C3.34 VCMLA

Vector Complex Multiply Accumulate.

Syntax

```
VCMLA{q}.dt {Dd,} Dn, Dm, #rotate ; 64-bit SIMD vector FP/SIMD registers
VCMLA{q}.dt {Qd,} Qn, Qm, #rotate ; 128-bit SIMD vector FP/SIMD registers
```

Where:

Dd
Is the 64-bit name of the SIMD and FP destination register.

Dn
Is the 64-bit name of the first SIMD and FP source register.

Dm
Is the 64-bit name of the second SIMD and FP source register.

Qd
Is the 128-bit name of the SIMD and FP destination register.

Qn
Is the 128-bit name of the first SIMD and FP source register.

Qm
Is the 128-bit name of the second SIMD and FP source register.

q
Is an optional instruction width specifier. See *C2.2 Instruction width specifiers* on page C2-111.

dt
Is the data type for the elements of the vectors, and can be either `F16` or `F32`.

rotate
Is the rotation to be applied to elements in the second SIMD and FP source register, and can be one of `0`, `90`, `180` or `270`.

Architectures supported

Supported in the Armv8.3-A architecture and later.

Usage

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the security state and mode in which the instruction is executed, an 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 the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references
C3.1 Summary of Advanced SIMD instructions on page C3-391

C3.35 VCMLA (by element)

Vector Complex Multiply Accumulate (by element).

Syntax

```
VCMLA{q}.F16 Dd, Dn, Dm[index], #rotate ; A1 Double,halfprec FP/SIMD registers (A32)
VCMLA{q}.F32 Dd, Dn, Dm[0], #rotate ; A1 Double,singleprec FP/SIMD registers (A32)
VCMLA{q}.F32 Qd, Qn, Dm[0], #rotate ; A1 Quad,singleprec FP/SIMD registers (A32)
VCMLA{q}.F16 Qd, Qn, Dm[index], #rotate ; A1 Halfprec,quad FP/SIMD registers (A32)
```

Where:

Dd
Is the 64-bit name of the SIMD and FP destination register.

Dn
Is the 64-bit name of the first SIMD and FP source register.

Dm
Is the 64-bit name of the second SIMD and FP source register.

index
Is the element index in the range 0 to 1.

Qd
Is the 128-bit name of the SIMD and FP destination register.

Qn
Is the 128-bit name of the first SIMD and FP source register.

q
Is an optional instruction width specifier. See *C2.2 Instruction width specifiers* on page C2-111.

rotate
Is the rotation to be applied to elements in the second SIMD and FP source register, and can be one of `0`, `90`, `180` or `270`.

Architectures supported

Supported in the Armv8.3-A architecture and later.

Usage

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the security state and mode in which the instruction is executed, an 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 the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references
C3.1 Summary of Advanced SIMD instructions on page C3-391

C3.36 VCNT

Vector Count set bits.

Syntax

`VCNT{cond}.datatype Qd, Qm`

`VCNT{cond}.datatype Dd, Dm`

where:

cond

is an optional condition code.

datatype

must be `I8`.

Qd, Qm

are the destination vector and the operand vector, for a quadword operation.

Dd, Dm

are the destination vector and the operand vector, for a doubleword operation.

Operation

`VCNT` counts the number of bits that are one in each element in a vector, and places the results in a second vector.

Related references

C1.9 Condition code suffixes on page C1-92

C3.37 VCVT (between fixed-point or integer, and floating-point)

Vector Convert.

Syntax

`VCVT{cond}.type Qd, Qm {, #fbits}`

`VCVT{cond}.type Dd, Dm {, #fbits}`

where:

cond

is an optional condition code.

type

specifies the data types for the elements of the vectors. It must be one of:

`S32.F32`

Floating-point to signed integer or fixed-point.

`U32.F32`

Floating-point to unsigned integer or fixed-point.

`F32.S32`

Signed integer or fixed-point to floating-point.

`F32.U32`

Unsigned integer or fixed-point to floating-point.

Qd, Qm

specifies the destination vector and the operand vector, for a quadword operation.

Dd, Dm

specifies the destination vector and the operand vector, for a doubleword operation.

fbits

if present, specifies the number of fraction bits in the fixed point number. Otherwise, the conversion is between floating-point and integer. *fbits* must lie in the range 0-32. If *fbits* is omitted, the number of fraction bits is 0.

Operation

VCVT converts each element in a vector in one of the following ways, and places the results in the destination vector:

- From floating-point to integer.
- From integer to floating-point.
- From floating-point to fixed-point.
- From fixed-point to floating-point.

Rounding

Integer or fixed-point to floating-point conversions use round to nearest.

Floating-point to integer or fixed-point conversions use round towards zero.

Related references

C1.9 Condition code suffixes on page C1-92

C3.38 VCVT (between half-precision and single-precision floating-point)

Vector Convert.

Syntax

`VCVT{cond}.F32.F16 Qd, Dm`

`VCVT{cond}.F16.F32 Dd, Qm`

where:

cond
is an optional condition code.

Qd, Dm
specifies the destination vector for the single-precision results and the half-precision operand vector.

Dd, Qm
specifies the destination vector for half-precision results and the single-precision operand vector.

Operation

`VCVT` with half-precision extension, converts each element in a vector in one of the following ways, and places the results in the destination vector:

- From half-precision floating-point to single-precision floating-point (`F32.F16`).
- From single-precision floating-point to half-precision floating-point (`F16.F32`).

Architectures

This instruction is available in Armv8. In earlier architectures, it is only available in NEON systems with the half-precision extension.

Related references

C1.9 Condition code suffixes on page C1-92

C3.39 VCVT (from floating-point to integer with directed rounding modes)

VCVT (Vector Convert) converts each element in a vector from floating-point to signed or unsigned integer, and places the results in the destination vector.

Note

- This instruction is supported only in Armv8.
- You cannot use VCVT with a directed rounding mode inside an IT block.

Syntax

VCVT*mode*.*type Qd, Qm*

VCVT*mode*.type *Dd, Dm*

where:

mode

must be one of:

A

meaning round to nearest, ties away from zero

N

meaning round to nearest, ties to even

P

meaning round towards plus infinity

M

meaning round towards minus infinity.

type

specifies the data types for the elements of the vectors. It must be one of:

S32.F32

floating-point to signed integer

U32.F32

floating-point to unsigned integer.

Qd, Qm

specifies the destination and operand vectors, for a quadword operation.

Dd, Dm

specifies the destination and operand vectors, for a doubleword operation.

C3.40 VCVTB, VCVTT (between half-precision and double-precision)

These instructions convert between half-precision and double-precision floating-point numbers.

The conversion can be done in either of the following ways:

- From half-precision floating-point to double-precision floating-point (`F64.F16`).
- From double-precision floating-point to half-precision floating-point (`F16.F64`).

`VCVTB` uses the bottom half (bits[15:0]) of the single word register to obtain or store the half-precision value.

`VCVTT` uses the top half (bits[31:16]) of the single word register to obtain or store the half-precision value.

Note

These instructions are supported only in Armv8.

Syntax

`VCVTB{`*cond*`}.F64.F16` *Dd, Sm*

`VCVTB{`*cond*`}.F16.F64` *Sd, Dm*

`VCVTT{`*cond*`}.F64.F16` *Dd, Sm*

`VCVTT{`*cond*`}.F16.F64` *Sd, Dm*

where:

cond
: is an optional condition code.

Dd
: is a double-precision register for the result.

Sm
: is a single word register holding the operand.

Sd
: is a single word register for the result.

Dm
: is a double-precision register holding the operand.

Usage

These instructions convert the half-precision value in *Sm* to double-precision and place the result in *Dd*, or the double-precision value in *Dm* to half-precision and place the result in *Sd*.

Floating-point exceptions

These instructions can produce Input Denormal, Invalid Operation, Overflow, Underflow, or Inexact exceptions.

C3.41 VDUP

Vector Duplicate.

Syntax

`VDUP{cond}.size Qd, Dm[x]`

`VDUP{cond}.size Dd, Dm[x]`

`VDUP{cond}.size Qd, Rm`

`VDUP{cond}.size Dd, Rm`

where:

cond

is an optional condition code.

size

must be `8`, `16`, or `32`.

Qd

specifies the destination register for a quadword operation.

Dd

specifies the destination register for a doubleword operation.

Dm`[x]`

specifies the Advanced SIMD scalar.

Rm

specifies the general-purpose register. *Rm* must not be PC.

Operation

`VDUP` duplicates a scalar into every element of the destination vector. The source can be an Advanced SIMD scalar or a general-purpose register.

Related references

C1.9 Condition code suffixes on page C1-92

C3.42 VEOR

Vector Bitwise Exclusive OR.

Syntax

`VEOR{cond}{.datatype} {Qd}, Qn, Qm`

`VEOR{cond}{.datatype} {Dd}, Dn, Dm`

where:

cond

is an optional condition code.

datatype

is an optional data type. The assembler ignores *datatype*.

Qd, Qn, Qm

specifies the destination register, the first operand register, and the second operand register, for a quadword operation.

Dd, Dn, Dm

specifies the destination register, the first operand register, and the second operand register, for a doubleword operation.

Operation

`VEOR` performs a logical exclusive OR between two registers, and places the result in the destination register.

Related references

C1.9 Condition code suffixes on page C1-92

C3.43 VEXT

Vector Extract.

Syntax

`VEXT{cond}.8 {Qd}, Qn, Qm, #imm`

`VEXT{cond}.8 {Dd}, Dn, Dm, #imm`

where:

cond

is an optional condition code.

Qd, Qn, Qm

specifies the destination register, the first operand register, and the second operand register, for a quadword operation.

Dd, Dn, Dm

specifies the destination register, the first operand register, and the second operand register, for a doubleword operation.

imm

is the number of 8-bit elements to extract from the bottom of the second operand vector, in the range 0-7 for doubleword operations, or 0-15 for quadword operations.

Operation

`VEXT` extracts 8-bit 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. See the following figure for an example:

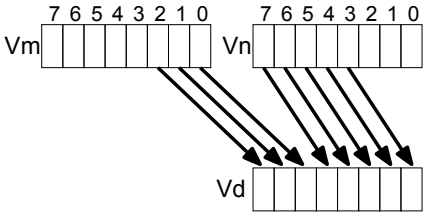


Figure C3-2 Operation of doubleword VEXT for imm = 3

VEXT pseudo-instruction

You can specify a datatype of 16, 32, or 64 instead of 8. In this case, `#imm` refers to halfwords, words, or doublewords instead of referring to bytes, and the permitted ranges are correspondingly reduced.

Related references

C1.9 Condition code suffixes on page C1-92

C3.44 VFMA, VFMS

Vector Fused Multiply Accumulate, Vector Fused Multiply Subtract.

Syntax

`Vop{cond}.F32 {Qd}, Qn, Qm`

`Vop{cond}.F32 {Dd}, Dn, Dm`

where:

op

is one of `FMA` or `FMS`.

cond

is an optional condition code.

Dd, Dn, Dm

are the destination and operand vectors for doubleword operation.

Qd, Qn, Qm

are the destination and operand vectors for quadword operation.

Operation

`VFMA` multiplies corresponding elements in the two operand vectors, and accumulates the results into the elements of the destination vector. The result of the multiply is not rounded before the accumulation.

`VFMS` multiplies corresponding elements in the two operand vectors, then subtracts the products from the corresponding elements of the destination vector, and places the final results in the destination vector. The result of the multiply is not rounded before the subtraction.

Related references

C3.77 VMUL on page C3-472
C1.9 Condition code suffixes on page C1-92

C3.45 VFMAL (by scalar)

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

Syntax

```
VFMAL{q}.F16 Dd, Sn, Sm[index] ; 64-bit SIMD vector
VFMAL{q}.F16 Qd, Dn, Dm[index] ; 128-bit SIMD vector FP/SIMD registers (A32)
```

Where:

Dd
Is the 64-bit name of the SIMD and FP destination register.

Sn
Is the 32-bit name of the first SIMD and FP source register.

Sm
Is the 32-bit name of the second SIMD and FP source register.

index
Depends on the instruction variant:

64
For the 64-bit SIMD vector variant: is the element index in the range 0 to 1.

128
For the 128-bit SIMD vector variant: is the element index in the range 0 to 3.

q
Is an optional instruction width specifier. See *C2.2 Instruction width specifiers* on page C2-111.

Qd
Is the 128-bit name of the SIMD and FP destination register.

Dn
Is the 64-bit name of the first SIMD and FP source register.

Dm
Is the 64-bit name of the second SIMD and FP source register.

Architectures supported

Supported in Armv8.2 and later.

Usage

Vector Floating-point Multiply-Add Long to accumulator (by scalar). This instruction multiplies the vector elements in the first source SIMD and FP register by the specified value in the second source SIMD and FP register, and accumulates the product to the corresponding vector element of the destination SIMD and 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 the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

Note

ID_ISAR6.FHM indicates whether this instruction is supported.

Related references

C3.1 Summary of Advanced SIMD instructions on page C3-391

C3.46 VFMAL (vector)

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

Syntax

```
VFMAL{q}.F16 Dd, Sn, Sm ; 64-bit SIMD vector

VFMAL{q}.F16 Qd, Dn, Dm ; 128-bit SIMD vector FP/SIMD registers (A32)
```

Where:

Dd
Is the 64-bit name of the SIMD and FP destination register.

Sn
Is the 32-bit name of the first SIMD and FP source register.

Sm
Is the 32-bit name of the second SIMD and FP source register.

q
Is an optional instruction width specifier. See *C2.2 Instruction width specifiers* on page C2-111.

Qd
Is the 128-bit name of the SIMD and FP destination register.

Dn
Is the 64-bit name of the first SIMD and FP source register.

Dm
Is the 64-bit name of the second SIMD and FP source register.

Architectures supported

Supported in Armv8.2 and later.

Usage

Vector Floating-point Multiply-Add Long to accumulator (vector). This instruction multiplies corresponding values in the vectors in the two source SIMD and FP registers, and accumulates the product to the corresponding vector element of the destination SIMD and 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 the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

Note

ID_ISAR6.FHM indicates whether this instruction is supported.

Related references

C3.1 Summary of Advanced SIMD instructions on page C3-391

C3.47 VFMSL (by scalar)

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

Syntax

```
VFMSL{q}.F16 Dd, Sn, Sm[index] ; 64-bit SIMD vector
VFMSL{q}.F16 Qd, Dn, Dm[index] ; 128-bit SIMD vector FP/SIMD registers (A32)
```

Where:

Dd
Is the 64-bit name of the SIMD and FP destination register.

Sn
Is the 32-bit name of the first SIMD and FP source register.

Sm
Is the 32-bit name of the second SIMD and FP source register.

index
Depends on the instruction variant:

64
For the 64-bit SIMD vector variant: is the element index in the range 0 to 1.

128
For the 128-bit SIMD vector variant: is the element index in the range 0 to 3.

q
Is an optional instruction width specifier. See *C2.2 Instruction width specifiers* on page C2-111.

Qd
Is the 128-bit name of the SIMD and FP destination register.

Dn
Is the 64-bit name of the first SIMD and FP source register.

Dm
Is the 64-bit name of the second SIMD and FP source register.

Architectures supported

Supported in Armv8.2 and later.

Usage

Vector Floating-point Multiply-Subtract Long from accumulator (by scalar). This instruction multiplies the negated vector elements in the first source SIMD and FP register by the specified value in the second source SIMD and FP register, and accumulates the product to the corresponding vector element of the destination SIMD and 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 the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

Note

ID_ISAR6.FHM indicates whether this instruction is supported.

Related references

C3.1 Summary of Advanced SIMD instructions on page C3-391

C3.48 VFMSL (vector)

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

Syntax

```
VFMSL{q}.F16 Dd, Sn, Sm ; 64-bit SIMD vector

VFMSL{q}.F16 Qd, Dn, Dm ; 128-bit SIMD vector FP/SIMD registers (A32)
```

Where:

Dd
Is the 64-bit name of the SIMD and FP destination register.

Sn
Is the 32-bit name of the first SIMD and FP source register.

Sm
Is the 32-bit name of the second SIMD and FP source register.

q
Is an optional instruction width specifier. See *C2.2 Instruction width specifiers* on page C2-111.

Qd
Is the 128-bit name of the SIMD and FP destination register.

Dn
Is the 64-bit name of the first SIMD and FP source register.

Dm
Is the 64-bit name of the second SIMD and FP source register.

Architectures supported

Supported in Armv8.2 and later.

Usage

Vector Floating-point Multiply-Subtract Long from accumulator (vector). This instruction negates the values in the vector of one SIMD and FP register, multiplies these with the corresponding values in another vector, and accumulates the product to the corresponding vector element of the destination SIMD and 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 the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

Note

ID_ISAR6.FHM indicates whether this instruction is supported.

Related references

C3.1 Summary of Advanced SIMD instructions on page C3-391

C3.49 VHADD

Vector Halving Add.

Syntax

`VHADD{cond}.datatype {Qd}, Qn, Qm`

`VHADD{cond}.datatype {Dd}, Dn, Dm`

where:

cond

is an optional condition code.

datatype

must be one of `S8`, `S16`, `S32`, `U8`, `U16`, or `U32`.

Qd, Qn, Qm

are the destination vector, the first operand vector, and the second operand vector, for a quadword operation.

Dd, Dn, Dm

are the destination vector, the first operand vector, and the second operand vector, for a doubleword operation.

Operation

`VHADD` adds corresponding elements in two vectors, shifts each result right one bit, and places the results in the destination vector. Results are truncated.

Related references

C1.9 Condition code suffixes on page C1-92

C3.50 VHSUB

Vector Halving Subtract.

Syntax

`VHSUB{cond}.datatype {Qd}, Qn, Qm`

`VHSUB{cond}.datatype {Dd}, Dn, Dm`

where:

cond

is an optional condition code.

datatype

must be one of `S8`, `S16`, `S32`, `U8`, `U16`, or `U32`.

Qd, Qn, Qm

are the destination vector, the first operand vector, and the second operand vector, for a quadword operation.

Dd, Dn, Dm

are the destination vector, the first operand vector, and the second operand vector, for a doubleword operation.

Operation

`VHSUB` subtracts the elements of one vector from the corresponding elements of another vector, shifts each result right one bit, and places the results in the destination vector. Results are always truncated.

Related references

C1.9 Condition code suffixes on page C1-92

C3.51 VLDn (single n-element structure to one lane)

Vector Load single *n*-element structure to one lane.

Syntax

`VLDn{cond}.datatype list, [Rn{@align}]{!}`

`VLDn{cond}.datatype list, [Rn{@align}], Rm`

where:

n

must be one of 1, 2, 3, or 4.

cond

is an optional condition code.

datatype

see the following table.

list

is the list of Advanced SIMD registers enclosed in braces, { and }. See the following table for options.

Rn

is the general-purpose register containing the base address. *Rn* cannot be PC.

align

specifies an optional alignment. See the following table for options.

!

if ! is present, *Rn* is updated to (*Rn* + the number of bytes transferred by the instruction). The update occurs after all the loads have taken place.

Rm

is a general-purpose register containing an offset from the base address. If *Rm* is present, the instruction updates *Rn* to (*Rn* + *Rm*) after using the address to access memory. *Rm* cannot be SP or PC.

Operation

VLD*n* loads one *n*-element structure from memory into one or more Advanced SIMD registers. Elements of the register that are not loaded are unaltered.

Table C3-4 Permitted combinations of parameters for VLDn (single n-element structure to one lane)

n	*datatype*	list [ag]	*align* [ah]	alignment
1	8	{Dd[x]}	-	Standard only
	16	{Dd[x]}	@16	2-byte
	32	{Dd[x]}	@32	4-byte
2	8	{Dd[x], D(d+1)[x]}	@16	2-byte

[ag] Every register in the list must be in the range D0-D31.
[ah] *align* can be omitted. In this case, standard alignment rules apply.

Table C3-4 Permitted combinations of parameters for VLDn (single n-element structure to one lane) (continued)

n	*datatype*	list [ag]	*align* [ah]	alignment
	16	{Dd[x], D(d+1)[x]}	@32	4-byte
		{Dd[x], D(d+2)[x]}	@32	4-byte
	32	{Dd[x], D(d+1)[x]}	@64	8-byte
		{Dd[x], D(d+2)[x]}	@64	8-byte
3	8	{Dd[x], D(d+1)[x], D(d+2)[x]}	-	Standard only
	16 or 32	{Dd[x], D(d+1)[x], D(d+2)[x]}	-	Standard only
		{Dd[x], D(d+2)[x], D(d+4)[x]}	-	Standard only
4	8	{Dd[x], D(d+1)[x], D(d+2)[x], D(d+3)[x]}	@32	4-byte
	16	{Dd[x], D(d+1)[x], D(d+2)[x], D(d+3)[x]}	@64	8-byte
		{Dd[x], D(d+2)[x], D(d+4)[x], D(d+6)[x]}	@64	8-byte
	32	{Dd[x], D(d+1)[x], D(d+2)[x], D(d+3)[x]}	@64 or @128	8-byte or 16-byte
		{Dd[x], D(d+2)[x], D(d+4)[x], D(d+6)[x]}	@64 or @128	8-byte or 16-byte

Related concepts

C3.3 Interleaving provided by load and store element and structure instructions on page C3-395

Related references

C1.9 Condition code suffixes on page C1-92

C3.52 VLDn (single n-element structure to all lanes)

Vector Load single *n*-element structure to all lanes.

Syntax

`VLDn{cond}.datatype list, [Rn{@align}]{!}`

`VLDn{cond}.datatype list, [Rn{@align}], Rm`

where:

n

must be one of 1, 2, 3, or 4.

cond

is an optional condition code.

datatype

see the following table.

list

is the list of Advanced SIMD registers enclosed in braces, `{` and `}`. See the following table for options.

Rn

is the general-purpose register containing the base address. *Rn* cannot be PC.

align

specifies an optional alignment. See the following table for options.

!

if ! is present, *Rn* is updated to (*Rn* + the number of bytes transferred by the instruction). The update occurs after all the loads have taken place.

Rm

is a general-purpose register containing an offset from the base address. If *Rm* is present, the instruction updates *Rn* to (*Rn* + *Rm*) after using the address to access memory. *Rm* cannot be SP or PC.

Operation

VLD*n* loads multiple copies of one *n*-element structure from memory into one or more Advanced SIMD registers.

Table C3-5 Permitted combinations of parameters for VLDn (single n-element structure to all lanes)

n	*datatype*	list [ai]	*align* [aj]	alignment
1	8	{Dd[]}	-	Standard only
		{Dd[],D(d+1)[]}	-	Standard only
	16	{Dd[]}	@16	2-byte
		{Dd[],D(d+1)[]}	@16	2-byte

[ai] Every register in the list must be in the range D0-D31.
[aj] *align* can be omitted. In this case, standard alignment rules apply.

Table C3-5 Permitted combinations of parameters for VLDn (single n-element structure to all lanes) (continued)

n	*datatype*	list [ai]	*align* [aj]	alignment
	32	{Dd[]}	@32	4-byte
		{Dd[],D(d+1)[]}	@32	4-byte
2	8	{Dd[], D(d+1)[]}	@8	byte
		{Dd[], D(d+2)[]}	@8	byte
	16	{Dd[], D(d+1)[]}	@16	2-byte
		{Dd[], D(d+2)[]}	@16	2-byte
	32	{Dd[], D(d+1)[]}	@32	4-byte
		{Dd[], D(d+2)[]}	@32	4-byte
3	8, 16, or 32	{Dd[], D(d+1)[], D(d+2)[]}	-	Standard only
		{Dd[], D(d+2)[], D(d+4)[]}	-	Standard only
4	8	{Dd[], D(d+1)[], D(d+2)[], D(d+3)[]}	@32	4-byte
		{Dd[], D(d+2)[], D(d+4)[], D(d+6)[]}	@32	4-byte
	16	{Dd[], D(d+1)[], D(d+2)[], D(d+3)[]}	@64	8-byte
		{Dd[], D(d+2)[], D(d+4)[], D(d+6)[]}	@64	8-byte
	32	{Dd[], D(d+1)[], D(d+2)[], D(d+3)[]}	@64 or @128	8-byte or 16-byte
		{Dd[], D(d+2)[], D(d+4)[], D(d+6)[]}	@64 or @128	8-byte or 16-byte

Related concepts

C3.3 Interleaving provided by load and store element and structure instructions on page C3-395

Related references

C1.9 Condition code suffixes on page C1-92

C3.53 VLDn (multiple n-element structures)

Vector Load multiple *n*-element structures.

Syntax

`VLDn{cond}.datatype list, [Rn{@align}]{!}`

`VLDn{cond}.datatype list, [Rn{@align}], Rm`

where:

n

must be one of 1, 2, 3, or 4.

cond

is an optional condition code.

datatype

see the following table for options.

list

is the list of Advanced SIMD registers enclosed in braces, `{` and `}`. See the following table for options.

Rn

is the general-purpose register containing the base address. *Rn* cannot be PC.

align

specifies an optional alignment. See the following table for options.

!

if ! is present, *Rn* is updated to (*Rn* + the number of bytes transferred by the instruction). The update occurs after all the loads have taken place.

Rm

is a general-purpose register containing an offset from the base address. If *Rm* is present, the instruction updates *Rn* to (*Rn* + *Rm*) after using the address to access memory. *Rm* cannot be SP or PC.

Operation

VLD*n* loads multiple *n*-element structures from memory into one or more Advanced SIMD registers, with de-interleaving (unless *n* == 1). Every element of each register is loaded.

Table C3-6 Permitted combinations of parameters for VLDn (multiple n-element structures)

n	*datatype*	list [ak]	*align* [al]	*alignment*
1	8, 16, 32, or 64	{Dd}	@64	8-byte
		{Dd, D(d+1)}	@64 or @128	8-byte or 16-byte
		{Dd, D(d+1), D(d+2)}	@64	8-byte
		{Dd, D(d+1), D(d+2), D(d+3)}	@64, @128, or @256	8-byte, 16-byte, or 32-byte

[ak] Every register in the list must be in the range D0-D31.
[al] *align* can be omitted. In this case, standard alignment rules apply.

Table C3-6 Permitted combinations of parameters for VLDn (multiple n-element structures) (continued)

n	*datatype*	list [ak]	*align* [al]	*alignment*
2	8, 16, or 32	{Dd, D(d+1)}	@64, @128	8-byte or 16-byte
		{Dd, D(d+2)}	@64, @128	8-byte or 16-byte
		{Dd, D(d+1), D(d+2), D(d+3)}	@64, @128, or @256	8-byte, 16-byte, or 32-byte
3	8, 16, or 32	{Dd, D(d+1), D(d+2)}	@64	8-byte
		{Dd, D(d+2), D(d+4)}	@64	8-byte
4	8, 16, or 32	{Dd, D(d+1), D(d+2), D(d+3)}	@64, @128, or @256	8-byte, 16-byte, or 32-byte
		{Dd, D(d+2), D(d+4), D(d+6)}	@64, @128, or @256	8-byte, 16-byte, or 32-byte

Related concepts

C3.3 Interleaving provided by load and store element and structure instructions on page C3-395

Related references

C1.9 Condition code suffixes on page C1-92

C3.54 VLDM

Extension register load multiple.

Syntax

`VLDM`*`mode`*`{`*`cond`*`} `*`Rn`*`{!}, `*`Registers`*

where:

`mode`

must be one of:

`IA`

meaning Increment address After each transfer. `IA` is the default, and can be omitted.

`DB`

meaning Decrement address Before each transfer.

`EA`

meaning Empty Ascending stack operation. This is the same as `DB` for loads.

`FD`

meaning Full Descending stack operation. This is the same as `IA` for loads.

`cond`

is an optional condition code.

`Rn`

is the general-purpose register holding the base address for the transfer.

`!`

is optional. `!` specifies that the updated base address must be written back to *`Rn`*. If `!` is not specified, *`mode`* must be `IA`.

`Registers`

is a list of consecutive extension registers enclosed in braces, `{` and `}`. The list can be comma-separated, or in range format. There must be at least one register in the list.

You can specify `D` or `Q` registers, but they must not be mixed. The number of registers must not exceed 16 `D` registers, or 8 `Q` registers. If `Q` registers are specified, on disassembly they are shown as `D` registers.

Note

`VPOP` *`Registers`* is equivalent to `VLDM sp!,` *`Registers`*.

You can use either form of this instruction. They both disassemble to `VPOP`.

Related references

C1.9 Condition code suffixes on page C1-92
C4.14 VLDM (floating-point) on page C4-561

C3.55 VLDR

Extension register load.

Syntax

`VLDR{cond}{.64} Dd, [Rn{, #offset}]`

`VLDR{cond}{.64} Dd, label`

where:

cond

is an optional condition code.

Dd

is the extension register to be loaded.

Rn

is the general-purpose register holding the base address for the transfer.

offset

is an optional numeric expression. It must evaluate to a numeric value at assembly time. The value must be a multiple of 4, and lie in the range -1020 to +1020. The value is added to the base address to form the address used for the transfer.

label

is a PC-relative expression.

label must be aligned on a word boundary within ±1KB of the current instruction.

Operation

The `VLDR` instruction loads an extension register from memory.

Two words are transferred.

There is also a `VLDR` pseudo-instruction.

Related references

C3.57 VLDR pseudo-instruction on page C3-452
C1.9 Condition code suffixes on page C1-92
C4.15 VLDR (floating-point) on page C4-562

C3.56 VLDR (post-increment and pre-decrement)

Pseudo-instruction that loads extension registers, with post-increment and pre-decrement forms.

Note

There are also `VLDR` and `VSTR` instructions without post-increment and pre-decrement.

Syntax

```
VLDR{cond}{.64} Dd, [Rn], #offset ; post-increment
VLDR{cond}{.64} Dd, [Rn, #-offset]! ; pre-decrement
```

where:

cond

is an optional condition code.

Dd

is the extension register to load.

Rn

is the general-purpose register holding the base address for the transfer.

offset

is a numeric expression that must evaluate to 8 at assembly time.

Operation

The post-increment instruction increments the base address in the register by the offset value, after the transfer. The pre-decrement instruction decrements the base address in the register by the offset value, and then performs the transfer using the new address in the register. This pseudo-instruction assembles to a `VLDM` instruction.

Related references

C3.54 VLDM on page C3-449

C3.55 VLDR on page C3-450

C1.9 Condition code suffixes on page C1-92

C4.16 VLDR (post-increment and pre-decrement, floating-point) on page C4-563

C3.57 VLDR pseudo-instruction

The VLDR pseudo-instruction loads a constant value into every element of a 64-bit Advanced SIMD vector.

Note

This description is for the VLDR pseudo-instruction only.

Syntax

`VLDR{cond}.datatype Dd,=constant`

where:

cond
: is an optional condition code.

datatype
: must be one of I*n*, S*n*, U*n*, or F32.

n
: must be one of 8, 16, 32, or 64.

Dd
: is the extension register to be loaded.

constant
: is an immediate value of the appropriate type for *datatype*.

Usage

If an instruction (for example, VMOV) is available that can generate the constant directly into the register, the assembler uses it. Otherwise, it generates a doubleword literal pool entry containing the constant and loads the constant using a VLDR instruction.

Related references

C3.55 VLDR on page C3-450

C1.9 Condition code suffixes on page C1-92

C3.57 VLDR pseudo-instruction on page C3-452

C3.58 VMAX and VMIN

Vector Maximum, Vector Minimum.

Syntax

`V`*op*`{`*cond*`}.`*datatype* *Qd*`,` *Qn*`,` *Qm*

`V`*op*`{`*cond*`}.`*datatype* *Dd*`,` *Dn*`,` *Dm*

where:

op

must be either `MAX` or `MIN`.

cond

is an optional condition code.

datatype

must be one of `S8`, `S16`, `S32`, `U8`, `U16`, `U32`, or `F32`.

Qd`,` *Qn*`,` *Qm*

are the destination vector, the first operand vector, and the second operand vector, for a quadword operation.

Dd`,` *Dn*`,` *Dm*

are the destination vector, the first operand vector, and the second operand vector, for a doubleword operation.

Operation

`VMAX` compares corresponding elements in two vectors, and copies the larger of each pair into the corresponding element in the destination vector.

`VMIN` compares corresponding elements in two vectors, and copies the smaller of each pair into the corresponding element in the destination vector.

Floating-point maximum and minimum

max(+0.0, -0.0) = +0.0.

min(+0.0, -0.0) = -0.0

If any input is a NaN, the corresponding result element is the default NaN.

Related references

C3.89 VPADD on page C3-484
C1.9 Condition code suffixes on page C1-92

C3.59 VMAXNM, VMINNM

Vector Minimum, Vector Maximum.

Note

- These instructions are supported only in Armv8.
- You cannot use `VMAXNM` or `VMINNM` inside an IT block.

Syntax

`V`*`op`*`.F32 `*`Qd, Qn, Qm`

`V`*`op`*`.F32 `*`Dd, Dn, Dm`

where:

op
: must be either `MAXNM` or `MINNM`.

Qd, Qn, Qm
: are the destination vector, the first operand vector, and the second operand vector, for a quadword operation.

Dd, Dn, Dm
: are the destination vector, the first operand vector, and the second operand vector, for a doubleword operation.

Operation

`VMAXNM` compares corresponding elements in two vectors, and copies the larger of each pair into the corresponding element in the destination vector.

`VMINNM` compares corresponding elements in two vectors, and copies the smaller of each pair into the corresponding element in the destination vector.

If one of the elements in a pair is a number and the other element is NaN, the corresponding result element is the number. This is consistent with the IEEE 754-2008 standard.

C3.60 VMLA

Vector Multiply Accumulate.

Syntax

`VMLA{cond}.datatype {Qd}, Qn, Qm`

`VMLA{cond}.datatype {Dd}, Dn, Dm`

where:

cond

is an optional condition code.

datatype

must be one of `I8`, `I16`, `I32`, or `F32`.

Qd, Qn, Qm

are the destination vector, the first operand vector, and the second operand vector, for a quadword operation.

Dd, Dn, Dm

are the destination vector, the first operand vector, and the second operand vector, for a doubleword operation.

Operation

`VMLA` multiplies corresponding elements in two vectors, and accumulates the results into the elements of the destination vector.

Related references

C1.9 Condition code suffixes on page C1-92

C3.61 VMLA (by scalar)

Vector Multiply by scalar and Accumulate.

Syntax

`VMLA{cond}.datatype {Qd}, Qn, Dm[x]`

`VMLA{cond}.datatype {Dd}, Dn, Dm[x]`

where:

cond

is an optional condition code.

datatype

must be one of `I16`, `I32`, or `F32`.

Qd, Qn

are the destination vector and the first operand vector, for a quadword operation.

Dd, Dn

are the destination vector and the first operand vector, for a doubleword operation.

Dm[x]

is the scalar holding the second operand.

Operation

`VMLA` multiplies each element in a vector by a scalar, and accumulates the results into the corresponding elements of the destination vector.

Related references

C1.9 Condition code suffixes on page C1-92

C3.62 VMLAL (by scalar)

Vector Multiply by scalar and Accumulate Long.

Syntax

`VMLAL{cond}.datatype Qd, Dn, Dm[x]`

where:

cond

is an optional condition code.

datatype

must be one of `S16`, `S32`, `U16`, or `U32`

Qd, Dn

are the destination vector and the first operand vector, for a long operation.

Dm[x]

is the scalar holding the second operand.

Operation

`VMLAL` multiplies each element in a vector by a scalar, and accumulates the results into the corresponding elements of the destination vector.

Related references

C1.9 Condition code suffixes on page C1-92

C3.63 VMLAL

Vector Multiply Accumulate Long.

Syntax

`VMLAL{cond}.datatype Qd, Dn, Dm`

where:

cond

is an optional condition code.

datatype

must be one of `S8`, `S16`, `S32`, `U8`, `U16`, or `U32`.

Qd, Dn, Dm

are the destination vector, the first operand vector, and the second operand vector, for a long operation.

Operation

`VMLAL` multiplies corresponding elements in two vectors, and accumulates the results into the elements of the destination vector.

Related concepts

B1.8 Polynomial arithmetic over {0,1} on page B1-54

C3.64 VMLS (by scalar)

Vector Multiply by scalar and Subtract.

Syntax

`VMLS{cond}.datatype {Qd}, Qn, Dm[x]`

`VMLS{cond}.datatype {Dd}, Dn, Dm[x]`

where:

cond

is an optional condition code.

datatype

must be one of `I16`, `I32`, or `F32`.

Qd, Qn

are the destination vector and the first operand vector, for a quadword operation.

Dd, Dn

are the destination vector and the first operand vector, for a doubleword operation.

Dm[x]

is the scalar holding the second operand.

Operation

`VMLS` multiplies each element in a vector by a scalar, subtracts the results from the corresponding elements of the destination vector, and places the final results in the destination vector.

Related references

C1.9 Condition code suffixes on page C1-92

C3.65 VMLS

Vector Multiply Subtract.

Syntax

`VMLS{cond}.datatype {Qd}, Qn, Qm`

`VMLS{cond}.datatype {Dd}, Dn, Dm`

where:

cond

is an optional condition code.

datatype

must be one of `I8`, `I16`, `I32`, `F32`.

Qd, Qn, Qm

are the destination vector, the first operand vector, and the second operand vector, for a quadword operation.

Dd, Dn, Dm

are the destination vector, the first operand vector, and the second operand vector, for a doubleword operation.

Operation

`VMLS` multiplies corresponding elements in two vectors, subtracts the results from corresponding elements of the destination vector, and places the final results in the destination vector.

Related references

C1.9 Condition code suffixes on page C1-92

C3.66 VMLSL

Vector Multiply Subtract Long.

Syntax

`VMLSL{cond}.datatype Qd, Dn, Dm`

where:

cond

is an optional condition code.

datatype

must be one of `S8`, `S16`, `S32`, `U8`, `U16`, or `U32`.

Qd, Dn, Dm

are the destination vector, the first operand vector, and the second operand vector, for a long operation.

Operation

`VMLSL` multiplies corresponding elements in two vectors, subtracts the results from corresponding elements of the destination vector, and places the final results in the destination vector.

Related references

C1.9 Condition code suffixes on page C1-92

C3.67 VMLSL (by scalar)

Vector Multiply by scalar and Subtract Long.

Syntax

`VMLSL{cond}.datatype Qd, Dn, Dm[x]`

where:

cond

is an optional condition code.

datatype

must be one of `S16`, `S32`, `U16`, or `U32`.

Qd, Dn

are the destination vector and the first operand vector, for a long operation.

Dm[x]

is the scalar holding the second operand.

Operation

`VMLSL` multiplies each element in a vector by a scalar, subtracts the results from the corresponding elements of the destination vector, and places the final results in the destination vector.

Related references

C1.9 Condition code suffixes on page C1-92

C3.68 VMOV (immediate)

Vector Move.

Syntax

`VMOV{cond}.datatype Qd, #imm`

`VMOV{cond}.datatype Dd, #imm`

where:

cond

is an optional condition code.

datatype

must be one of `I8`, `I16`, `I32`, `I64`, or `F32`.

Qd or *Dd*

is the Advanced SIMD register for the result.

imm

is an immediate value of the type specified by *datatype*. This is replicated to fill the destination register.

Operation

`VMOV` replicates an immediate value in every element of the destination register.

Table C3-7 Available immediate values in VMOV (immediate)

datatype	imm
I8	0xXY
I16	0x00XY, 0xXY00
I32	0x000000XY, 0x0000XY00, 0x00XY0000, 0xXY000000
	0x0000XYFF, 0x00XYFFFF
I64	byte masks, 0xGGHHJJKKLLMMNNPP [am]
F32	floating-point numbers [an]

Related references

C1.9 Condition code suffixes on page C1-92

[am] Each of `0xGG`, `0xHH`, `0xJJ`, `0xKK`, `0xLL`, `0xMM`, `0xNN`, and `0xPP` must be either `0x00` or `0xFF`.

[an] Any number that can be expressed as +/–*n* * 2^{-r}, where *n* and *r* are integers, 16 <= *n* <= 31, 0 <= r <= 7.

C3.69 VMOV (register)

Vector Move.

Syntax

`VMOV{cond}{.datatype} Qd, Qm`

`VMOV{cond}{.datatype} Dd, Dm`

where:

cond

is an optional condition code.

datatype

is an optional datatype. The assembler ignores *datatype*.

Qd, Qm

specifies the destination vector and the source vector, for a quadword operation.

Dd, Dm

specifies the destination vector and the source vector, for a doubleword operation.

Operation

`VMOV` copies the contents of the source register into the destination register.

Related references

C1.9 Condition code suffixes on page C1-92

C3.70 VMOV (between two general-purpose registers and a 64-bit extension register)

Transfer contents between two general-purpose registers and a 64-bit extension register.

Syntax

`VMOV{cond} Dm, Rd, Rn`

`VMOV{cond} Rd, Rn, Dm`

where:

cond

is an optional condition code.

Dm

is a 64-bit extension register.

Rd, Rn

are the general-purpose registers. R*d* and R*n* must not be PC.

Operation

`VMOV Dm, Rd, Rn` transfers the contents of *Rd* into the low half of *Dm*, and the contents of *Rn* into the high half of *Dm*.

`VMOV Rd, Rn, Dm` transfers the contents of the low half of *Dm* into *Rd*, and the contents of the high half of *Dm* into *Rn*.

Related references

C1.9 Condition code suffixes on page C1-92

C3.71 VMOV (between a general-purpose register and an Advanced SIMD scalar)

Transfer contents between a general-purpose register and an Advanced SIMD scalar.

Syntax

`VMOV{cond}{.size} Dn[x], Rd`

`VMOV{cond}{.datatype} Rd, Dn[x]`

where:

cond

is an optional condition code.

size

the data size. Can be `8`, `16`, or `32`. If omitted, *size* is `32`.

datatype

the data type. Can be `U8`, `S8`, `U16`, `S16`, or `32`. If omitted, *datatype* is `32`.

Dn`[x]`

is the Advanced SIMD scalar.

Rd

is the general-purpose register. *Rd* must not be PC.

Operation

`VMOV` *Dn*`[x],` *Rd* transfers the contents of the least significant byte, halfword, or word of *Rd* into *Dn*[x].

`VMOV` *Rd*`,` *Dn*`[x]` transfers the contents of *Dn*[x] into the least significant byte, halfword, or word of *Rd*. The remaining bits of *Rd* are either zero or sign extended.

Related references

C1.9 Condition code suffixes on page C1-92

C3.72 VMOVL

Vector Move Long.

Syntax

`VMOVL{cond}.datatype Qd, Dm`

where:

cond

is an optional condition code.

datatype

must be one of `S8`, `S16`, `S32`, `U8`, `U16`, or `U32`.

Qd, Dm

specifies the destination vector and the operand vector.

Operation

`VMOVL` 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.

Related references

C1.9 Condition code suffixes on page C1-92

C3.73 VMOVN

Vector Move and Narrow.

Syntax

`VMOVN{cond}.datatype Dd, Qm`

where:

cond

is an optional condition code.

datatype

must be one of `I16`, `I32`, or `I64`.

Dd, Qm

specifies the destination vector and the operand vector.

Operation

`VMOVN` copies the least significant half of each element of a quadword vector into the corresponding elements of a doubleword vector.

Related references

C1.9 Condition code suffixes on page C1-92

C3.74 VMOV2

Pseudo-instruction that generates an immediate value and places it in every element of an Advanced SIMD vector, without loading a value from a literal pool.

Syntax

`VMOV2{cond}.datatype Qd, #constant`

`VMOV2{cond}.datatype Dd, #constant`

where:

datatype
must be one of:

- `I8`, `I16`, `I32`, or `I64`.
- `S8`, `S16`, `S32`, or `S64`.
- `U8`, `U16`, `U32`, or `U64`.
- `F32`.

cond
is an optional condition code.

Qd or *Dd*
is the extension register to be loaded.

constant
is an immediate value of the appropriate type for *datatype*.

Operation

`VMOV2` can generate any 16-bit immediate value, and a restricted range of 32-bit and 64-bit immediate values.

`VMOV2` is a pseudo-instruction that always assembles to exactly two instructions. It typically assembles to a `VMOV` or `VMVN` instruction, followed by a `VBIC` or `VORR` instruction.

Related references

C3.68 VMOV (immediate) on page C3-463
C3.16 VBIC (immediate) on page C3-408
C1.9 Condition code suffixes on page C1-92

C3.75 VMRS

Transfer contents from an Advanced SIMD system register to a general-purpose register.

Syntax

`VMRS{`*`cond`*`} `*`Rd, extsysreg`

where:

`cond`
: is an optional condition code.

`extsysreg`
: is the Advanced SIMD and floating-point system register, usually `FPSCR`, `FPSID`, or `FPEXC`.

`Rd`
: is the general-purpose register. *`Rd`* must not be PC.

  It can be `APSR_nzcv`, if *`extsysreg`* is `FPSCR`. In this case, the floating-point status flags are transferred into the corresponding flags in the special-purpose APSR.

Usage

The `VMRS` instruction transfers the contents of *`extsysreg`* into *`Rd`*.

Note

The instruction stalls the processor until all current Advanced SIMD or floating-point operations complete.

Example

```
    VMRS    r2,FPCID
    VMRS    APSR_nzcv, FPSCR     ; transfer FP status register to the
                                 ; special-purpose APSR
```

Related references

B1.14 Advanced SIMD system registers in AArch32 state on page B1-60

C1.9 Condition code suffixes on page C1-92

C4.26 VMRS (floating-point) on page C4-573

C3.76 VMSR

Transfer contents of a general-purpose register to an Advanced SIMD system register.

Syntax

`VMSR{`*`cond`*`} `*`extsysreg, Rd`

where:

`cond`
is an optional condition code.

`extsysreg`
is the Advanced SIMD and floating-point system register, usually `FPSCR`, `FPSID`, or `FPEXC`.

`Rd`
is the general-purpose register. *`Rd`* must not be PC.

It can be `APSR_nzcv`, if *`extsysreg`* is `FPSCR`. In this case, the floating-point status flags are transferred into the corresponding flags in the special-purpose APSR.

Usage

The `VMSR` instruction transfers the contents of *`Rd`* into *`extsysreg`*.

Note

The instruction stalls the processor until all current Advanced SIMD operations complete.

Example

```
    VMSR    FPSCR, r4
```

Related references

B1.14 Advanced SIMD system registers in AArch32 state on page B1-60
C1.9 Condition code suffixes on page C1-92
C4.27 VMSR (floating-point) on page C4-574

C3.77 VMUL

Vector Multiply.

Syntax

`VMUL{cond}.datatype {Qd}, Qn, Qm`

`VMUL{cond}.datatype {Dd}, Dn, Dm`

where:

cond

is an optional condition code.

datatype

must be one of `I8`, `I16`, `I32`, `F32`, or `P8`.

Qd, Qn, Qm

are the destination vector, the first operand vector, and the second operand vector, for a quadword operation.

Dd, Dn, Dm

are the destination vector, the first operand vector, and the second operand vector, for a doubleword operation.

Operation

`VMUL` multiplies corresponding elements in two vectors, and places the results in the destination vector.

Related references

C1.9 Condition code suffixes on page C1-92

C3.78 VMUL (by scalar)

Vector Multiply by scalar.

Syntax

`VMUL{cond}.datatype {Qd}, Qn, Dm[x]`

`VMUL{cond}.datatype {Dd}, Dn, Dm[x]`

where:

cond

is an optional condition code.

datatype

must be one of `I16`, `I32`, or `F32`.

Qd, Qn

are the destination vector and the first operand vector, for a quadword operation.

Dd, Dn

are the destination vector and the first operand vector, for a doubleword operation.

Dm[x]

is the scalar holding the second operand.

Operation

`VMUL` multiplies each element in a vector by a scalar, and places the results in the destination vector.

Related references

C1.9 Condition code suffixes on page C1-92

C3.79 VMULL

Vector Multiply Long

Syntax

`VMULL{cond}.datatype Qd, Dn, Dm`

where:

cond

is an optional condition code.

datatype

must be one of `U8`, `U16`, `U32`, `S8`, `S16`, `S32`, or `P8`.

Qd, Dn, Dm

are the destination vector, the first operand vector, and the second operand vector, for a long operation.

Operation

`VMULL` multiplies corresponding elements in two vectors, and places the results in the destination vector.

Related references

C1.9 Condition code suffixes on page C1-92

C3.80 VMULL (by scalar)

Vector Multiply Long by scalar

Syntax

`VMULL{cond}.datatype Qd, Dn, Dm[x]`

where:

cond

is an optional condition code.

datatype

must be one of `S16`, `S32`, `U16`, or `U32`.

Qd, Dn

are the destination vector and the first operand vector, for a long operation.

Dm[x]

is the scalar holding the second operand.

Operation

`VMULL` multiplies each element in a vector by a scalar, and places the results in the destination vector.

Related references

C1.9 Condition code suffixes on page C1-92

C3.81 VMVN (register)

Vector Move NOT (register).

Syntax

`VMVN{cond}{.datatype} Qd, Qm`

`VMVN{cond}{.datatype} Dd, Dm`

where:

cond

is an optional condition code.

datatype

is an optional datatype. The assembler ignores *datatype*.

Qd, Qm

specifies the destination vector and the source vector, for a quadword operation.

Dd, Dm

specifies the destination vector and the source vector, for a doubleword operation.

Operation

`VMVN` inverts the value of each bit from the source register and places the results into the destination register.

Related references

C1.9 Condition code suffixes on page C1-92

C3.82 VMVN (immediate)

Vector Move NOT (immediate).

Syntax

`VMVN{cond}.datatype Qd, #imm`

`VMVN{cond}.datatype Dd, #imm`

where:

cond

is an optional condition code.

datatype

must be one of `I8`, `I16`, `I32`, `I64`, or `F32`.

Qd or *Dd*

is the Advanced SIMD register for the result.

imm

is an immediate value of the type specified by *datatype*. This is replicated to fill the destination register.

Operation

`VMVN` inverts the value of each bit from an immediate value and places the results into each element in the destination register.

Table C3-8 Available immediate values in VMVN (immediate)

datatype	imm
I8	-
I16	0xFFXY, 0xXYFF
I32	0xFFFFFFXY, 0xFFFFXYFF, 0xFFXYFFFF, 0xXYFFFFFF
	0xFFFFXY00, 0xFFXY0000
I64	-
F32	-

Related references

C1.9 Condition code suffixes on page C1-92

C3.83 VNEG

Vector Negate.

Syntax

`VNEG{`*cond*`}.`*datatype* *Qd, Qm*

`VNEG{`*cond*`}.`*datatype* *Dd, Dm*

where:

cond

is an optional condition code.

datatype

must be one of `S8`, `S16`, `S32`, or `F32`.

Qd, Qm

are the destination vector and the operand vector, for a quadword operation.

Dd, Dm

are the destination vector and the operand vector, for a doubleword operation.

Operation

`VNEG` negates each element in a vector, and places the results in a second vector. (The floating-point version only inverts the sign bit.)

Related references

C4.29 VNEG (floating-point) on page C4-576
C1.9 Condition code suffixes on page C1-92

C3.84 VORN (register)

Vector bitwise OR NOT (register).

Syntax

`VORN{cond}{.datatype} {Qd}, Qn, Qm`

`VORN{cond}{.datatype} {Dd}, Dn, Dm`

where:

cond

is an optional condition code.

datatype

is an optional data type. The assembler ignores *datatype*.

Qd, Qn, Qm

specifies the destination register, the first operand register, and the second operand register, for a quadword operation.

Dd, Dn, Dm

specifies the destination register, the first operand register, and the second operand register, for a doubleword operation.

Operation

`VORN` performs a bitwise logical OR complement between two registers, and places the results in the destination register.

Related references

C1.9 Condition code suffixes on page C1-92

C3.85 VORN (immediate)

Vector bitwise OR NOT (immediate) pseudo-instruction.

Syntax

`VORN{`*cond*`}.`*datatype* *Qd*`, #`*imm*

`VORN{`*cond*`}.`*datatype* *Dd*`, #`*imm*

where:

cond

is an optional condition code.

datatype

must be either `I8`, `I16`, `I32`, or `I64`.

Qd or *Dd*

is the Advanced SIMD register for the result.

imm

is the immediate value.

Operation

`VORN` takes each element of the destination vector, performs a bitwise OR complement with an immediate value, and returns the results in the destination vector.

Note

On disassembly, this pseudo-instruction is disassembled to a corresponding `VORR` instruction, with a complementary immediate value.

Immediate values

If *datatype* is `I16`, the immediate value must have one of the following forms:

- `0xFFXY`.
- `0xXYFF`.

If *datatype* is `I32`, the immediate value must have one of the following forms:

- `0xFFFFFFXY`.
- `0xFFFFXYFF`.
- `0xFFXYFFFF`.
- `0xXYFFFFFF`.

Related references

C3.87 VORR (immediate) on page C3-482
C1.9 Condition code suffixes on page C1-92

C3.86 VORR (register)

Vector bitwise OR (register).

Syntax

`VORR{cond}{.datatype} {Qd}, Qn, Qm`

`VORR{cond}{.datatype} {Dd}, Dn, Dm`

where:

cond

is an optional condition code.

datatype

is an optional data type. The assembler ignores *datatype*.

Qd, Qn, Qm

specifies the destination register, the first operand register, and the second operand register, for a quadword operation.

Dd, Dn, Dm

specifies the destination register, the first operand register, and the second operand register, for a doubleword operation.

Note

`VORR` with the same register for both operands is a `VMOV` instruction. You can use `VORR` in this way, but disassembly of the resulting code produces the `VMOV` syntax.

Operation

`VORR` performs a bitwise logical OR between two registers, and places the result in the destination register.

Related references

C3.69 VMOV (register) on page C3-464

C1.9 Condition code suffixes on page C1-92

C3.87 VORR (immediate)

Vector bitwise OR immediate.

Syntax

`VORR{cond}.datatype Qd, #imm`

`VORR{cond}.datatype Dd, #imm`

where:

cond

is an optional condition code.

datatype

must be either `I8`, `I16`, `I32`, or `I64`.

Qd or *Dd*

is the Advanced SIMD register for the source and result.

imm

is the immediate value.

Operation

`VORR` takes each element of the destination vector, performs a bitwise logical OR with an immediate value, and places the results in the destination vector.

Immediate values

You can either specify *imm* as a pattern which the assembler repeats to fill the destination register, or you can directly specify the immediate value (that conforms to the pattern) in full. The pattern for *imm* depends on the datatype, as shown in the following table:

Table C3-9 Patterns for immediate value in VORR (immediate)

I16	I32
0x00XY	0x000000XY
0xXY00	0x0000XY00
-	0x00XY0000
-	0xXY000000

If you use the `I8` or `I64` datatypes, the assembler converts it to either the `I16` or `I32` instruction to match the pattern of *imm*. If the immediate value does not match any of the patterns in the preceding table, the assembler generates an error.

Related references

C1.9 Condition code suffixes on page C1-92

C3.88 VPADAL

Vector Pairwise Add and Accumulate Long.

Syntax

`VPADAL{cond}.datatype Qd, Qm`

`VPADAL{cond}.datatype Dd, Dm`

where:

cond

is an optional condition code.

datatype

must be one of `S8`, `S16`, `S32`, `U8`, `U16`, or `U32`.

Qd, Qm

are the destination vector and the operand vector, for a quadword instruction.

Dd, Dm

are the destination vector and the operand vector, for a doubleword instruction.

Operation

`VPADAL` adds adjacent pairs of elements of a vector, and accumulates the absolute values of the results into the elements of the destination vector.

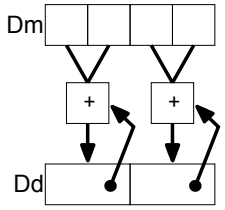


Figure C3-3 Example of operation of VPADAL (in this case for data type S16)

Related references

C1.9 Condition code suffixes on page C1-92

C3.89 VPADD

Vector Pairwise Add.

Syntax

`VPADD{cond}.datatype {Dd}, Dn, Dm`

where:

cond

is an optional condition code.

datatype

must be one of `I8`, `I16`, `I32`, or `F32`.

Dd, Dn, Dm

are the destination vector, the first operand vector, and the second operand vector.

Operation

`VPADD` adds adjacent pairs of elements of two vectors, and places the results in the destination vector.

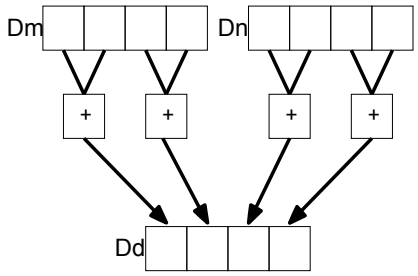


Figure C3-4 Example of operation of VPADD (in this case, for data type I16)

Related references

C1.9 Condition code suffixes on page C1-92

C3.90 VPADDL

Vector Pairwise Add Long.

Syntax

`VPADDL{cond}.datatype Qd, Qm`

`VPADDL{cond}.datatype Dd, Dm`

where:

cond

is an optional condition code.

datatype

must be one of `S8`, `S16`, `S32`, `U8`, `U16`, or `U32`.

Qd, Qm

are the destination vector and the operand vector, for a quadword instruction.

Dd, Dm

are the destination vector and the operand vector, for a doubleword instruction.

Operation

`VPADDL` adds adjacent pairs of elements of a vector, sign or zero extends the results to twice their original width, and places the final results in the destination vector.

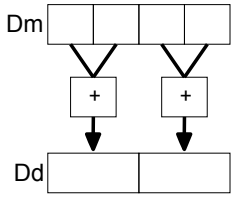


Figure C3-5 Example of operation of doubleword VPADDL (in this case, for data type S16)

Related references

C1.9 Condition code suffixes on page C1-92

C3.91 VPMAX and VPMIN

Vector Pairwise Maximum, Vector Pairwise Minimum.

Syntax

`VPop{cond}.datatype Dd, Dn, Dm`

where:

op

must be either `MAX` or `MIN`.

cond

is an optional condition code.

datatype

must be one of `S8`, `S16`, `S32`, `U8`, `U16`, `U32`, or `F32`.

Dd, Dn, Dm

are the destination doubleword vector, the first operand doubleword vector, and the second operand doubleword vector.

Operation

`VPMAX` compares adjacent pairs of elements in two vectors, and copies the larger of each pair into the corresponding element in the destination vector. Operands and results must be doubleword vectors.

`VPMIN` compares adjacent pairs of elements in two vectors, and copies the smaller of each pair into the corresponding element in the destination vector. Operands and results must be doubleword vectors.

Floating-point maximum and minimum

max(+0.0, -0.0) = +0.0.

min(+0.0, -0.0) = -0.0

If any input is a NaN, the corresponding result element is the default NaN.

Related references

C1.9 Condition code suffixes on page C1-92

C3.92 VPOP

Pop extension registers from the stack.

Syntax

`VPOP{cond} Registers`

where:

cond

is an optional condition code.

Registers

is a list of consecutive extension registers enclosed in braces, `{` and `}`. The list can be comma-separated, or in range format. There must be at least one register in the list.

You can specify `D` or `Q` registers, but they must not be mixed. The number of registers must not exceed 16 `D` registers, or 8 `Q` registers. If `Q` registers are specified, on disassembly they are shown as `D` registers.

Note

`VPOP Registers` is equivalent to `VLDM sp!, Registers`.

You can use either form of this instruction. They both disassemble to `VPOP`.

Related references

C1.9 Condition code suffixes on page C1-92
C3.93 VPUSH on page C3-488
C4.33 VPOP (floating-point) on page C4-580

C3.93 VPUSH

Push extension registers onto the stack.

Syntax

`VPUSH{cond} Registers`

where:

cond

is an optional condition code.

Registers

is a list of consecutive extension registers enclosed in braces, `{` and `}`. The list can be comma-separated, or in range format. There must be at least one register in the list.

You can specify `D` or `Q` registers, but they must not be mixed. The number of registers must not exceed 16 `D` registers, or 8 `Q` registers. If `Q` registers are specified, on disassembly they are shown as `D` registers.

Note

`VPUSH Registers` is equivalent to `VSTMDB sp!, Registers`.

You can use either form of this instruction. They both disassemble to `VPUSH`.

Related references

C1.9 Condition code suffixes on page C1-92
C3.92 VPOP on page C3-487
C4.34 VPUSH (floating-point) on page C4-581

C3.94 VQABS

Vector Saturating Absolute.

Syntax

`VQABS{cond}.datatype Qd, Qm`

`VQABS{cond}.datatype Dd, Dm`

where:

cond

is an optional condition code.

datatype

must be one of `S8`, `S16`, or `S32`.

Qd, Qm

are the destination vector and the operand vector, for a quadword operation.

Dd, Dm

are the destination vector and the operand vector, for a doubleword operation.

Operation

`VQABS` takes the absolute value of each element in a vector, and places the results in a second vector.

The sticky QC flag (FPSCR bit[27]) is set if saturation occurs.

Related references

C1.9 Condition code suffixes on page C1-92

C3.95 VQADD

Vector Saturating Add.

Syntax

`VQADD{cond}.datatype {Qd}, Qn, Qm`

`VQADD{cond}.datatype {Dd}, Dn, Dm`

where:

cond

is an optional condition code.

datatype

must be one of `S8`, `S16`, `S32`, `S64`, `U8`, `U16`, `U32`, or `U64`.

Qd, Qn, Qm

are the destination vector, the first operand vector, and the second operand vector, for a quadword operation.

Dd, Dn, Dm

are the destination vector, the first operand vector, and the second operand vector, for a doubleword operation.

Operation

`VQADD` adds corresponding elements in two vectors, and places the results in the destination vector.

The sticky QC flag (FPSCR bit[27]) is set if saturation occurs.

Related references

C1.9 Condition code suffixes on page C1-92

C3.96 VQDMLAL and VQDMLSL (by vector or by scalar)

Vector Saturating Doubling Multiply Accumulate Long, Vector Saturating Doubling Multiply Subtract Long.

Syntax

`VQDopL{cond}.datatype Qd, Dn, Dm`

`VQDopL{cond}.datatype Qd, Dn, Dm[x]`

where:

op

must be one of:

MLA

Multiply Accumulate.

MLS

Multiply Subtract.

cond

is an optional condition code.

datatype

must be either `S16` or `S32`.

Qd, Dn

are the destination vector and the first operand vector.

Dm

is the vector holding the second operand, for a *by vector* operation.

Dm[x]

is the scalar holding the second operand, for a *by scalar* operation.

Operation

These instructions multiply their operands and double the results. `VQDMLAL` adds the results to the values in the destination register. `VQDMLSL` subtracts the results from the values in the destination register.

If any of the results overflow, they are saturated. The sticky QC flag (FPSCR bit[27]) is set if saturation occurs.

Related references

C1.9 Condition code suffixes on page C1-92

C3.97 VQDMULH (by vector or by scalar)

Vector Saturating Doubling Multiply Returning High Half.

Syntax

`VQDMULH{cond}.datatype {Qd}, Qn, Qm`

`VQDMULH{cond}.datatype {Dd}, Dn, Dm`

`VQDMULH{cond}.datatype {Qd}, Qn, Dm[x]`

`VQDMULH{cond}.datatype {Dd}, Dn, Dm[x]`

where:

cond

is an optional condition code.

datatype

must be either `S16` or `S32`.

Qd, Qn

are the destination vector and the first operand vector, for a quadword operation.

Dd, Dn

are the destination vector and the first operand vector, for a doubleword operation.

Qm or *Dm*

is the vector holding the second operand, for a *by vector* operation.

Dm[*x*]

is the scalar holding the second operand, for a *by scalar* operation.

Operation

`VQDMULH` multiplies corresponding elements in two vectors, doubles the results, and places the most significant half of the final results in the destination vector.

The second operand can be a scalar instead of a vector.

If any of the results overflow, they are saturated. The sticky QC flag (FPSCR bit[27]) is set if saturation occurs. Each result is truncated.

Related references

C1.9 Condition code suffixes on page C1-92

C3.98 VQDMULL (by vector or by scalar)

Vector Saturating Doubling Multiply Long.

Syntax

`VQDMULL{cond}.datatype Qd, Dn, Dm`

`VQDMULL{cond}.datatype Qd, Dn, Dm[x]`

where:

cond

is an optional condition code.

datatype

must be either `S16` or `S32`.

Qd, Dn

are the destination vector and the first operand vector.

Dm

is the vector holding the second operand, for a *by vector* operation.

Dm`[x]`

is the scalar holding the second operand, for a *by scalar* operation.

Operation

`VQDMULL` multiplies corresponding elements in two vectors, doubles the results and places the results in the destination register.

The second operand can be a scalar instead of a vector.

If any of the results overflow, they are saturated. The sticky QC flag (FPSCR bit[27]) is set if saturation occurs.

Related references

C1.9 Condition code suffixes on page C1-92

C3.99 VQMOVN and VQMOVUN

Vector Saturating Move and Narrow.

Syntax

`VQMOVN{cond}.datatype Dd, Qm`

`VQMOVUN{cond}.datatype Dd, Qm`

where:

cond

is an optional condition code.

datatype

must be one of:

S16, S32, S64

for VQMOVN or VQMOVUN.

U16, U32, U64

for VQMOVN.

Dd, Qm

specifies the destination vector and the operand vector.

Operation

VQMOVN copies each element of the operand vector to the corresponding element of the destination vector. The result element is half the width of the operand element, and values are saturated to the result width. The results are the same type as the operands.

VQMOVUN copies each element of the operand vector to the corresponding element of the destination vector. The result element is half the width of the operand element, and values are saturated to the result width. The elements in the operand are signed and the elements in the result are unsigned.

Related references

C1.9 Condition code suffixes on page C1-92

C3.100 VQNEG

Vector Saturating Negate.

Syntax

`VQNEG{cond}.datatype Qd, Qm`

`VQNEG{cond}.datatype Dd, Dm`

where:

cond

is an optional condition code.

datatype

must be one of `S8`, `S16`, or `S32`.

Qd, Qm

are the destination vector and the operand vector, for a quadword operation.

Dd, Dm

are the destination vector and the operand vector, for a doubleword operation.

Operation

`VQNEG` negates each element in a vector, and places the results in a second vector.

The sticky QC flag (FPSCR bit[27]) is set if saturation occurs.

Related references

C1.9 Condition code suffixes on page C1-92

C3.101 VQRDMULH (by vector or by scalar)

Vector Saturating Rounding Doubling Multiply Returning High Half.

Syntax

`VQRDMULH{cond}.datatype {Qd}, Qn, Qm`

`VQRDMULH{cond}.datatype {Dd}, Dn, Dm`

`VQRDMULH{cond}.datatype {Qd}, Qn, Dm[x]`

`VQRDMULH{cond}.datatype {Dd}, Dn, Dm[x]`

where:

cond

is an optional condition code.

datatype

must be either `S16` or `S32`.

Qd, Qn

are the destination vector and the first operand vector, for a quadword operation.

Dd, Dn

are the destination vector and the first operand vector, for a doubleword operation.

Qm or *Dm*

is the vector holding the second operand, for a *by vector* operation.

Dm[*x*]

is the scalar holding the second operand, for a *by scalar* operation.

Operation

`VQRDMULH` multiplies corresponding elements in two vectors, doubles the results, and places the most significant half of the final results in the destination vector.

The second operand can be a scalar instead of a vector.

If any of the results overflow, they are saturated. The sticky QC flag (FPSCR bit[27]) is set if saturation occurs. Each result is rounded.

Related references

C1.9 Condition code suffixes on page C1-92

C3.102 VQRSHL (by signed variable)

Vector Saturating Rounding Shift Left by signed variable.

Syntax

`VQRSHL{cond}.datatype {Qd}, Qm, Qn`

`VQRSHL{cond}.datatype {Dd}, Dm, Dn`

where:

cond

is an optional condition code.

datatype

must be one of `S8`, `S16`, `S32`, `S64`, `U8`, `U16`, `U32`, or `U64`.

Qd, Qm, Qn

are the destination vector, the first operand vector, and the second operand vector, for a quadword operation.

Dd, Dm, Dn

are the destination vector, the first operand vector, and the second operand vector, for a doubleword operation.

Operation

`VQRSHL` 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 rounding right shift.

The sticky QC flag (FPSCR bit[27]) is set if saturation occurs.

Related references

C1.9 Condition code suffixes on page C1-92

C3.103 VQRSHRN and VQRSHRUN (by immediate)

Vector Saturating Shift Right, Narrow, by immediate value, with Rounding.

Syntax

`VQRSHR{U}N{`*cond*`}.`*datatype* *Dd, Qm, #imm*

where:

U

if present, indicates that the results are unsigned, although the operands are signed. Otherwise, the results are the same type as the operands.

cond

is an optional condition code.

datatype

must be one of:

I16, I32, I64

for VQRSHRN or VQRSHRUN. Only a `#0` immediate is permitted with these datatypes.

S16, S32, S64

for VQRSHRN or VQRSHRUN.

U16, U32, U64

for VQRSHRN only.

Dd, Qm

are the destination vector and the operand vector.

imm

is the immediate value specifying the size of the shift. The ranges are shown in the following table:

Table C3-10 Available immediate ranges in VQRSHRN and VQRSHRUN (by immediate)

datatype	imm range
S16 or U16	0 to 8
S32 or U32	0 to 16
S64 or U64	0 to 32

Operation

VQRSHR{U}N takes each element in a quadword vector of integers, right shifts them by an immediate value, and places the results in a doubleword vector.

The sticky QC flag (FPSCR bit[27]) is set if saturation occurs.

Results are rounded.

Related references

C1.9 Condition code suffixes on page C1-92

C3.104 VQSHL (by signed variable)

Vector Saturating Shift Left by signed variable.

Syntax

`VQSHL{cond}.datatype {Qd}, Qm, Qn`

`VQSHL{cond}.datatype {Dd}, Dm, Dn`

where:

cond

is an optional condition code.

datatype

must be one of `S8`, `S16`, `S32`, `S64`, `U8`, `U16`, `U32`, or `U64`.

Qd, Qm, Qn

are the destination vector, the first operand vector, and the second operand vector, for a quadword operation.

Dd, Dm, Dn

are the destination vector, the first operand vector, and the second operand vector, for a doubleword operation.

Operation

`VQSHL` 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 truncating right shift.

The sticky QC flag (FPSCR bit[27]) is set if saturation occurs.

Related references

C1.9 Condition code suffixes on page C1-92

C3.105 VQSHL and VQSHLU (by immediate)

Vector Saturating Shift Left.

Syntax

`VQSHL{U}{cond}.datatype {Qd}, Qm, #imm`

`VQSHL{U}{cond}.datatype {Dd}, Dm, #imm`

where:

U

only permitted if Q is also present. Indicates that the results are unsigned even though the operands are signed.

cond

is an optional condition code.

datatype

must be one of :

S8, S16, S32, S64

for VQSHL or VQSHLU.

U8, U16, U32, U64

for VQSHL only.

Qd, Qm

are the destination and operand vectors, for a quadword operation.

Dd, Dm

are the destination and operand vectors, for a doubleword operation.

imm

is the immediate value specifying the size of the shift, in the range 0 to (size(*datatype*) – 1). The ranges are shown in the following table:

Table C3-11 Available immediate ranges in VQSHL and VQSHLU (by immediate)

datatype	imm range
S8 or U8	0 to 7
S16 or U16	0 to 15
S32 or U32	0 to 31
S64 or U64	0 to 63

Operation

VQSHL and VQSHLU instructions take each element in a vector of integers, left shift them by an immediate value, and place the results in the destination vector.

The sticky QC flag (FPSCR bit[27]) is set if saturation occurs.

Related references

C1.9 Condition code suffixes on page C1-92

C3.106 VQSHRN and VQSHRUN (by immediate)

Vector Saturating Shift Right, Narrow, by immediate value.

Syntax

`VQSHR{U}N{cond}.datatype Dd, Qm, #imm`

where:

`U`

if present, indicates that the results are unsigned, although the operands are signed. Otherwise, the results are the same type as the operands.

cond

is an optional condition code.

datatype

must be one of:

I16, I32, I64

for VQSHRN or VQSHRUN. Only a `#0` immediate is permitted with these datatypes.

S16, S32, S64

for VQSHRN or VQSHRUN.

U16, U32, U64

for VQSHRN only.

Dd, Qm

are the destination vector and the operand vector.

imm

is the immediate value specifying the size of the shift. The ranges are shown in the following table:

Table C3-12 Available immediate ranges in VQSHRN and VQSHRUN (by immediate)

datatype	imm range
S16 or U16	0 to 8
S32 or U32	0 to 16
S64 or U64	0 to 32

Operation

VQSHR{U}N takes each element in a quadword vector of integers, right shifts them by an immediate value, and places the results in a doubleword vector.

The sticky QC flag (FPSCR bit[27]) is set if saturation occurs.

Results are truncated.

Related references

C1.9 Condition code suffixes on page C1-92

C3.107 VQSUB

Vector Saturating Subtract.

Syntax

`VQSUB{cond}.datatype {Qd}, Qn, Qm`

`VQSUB{cond}.datatype {Dd}, Dn, Dm`

where:

cond

is an optional condition code.

datatype

must be one of `S8`, `S16`, `S32`, `S64`, `U8`, `U16`, `U32`, or `U64`.

Qd, Qn, Qm

are the destination vector, the first operand vector, and the second operand vector, for a quadword operation.

Dd, Dn, Dm

are the destination vector, the first operand vector, and the second operand vector, for a doubleword operation.

Operation

`VQSUB` subtracts the elements of one vector from the corresponding elements of another vector, and places the results in the destination vector.

The sticky QC flag (FPSCR bit[27]) is set if saturation occurs.

Related references

C1.9 Condition code suffixes on page C1-92

C3.108 VRADDHN

Vector Rounding Add and Narrow, selecting High half.

Syntax

`VRADDHN{cond}.datatype Dd, Qn, Qm`

where:

cond

is an optional condition code.

datatype

must be one of `I16`, `I32`, or `I64`.

Dd, Qn, Qm

are the destination vector, the first operand vector, and the second operand vector.

Operation

`VRADDHN` adds corresponding elements in two quadword vectors, selects the most significant halves of the results, and places the final results in the destination doubleword vector. Results are rounded.

Related references

C1.9 Condition code suffixes on page C1-92

C3.109 VRECPE

Vector Reciprocal Estimate.

Syntax

`VRECPE{cond}.datatype Qd, Qm`

`VRECPE{cond}.datatype Dd, Dm`

where:

cond

is an optional condition code.

datatype

must be either `U32` or `F32`.

Qd, Qm

are the destination vector and the operand vector, for a quadword operation.

Dd, Dm

are the destination vector and the operand vector, for a doubleword operation.

Operation

`VRECPE` finds an approximate reciprocal of each element in a vector, and places the results in a second vector.

Results for out-of-range inputs

The following table shows the results where input values are out of range:

Table C3-13 Results for out-of-range inputs in VRECPE

	Operand element	Result element
Integer	<= `0x7FFFFFFF`	`0xFFFFFFFF`
Floating-point	NaN	Default NaN
	Negative 0, Negative Denormal	Negative Infinity [ao]
	Positive 0, Positive Denormal	Positive Infinity [ao]
	Positive infinity	Positive 0
	Negative infinity	Negative 0

Related references

C1.9 Condition code suffixes on page C1-92

[ao] The Division by Zero exception bit in the FPSCR (FPSCR[1]) is set

C3.110 VRECPS

Vector Reciprocal Step.

Syntax

`VRECPS{cond}.F32 {Qd}, Qn, Qm`

`VRECPS{cond}.F32 {Dd}, Dn, Dm`

where:

cond

is an optional condition code.

Qd, Qn, Qm

are the destination vector, the first operand vector, and the second operand vector, for a quadword operation.

Dd, Dn, Dm

are the destination vector, the first operand vector, and the second operand vector, for a doubleword operation.

Operation

`VRECPS` multiplies the elements of one vector by the corresponding elements of another vector, subtracts each of the results from 2, and places the final results into the elements of the destination vector.

The Newton-Raphson iteration:

$x_{n+1} = x_n\,(2 - d x_n)$

converges to $(1/d)$ if x_0 is the result of `VRECPE` applied to d.

Results for out-of-range inputs

The following table shows the results where input values are out of range:

Table C3-14 Results for out-of-range inputs in VRECPS

1st operand element	2nd operand element	Result element
NaN	-	Default NaN
-	NaN	Default NaN
± 0.0 or denormal	± infinity	2.0
± infinity	± 0.0 or denormal	2.0

Related references

C1.9 Condition code suffixes on page C1-92

C3.111 VREV16, VREV32, and VREV64

Vector Reverse within halfwords, words, or doublewords.

Syntax

`VREV`*n*`{`*cond*`}.`*size* *Qd*`,` *Qm*

`VREV`*n*`{`*cond*`}.`*size* *Dd*`,` *Dm*

where:

n

must be one of `16`, `32`, or `64`.

cond

is an optional condition code.

size

must be one of `8`, `16`, or `32`, and must be less than *n*.

Qd, *Qm*

specifies the destination vector and the operand vector, for a quadword operation.

Dd, *Dm*

specifies the destination vector and the operand vector, for a doubleword operation.

Operation

`VREV16` reverses the order of 8-bit elements within each halfword of the vector, and places the result in the corresponding destination vector.

`VREV32` reverses the order of 8-bit or 16-bit elements within each word of the vector, and places the result in the corresponding destination vector.

`VREV64` reverses the order of 8-bit, 16-bit, or 32-bit elements within each doubleword of the vector, and places the result in the corresponding destination vector.

Related references

C1.9 Condition code suffixes on page C1-92

C3.112 VRHADD

Vector Rounding Halving Add.

Syntax

`VRHADD{cond}.datatype {Qd}, Qn, Qm`

`VRHADD{cond}.datatype {Dd}, Dn, Dm`

where:

cond

is an optional condition code.

datatype

must be one of `S8`, `S16`, `S32`, `U8`, `U16`, or `U32`.

Qd, Qn, Qm

are the destination vector, the first operand vector, and the second operand vector, for a quadword operation.

Dd, Dn, Dm

are the destination vector, the first operand vector, and the second operand vector, for a doubleword operation.

Operation

`VRHADD` adds corresponding elements in two vectors, shifts each result right one bit, and places the results in the destination vector. Results are rounded.

Related references

C1.9 Condition code suffixes on page C1-92

C3.113 VRSHL (by signed variable)

Vector Rounding Shift Left by signed variable.

Syntax

`VRSHL{cond}.datatype {Qd}, Qm, Qn`

`VRSHL{cond}.datatype {Dd}, Dm, Dn`

where:

cond

is an optional condition code.

datatype

must be one of `S8`, `S16`, `S32`, `S64`, `U8`, `U16`, `U32`, or `U64`.

Qd, Qm, Qn

are the destination vector, the first operand vector, and the second operand vector, for a quadword operation.

Dd, Dm, Dn

are the destination vector, the first operand vector, and the second operand vector, for a doubleword operation.

Operation

`VRSHL` 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 rounding right shift.

Related references

C1.9 Condition code suffixes on page C1-92

C3.114 VRSHR (by immediate)

Vector Rounding Shift Right by immediate value.

Syntax

`VRSHR{cond}.datatype {Qd}, Qm, #imm`

`VRSHR{cond}.datatype {Dd}, Dm, #imm`

where:

cond

is an optional condition code.

datatype

must be one of `S8`, `S16`, `S32`, `S64`, `U8`, `U16`, `U32`, or `U64`.

Qd, Qm

are the destination vector and the operand vector, for a quadword operation.

Dd, Dm

are the destination vector and the operand vector, for a doubleword operation.

imm

is the immediate value specifying the size of the shift, in the range 0 to (size(*datatype*)). The ranges are shown in the following table:

Table C3-15 Available immediate ranges in VRSHR (by immediate)

datatype	imm range
`S8` or `U8`	0 to 8
`S16` or `U16`	0 to 16
`S32` or `U32`	0 to 32
`S64` or `U64`	0 to 64

`VRSHR` with an immediate value of zero is a pseudo-instruction for `VORR`.

Operation

`VRSHR` takes each element in a vector, right shifts them by an immediate value, and places the results in the destination vector. The results are rounded.

Related references

C3.86 VORR (register) on page C3-481

C1.9 Condition code suffixes on page C1-92

C3.115 VRSHRN (by immediate)

Vector Rounding Shift Right, Narrow, by immediate value.

Syntax

`VRSHRN{cond}.datatype Dd, Qm, #imm`

where:

cond

is an optional condition code.

datatype

must be one of `I16`, `I32`, or `I64`.

Dd, Qm

are the destination vector and the operand vector.

imm

is the immediate value specifying the size of the shift, in the range 0 to (size(*datatype*)/2). The ranges are shown in the following table:

Table C3-16 Available immediate ranges in VRSHRN (by immediate)

datatype	imm range
I16	0 to 8
I32	0 to 16
I64	0 to 32

`VRSHRN` with an immediate value of zero is a pseudo-instruction for `VMOVN`.

Operation

`VRSHRN` takes each element in a quadword vector, right shifts them by an immediate value, and places the results in a doubleword vector. The results are rounded.

Related references

C3.73 VMOVN on page C3-468
C1.9 Condition code suffixes on page C1-92

C3.116 VRINT

VRINT (Vector Round to Integer) rounds each floating-point element in a vector to integer, and places the results in the destination vector.

The resulting integers are represented in floating-point format.

Note

This instruction is supported only in Armv8.

Syntax

VRINT*mode*.F32.F32 *Qd, Qm*

VRINT*mode*.F32.F32 *Dd, Dm*

where:

mode

must be one of:

A

meaning round to nearest, ties away from zero. This cannot generate an Inexact exception, even if the result is not exact.

N

meaning round to nearest, ties to even. This cannot generate an Inexact exception, even if the result is not exact.

X

meaning round to nearest, ties to even, generating an Inexact exception if the result is not exact.

P

meaning round towards plus infinity. This cannot generate an Inexact exception, even if the result is not exact.

M

meaning round towards minus infinity. This cannot generate an Inexact exception, even if the result is not exact.

Z

meaning round towards zero. This cannot generate an Inexact exception, even if the result is not exact.

Qd, Qm

specifies the destination vector and the operand vector, for a quadword operation.

Dd, Dm

specifies the destination and operand vectors, for a doubleword operation.

Notes

You cannot use VRINT inside an IT block.

C3.117 VRSQRTE

Vector Reciprocal Square Root Estimate.

Syntax

`VRSQRTE{cond}.datatype Qd, Qm`

`VRSQRTE{cond}.datatype Dd, Dm`

where:

cond

is an optional condition code.

datatype

must be either `U32` or `F32`.

Qd, Qm

are the destination vector and the operand vector, for a quadword operation.

Dd, Dm

are the destination vector and the operand vector, for a doubleword operation.

Operation

`VRSQRTE` finds an approximate reciprocal square root of each element in a vector, and places the results in a second vector.

Results for out-of-range inputs

The following table shows the results where input values are out of range:

Table C3-17 Results for out-of-range inputs in VRSQRTE

	Operand element	Result element
Integer	<= `0x3FFFFFFF`	`0xFFFFFFFF`
Floating-point	NaN, Negative Normal, Negative Infinity	Default NaN
	Negative 0, Negative Denormal	Negative Infinity [ap]
	Positive 0, Positive Denormal	Positive Infinity [ap]
	Positive infinity	Positive 0
		Negative 0

Related references

C1.9 Condition code suffixes on page C1-92

[ap] The Division by Zero exception bit in the FPSCR (FPSCR[1]) is set

C3.118 VRSQRTS

Vector Reciprocal Square Root Step.

Syntax

`VRSQRTS{cond}.F32 {Qd}, Qn, Qm`

`VRSQRTS{cond}.F32 {Dd}, Dn, Dm`

where:

cond

is an optional condition code.

Qd, Qn, Qm

are the destination vector, the first operand vector, and the second operand vector, for a quadword operation.

Dd, Dn, Dm

are the destination vector, the first operand vector, and the second operand vector, for a doubleword operation.

Operation

`VRSQRTS` multiplies the elements of one vector by the corresponding elements of another vector, subtracts each of the results from three, divides these results by two, and places the final results into the elements of the destination vector.

The Newton-Raphson iteration:

$x_{n+1} = x_n\,(3-dx_n^2)/2$

converges to $(1/\sqrt{d})$ if x_0 is the result of `VRSQRTE` applied to d.

Results for out-of-range inputs

The following table shows the results where input values are out of range:

Table C3-18 Results for out-of-range inputs in VRSQRTS

1st operand element	2nd operand element	Result element
NaN	-	Default NaN
-	NaN	Default NaN
± 0.0 or denormal	± infinity	1.5
± infinity	± 0.0 or denormal	1.5

Related references

C1.9 Condition code suffixes on page C1-92

C3.119 VRSRA (by immediate)

Vector Rounding Shift Right by immediate value and Accumulate.

Syntax

`VRSRA{cond}.datatype {Qd}, Qm, #imm`

`VRSRA{cond}.datatype {Dd}, Dm, #imm`

where:

cond

is an optional condition code.

datatype

must be one of `S8`, `S16`, `S32`, `S64`, `U8`, `U16`, `U32`, or `U64`.

Qd, Qm

are the destination vector and the operand vector, for a quadword operation.

Dd, Dm

are the destination vector and the operand vector, for a doubleword operation.

imm

is the immediate value specifying the size of the shift, in the range 1 to (size(*datatype*)). The ranges are shown in the following table:

Table C3-19 Available immediate ranges in VRSRA (by immediate)

datatype	imm range
`S8` or `U8`	1 to 8
`S16` or `U16`	1 to 16
`S32` or `U32`	1 to 32
`S64` or `U64`	1 to 64

Operation

`VRSRA` takes each element in a vector, right shifts them by an immediate value, and accumulates the results into the destination vector. The results are rounded.

Related references

C1.9 Condition code suffixes on page C1-92

C3.120 VRSUBHN

Vector Rounding Subtract and Narrow, selecting High half.

Syntax

`VRSUBHN{cond}.datatype Dd, Qn, Qm`

where:

cond

is an optional condition code.

datatype

must be one of `I16`, `I32`, or `I64`.

Dd, Qn, Qm

are the destination vector, the first operand vector, and the second operand vector.

Operation

`VRSUBHN` subtracts the elements of one quadword vector from the corresponding elements of another quadword vector, selects the most significant halves of the results, and places the final results in the destination doubleword vector. Results are rounded.

Related references

C1.9 Condition code suffixes on page C1-92

C3.121 VSDOT (vector)

Dot Product vector form with signed integers.

Syntax

```
VSDOT{q}.S8 Dd, Dn, Dm ; 64-bit SIMD vector
VSDOT{q}.S8 Qd, Qn, Qm ; A1 128-bit SIMD vector FP/SIMD registers (A32)
```

Where:

q
Is an optional instruction width specifier. See *C2.2 Instruction width specifiers* on page C2-111.

Dd
Is the 64-bit name of the SIMD and FP destination register.

Dn
Is the 64-bit name of the first SIMD and FP source register.

Dm
Is the 64-bit name of the second SIMD and FP source register.

Qd
Is the 128-bit name of the SIMD and FP destination register.

Qn
Is the 128-bit name of the first SIMD and FP source register.

Qm
Is the 128-bit name of the second SIMD and FP source register.

Architectures supported

Supported in Armv8.2 and later.

For Armv8.2 and Armv8.3, this is an OPTIONAL instruction.

Usage

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.

Note

ID_ISAR6.DP indicates whether this instruction is supported in the T32 and A32 instruction sets.

Related references

C3.1 Summary of Advanced SIMD instructions on page C3-391

C3.122 VSDOT (by element)

Dot Product index form with signed integers.

Syntax

```
VSDOT{q}.S8 Dd, Dn, Dm[index] ; 64-bit SIMD vector
VSDOT{q}.S8 Qd, Qn, Dm[index] ; A1 128-bit SIMD vector FP/SIMD registers (A32)
```

Where:

q
Is an optional instruction width specifier. See *C2.2 Instruction width specifiers* on page C2-111.

Dd
Is the 64-bit name of the SIMD and FP destination register.

Dn
Is the 64-bit name of the first SIMD and FP source register.

Dm
Is the 64-bit name of the second SIMD and FP source register.

index
Is the element index in the range 0 to 1.

Qd
Is the 128-bit name of the SIMD and FP destination register.

Qn
Is the 128-bit name of the first SIMD and FP source register.

Architectures supported

Supported in Armv8.2 and later.

For Armv8.2 and Armv8.3, this is an OPTIONAL instruction.

Usage

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.

Note

ID_ISAR6.DP indicates whether this instruction is supported in the T32 and A32 instruction sets.

Related references

C3.1 Summary of Advanced SIMD instructions on page C3-391

C3.123 VSHL (by immediate)

Vector Shift Left by immediate.

Syntax

`VSHL{cond}.datatype {Qd}, Qm, #imm`

`VSHL{cond}.datatype {Dd}, Dm, #imm`

where:

cond

is an optional condition code.

datatype

must be one of `I8`, `I16`, `I32`, or `I64`.

Qd, Qm

are the destination and operand vectors, for a quadword operation.

Dd, Dm

are the destination and operand vectors, for a doubleword operation.

imm

is the immediate value specifying the size of the shift. The ranges are shown in the following table:

Table C3-20 Available immediate ranges in VSHL (by immediate)

datatype	imm range
I8	0 to 7
I16	0 to 15
I32	0 to 31
I64	0 to 63

Operation

`VSHL` 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 following figure shows the operation of `VSHL` with two elements and a shift value of one. The least significant bit in each element in the destination vector is set to zero.

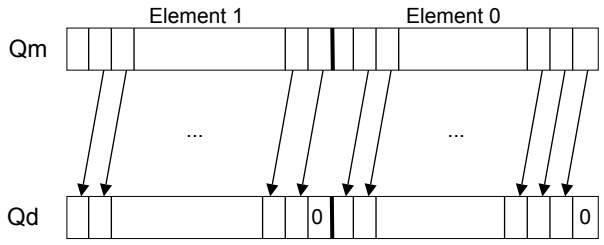


Figure C3-6 Operation of quadword VSHL.I64 Qd, Qm, #1

Related references

C1.9 Condition code suffixes on page C1-92

C3.124 VSHL (by signed variable)

Vector Shift Left by signed variable.

Syntax

`VSHL{cond}.datatype {Qd}, Qm, Qn`

`VSHL{cond}.datatype {Dd}, Dm, Dn`

where:

cond

is an optional condition code.

datatype

must be one of `S8`, `S16`, `S32`, `S64`, `U8`, `U16`, `U32`, or `U64`.

Qd, Qm, Qn

are the destination vector, the first operand vector, and the second operand vector, for a quadword operation.

Dd, Dm, Dn

are the destination vector, the first operand vector, and the second operand vector, for a doubleword operation.

Operation

`VSHL` takes each element in a vector, shifts them by the 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 truncating right shift.

Related references

C1.9 Condition code suffixes on page C1-92

C3.125 VSHLL (by immediate)

Vector Shift Left Long.

Syntax

`VSHLL{cond}.datatype Qd, Dm, #imm`

where:

cond

is an optional condition code.

datatype

must be one of `S8`, `S16`, `S32`, `U8`, `U16`, or `U32`.

Qd, Dm

are the destination and operand vectors, for a long operation.

imm

is the immediate value specifying the size of the shift. The ranges are shown in the following table:

Table C3-21 Available immediate ranges in VSHLL (by immediate)

datatype	imm range
`S8` or `U8`	1 to 8
`S16` or `U16`	1 to 16
`S32` or `U32`	1 to 32

0 is permitted, but the resulting code disassembles to `VMOVL`.

Operation

`VSHLL` takes each element in a vector of integers, left shifts them by an immediate value, and places the results in the destination vector. Values are sign or zero extended.

Related references

C1.9 Condition code suffixes on page C1-92

C3.126 VSHR (by immediate)

Vector Shift Right by immediate value.

Syntax

`VSHR{cond}.datatype {Qd}, Qm, #imm`

`VSHR{cond}.datatype {Dd}, Dm, #imm`

where:

cond

is an optional condition code.

datatype

must be one of `S8`, `S16`, `S32`, `S64`, `U8`, `U16`, `U32`, or `U64`.

Qd, Qm

are the destination vector and the operand vector, for a quadword operation.

Dd, Dm

are the destination vector and the operand vector, for a doubleword operation.

imm

is the immediate value specifying the size of the shift. The ranges are shown in the following table:

Table C3-22 Available immediate ranges in VSHR (by immediate)

datatype	imm range
`S8` or `U8`	0 to 8
`S16` or `U16`	0 to 16
`S32` or `U32`	0 to 32
`S64` or `U64`	0 to 64

`VSHR` with an immediate value of zero is a pseudo-instruction for `VORR`.

Operation

`VSHR` takes each element in a vector, right shifts them by an immediate value, and places the results in the destination vector. The results are truncated.

Related references

C3.86 VORR (register) on page C3-481

C1.9 Condition code suffixes on page C1-92

C3.127 VSHRN (by immediate)

Vector Shift Right, Narrow, by immediate value.

Syntax

`VSHRN{cond}.datatype Dd, Qm, #imm`

where:

cond

is an optional condition code.

datatype

must be one of `I16`, `I32`, or `I64`.

Dd, Qm

are the destination vector and the operand vector.

imm

is the immediate value specifying the size of the shift. The ranges are shown in the following table:

Table C3-23 Available immediate ranges in VSHRN (by immediate)

datatype	imm range
I16	0 to 8
I32	0 to 16
I64	0 to 32

`VSHRN` with an immediate value of zero is a pseudo-instruction for `VMOVN`.

Operation

`VSHRN` takes each element in a quadword vector, right shifts them by an immediate value, and places the results in a doubleword vector. The results are truncated.

Related references

C3.73 VMOVN on page C3-468

C1.9 Condition code suffixes on page C1-92

C3.128 VSLI

Vector Shift Left and Insert.

Syntax

`VSLI{cond}.size {Qd}, Qm, #imm`

`VSLI{cond}.size {Dd}, Dm, #imm`

where:

cond

is an optional condition code.

size

must be one of `8`, `16`, `32`, or `64`.

Qd, Qm

are the destination vector and the operand vector, for a quadword operation.

Dd, Dm

are the destination vector and the operand vector, for a doubleword operation.

imm

is the immediate value specifying the size of the shift, in the range 0 to (*size* – 1).

Operation

`VSLI` takes each element in a 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 following figure shows the operation of `VSLI` with two elements and a shift value of one. The least significant bit in each element in the destination vector is unchanged.

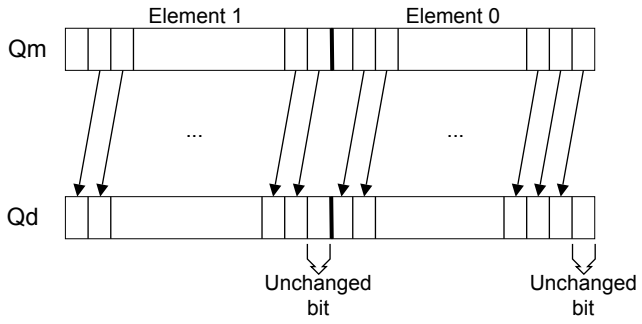


Figure C3-7 Operation of quadword VSLI.64 Qd, Qm, #1

Related references

C1.9 Condition code suffixes on page C1-92

C3.129 VSRA (by immediate)

Vector Shift Right by immediate value and Accumulate.

Syntax

`VSRA{cond}.datatype {Qd}, Qm, #imm`

`VSRA{cond}.datatype {Dd}, Dm, #imm`

where:

cond

is an optional condition code.

datatype

must be one of `S8`, `S16`, `S32`, `S64`, `U8`, `U16`, `U32`, or `U64`.

Qd, Qm

are the destination vector and the operand vector, for a quadword operation.

Dd, Dm

are the destination vector and the operand vector, for a doubleword operation.

imm

is the immediate value specifying the size of the shift. The ranges are shown in the following table:

Table C3-24 Available immediate ranges in VSRA (by immediate)

datatype	imm range
`S8` or `U8`	1 to 8
`S16` or `U16`	1 to 16
`S32` or `U32`	1 to 32
`S64` or `U64`	1 to 64

Operation

`VSRA` takes each element in a vector, right shifts them by an immediate value, and accumulates the results into the destination vector. The results are truncated.

Related references

C1.9 Condition code suffixes on page C1-92

C3.130 VSRI

Vector Shift Right and Insert.

Syntax

`VSRI{cond}.size {Qd}, Qm, #imm`

`VSRI{cond}.size {Dd}, Dm, #imm`

where:

cond

is an optional condition code.

size

must be one of `8`, `16`, `32`, or `64`.

Qd, Qm

are the destination vector and the operand vector, for a quadword operation.

Dd, Dm

are the destination vector and the operand vector, for a doubleword operation.

imm

is the immediate value specifying the size of the shift, in the range 1 to *size*.

Operation

`VSRI` takes each element in a 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 following figure shows the operation of `VSRI` with a single element and a shift value of two. The two most significant bits in the destination vector are unchanged.

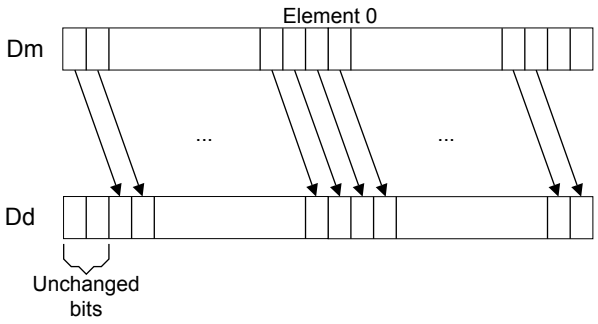


Figure C3-8 Operation of doubleword VSRI.64 Dd, Dm, #2

Related references

C1.9 Condition code suffixes on page C1-92

C3.131 VSTM

Extension register store multiple.

Syntax

`VSTM`*`mode`*`{`*`cond`*`} `*`Rn`*`{!}, `*`Registers`*

where:

`mode`

must be one of:

`IA`

meaning Increment address After each transfer. `IA` is the default, and can be omitted.

`DB`

meaning Decrement address Before each transfer.

`EA`

meaning Empty Ascending stack operation. This is the same as `IA` for stores.

`FD`

meaning Full Descending stack operation. This is the same as `DB` for stores.

`cond`

is an optional condition code.

`Rn`

is the general-purpose register holding the base address for the transfer.

`!`

is optional. `!` specifies that the updated base address must be written back to *`Rn`*. If `!` is not specified, *`mode`* must be `IA`.

`Registers`

is a list of consecutive extension registers enclosed in braces, `{` and `}`. The list can be comma-separated, or in range format. There must be at least one register in the list.

You can specify `D` or `Q` registers, but they must not be mixed. The number of registers must not exceed 16 `D` registers, or 8 `Q` registers. If `Q` registers are specified, on disassembly they are shown as `D` registers.

Note

`VPUSH` *`Registers`* is equivalent to `VSTMDB sp!,` *`Registers`*.

You can use either form of this instruction. They both disassemble to `VPUSH`.

Related references

C1.9 Condition code suffixes on page C1-92
C4.38 VSTM (floating-point) on page C4-585

C3.132 VSTn (multiple n-element structures)

Vector Store multiple *n*-element structures.

Syntax

`VST`*n*`{`*cond*`}.`*datatype list*`, [`*Rn*`{@`*align*`}]{!}`

`VST`*n*`{`*cond*`}.`*datatype list*`, [`*Rn*`{@`*align*`}], `*Rm*

where:

n

must be one of 1, 2, 3, or 4.

cond

is an optional condition code.

datatype

see the following table for options.

list

is the list of Advanced SIMD registers enclosed in braces, `{` and `}`. See the following table for options.

Rn

is the general-purpose register containing the base address. *Rn* cannot be PC.

align

specifies an optional alignment. See the following table for options.

!

if ! is present, *Rn* is updated to (*Rn* + the number of bytes transferred by the instruction). The update occurs after all the stores have taken place.

Rm

is a general-purpose register containing an offset from the base address. If *Rm* is present, the instruction updates *Rn* to (*Rn* + *Rm*) after using the address to access memory. *Rm* cannot be SP or PC.

Operation

`VST`*n* stores multiple *n*-element structures to memory from one or more Advanced SIMD registers, with interleaving (unless *n* == 1). Every element of each register is stored.

Table C3-25 Permitted combinations of parameters for VSTn (multiple n-element structures)

n	*datatype*	list [aq]	*align* [ar]	*alignment*
1	8, 16, 32, or 64	{Dd}	@64	8-byte
		{Dd, D(d+1)}	@64 or @128	8-byte or 16-byte
		{Dd, D(d+1), D(d+2)}	@64	8-byte
		{Dd, D(d+1), D(d+2), D(d+3)}	@64, @128, or @256	8-byte, 16-byte, or 32-byte

[aq] Every register in the list must be in the range D0-D31.
[ar] *align* can be omitted. In this case, standard alignment rules apply.

Table C3-25 Permitted combinations of parameters for VSTn (multiple n-element structures) (continued)

n	*datatype*	list [aq]	*align* [ar]	*alignment*
2	8, 16, or 32	{Dd, D(d+1)}	@64, @128	8-byte or 16-byte
		{Dd, D(d+2)}	@64, @128	8-byte or 16-byte
		{Dd, D(d+1), D(d+2), D(d+3)}	@64, @128, or @256	8-byte, 16-byte, or 32-byte
3	8, 16, or 32	{Dd, D(d+1), D(d+2)}	@64	8-byte
		{Dd, D(d+2), D(d+4)}	@64	8-byte
4	8, 16, or 32	{Dd, D(d+1), D(d+2), D(d+3)}	@64, @128, or @256	8-byte, 16-byte, or 32-byte
		{Dd, D(d+2), D(d+4), D(d+6)}	@64, @128, or @256	8-byte, 16-byte, or 32-byte

Related concepts

C3.3 Interleaving provided by load and store element and structure instructions on page C3-395

C3.4 Alignment restrictions in load and store element and structure instructions on page C3-396

Related references

C1.9 Condition code suffixes on page C1-92

C3.133 VSTn (single n-element structure to one lane)

Vector Store single *n*-element structure to one lane.

Syntax

`VSTn{cond}.datatype list, [Rn{@align}]{!}`

`VSTn{cond}.datatype list, [Rn{@align}], Rm`

where:

n

must be one of 1, 2, 3, or 4.

cond

is an optional condition code.

datatype

see the following table.

list

is the list of Advanced SIMD registers enclosed in braces, `{` and `}`. See the following table for options.

Rn

is the general-purpose register containing the base address. *Rn* cannot be PC.

align

specifies an optional alignment. See the following table for options.

!

if ! is present, *Rn* is updated to (*Rn* + the number of bytes transferred by the instruction). The update occurs after all the stores have taken place.

Rm

is a general-purpose register containing an offset from the base address. If *Rm* is present, the instruction updates *Rn* to (*Rn* + *Rm*) after using the address to access memory. *Rm* cannot be SP or PC.

Operation

VST*n* stores one *n*-element structure into memory from one or more Advanced SIMD registers.

Table C3-26 Permitted combinations of parameters for VSTn (single n-element structure to one lane)

n	*datatype*	list [as]	*align* [at]	alignment
1	8	{Dd[x]}	-	Standard only
	16	{Dd[x]}	@16	2-byte
	32	{Dd[x]}	@32	4-byte
2	8	{Dd[x], D(d+1)[x]}	@16	2-byte
	16	{Dd[x], D(d+1)[x]}	@32	4-byte

[as] Every register in the list must be in the range D0-D31.
[at] *align* can be omitted. In this case, standard alignment rules apply.

Table C3-26 Permitted combinations of parameters for VSTn (single n-element structure to one lane) (continued)

n	*datatype*	list [as]	*align* [at]	alignment
		{Dd[x], D(d+2)[x]}	@32	4-byte
	32	{Dd[x], D(d+1)[x]}	@64	8-byte
		{Dd[x], D(d+2)[x]}	@64	8-byte
3	8	{Dd[x], D(d+1)[x], D(d+2)[x]}	-	Standard only
	16 or 32	{Dd[x], D(d+1)[x], D(d+2)[x]}	-	Standard only
		{Dd[x], D(d+2)[x], D(d+4)[x]}	-	Standard only
4	8	{Dd[x], D(d+1)[x], D(d+2)[x], D(d+3)[x]}	@32	4-byte
	16	{Dd[x], D(d+1)[x], D(d+2)[x], D(d+3)[x]}	@64	8-byte
		{Dd[x], D(d+2)[x], D(d+4)[x], D(d+6)[x]}	@64	8-byte
	32	{Dd[x], D(d+1)[x], D(d+2)[x], D(d+3)[x]}	@64 or @128	8-byte or 16-byte
		{Dd[x], D(d+2)[x], D(d+4)[x], D(d+6)[x]}	@64 or @128	8-byte or 16-byte

Related concepts

C3.3 Interleaving provided by load and store element and structure instructions on page C3-395

C3.4 Alignment restrictions in load and store element and structure instructions on page C3-396

Related references

C1.9 Condition code suffixes on page C1-92

C3.134 VSTR

Extension register store.

Syntax

`VSTR{cond}{.64} Dd, [Rn{, #offset}]`

where:

cond

is an optional condition code.

Dd

is the extension register to be saved.

Rn

is the general-purpose register holding the base address for the transfer.

offset

is an optional numeric expression. It must evaluate to a numeric value at assembly time. The value must be a multiple of 4, and lie in the range -1020 to +1020. The value is added to the base address to form the address used for the transfer.

Operation

The `VSTR` instruction saves the contents of an extension register to memory.

Two words are transferred.

Related references

C1.9 Condition code suffixes on page C1-92
C4.39 VSTR (floating-point) on page C4-586

C3.135 VSTR (post-increment and pre-decrement)

Pseudo-instruction that stores extension registers with post-increment and pre-decrement forms.

Note

There are also `VLDR` and `VSTR` instructions without post-increment and pre-decrement.

Syntax

`VSTR{`*cond*`}{.64}` *Dd*`, [`*Rn*`], #`*offset* `; post-increment`

`VSTR{`*cond*`}{.64}` *Dd*`, [`*Rn*`, #-`*offset*`]! ; pre-decrement`

where:

cond

is an optional condition code.

Dd

is the extension register to be saved.

Rn

is the general-purpose register holding the base address for the transfer.

offset

is a numeric expression that must evaluate to 8 at assembly time.

Operation

The post-increment instruction increments the base address in the register by the offset value, after the transfer. The pre-decrement instruction decrements the base address in the register by the offset value, and then performs the transfer using the new address in the register. This pseudo-instruction assembles to a `VSTM` instruction.

Related references

C3.134 VSTR on page C3-531

C3.131 VSTM on page C3-526

C1.9 Condition code suffixes on page C1-92

C4.40 VSTR (post-increment and pre-decrement, floating-point) on page C4-587

C3.136 VSUB

Vector Subtract.

Syntax

`VSUB{cond}.datatype {Qd}, Qn, Qm`

`VSUB{cond}.datatype {Dd}, Dn, Dm`

where:

cond

is an optional condition code.

datatype

must be one of `I8`, `I16`, `I32`, `I64`, or `F32`.

Qd, Qn, Qm

are the destination vector, the first operand vector, and the second operand vector, for a quadword operation.

Operation

`VSUB` subtracts the elements of one vector from the corresponding elements of another vector, and places the results in the destination vector.

Related references

C1.9 Condition code suffixes on page C1-92

C3.137 VSUBHN

Vector Subtract and Narrow, selecting High half.

Syntax

`VSUBHN{cond}.datatype Dd, Qn, Qm`

where:

cond

is an optional condition code.

datatype

must be one of `I16`, `I32`, or `I64`.

Dd, Qn, Qm

are the destination vector, the first operand vector, and the second operand vector.

Operation

`VSUBHN` subtracts the elements of one quadword vector from the corresponding elements of another quadword vector, selects the most significant halves of the results, and places the final results in the destination doubleword vector. Results are truncated.

Related references

C1.9 Condition code suffixes on page C1-92

C3.138 VSUBL and VSUBW

Vector Subtract Long, Vector Subtract Wide.

Syntax

```
VSUBL{cond}.datatype Qd, Dn, Dm ; Long operation
VSUBW{cond}.datatype {Qd}, Qn, Dm ; Wide operation
```

where:

cond

is an optional condition code.

datatype

must be one of `S8`, `S16`, `S32`, `U8`, `U16`, or `U32`.

Qd, Dn, Dm

are the destination vector, the first operand vector, and the second operand vector, for a long operation.

Qd, Qn, Dm

are the destination vector, the first operand vector, and the second operand vector, for a wide operation.

Operation

`VSUBL` subtracts the elements of one doubleword vector from the corresponding elements of another doubleword vector, and places the results in the destination quadword vector.

`VSUBW` subtracts the elements of a doubleword vector from the corresponding elements of a quadword vector, and places the results in the destination quadword vector.

Related references

C1.9 Condition code suffixes on page C1-92

C3.139 VSWP

Vector Swap.

Syntax

`VSWP{cond}{.datatype} Qd, Qm`

`VSWP{cond}{.datatype} Dd, Dm`

where:

cond

is an optional condition code.

datatype

is an optional datatype. The assembler ignores *datatype*.

Qd, Qm

specifies the vectors for a quadword operation.

Dd, Dm

specifies the vectors for a doubleword operation.

Operation

`VSWP` exchanges the contents of two vectors. The vectors can be either doubleword or quadword. There is no distinction between data types.

Related references

C1.9 Condition code suffixes on page C1-92

C3.140 VTBL and VTBX

Vector Table Lookup, Vector Table Extension.

Syntax

`V`*op*{*cond*}`.8` *Dd, list, Dm*

where:

op

must be either `TBL` or `TBX`.

cond

is an optional condition code.

Dd

specifies the destination vector.

list

Specifies the vectors containing the table. It must be one of:

- {*Dn*}.
- {*Dn,D(n+1)*}.
- {*Dn,D(n+1),D(n+2)*}.
- {*Dn,D(n+1),D(n+2),D(n+3)*}.
- {*Qn,Q(n+1)*}.

All the registers in *list* must be in the range `D0-D31` or `Q0-Q15` and must not wrap around the end of the register bank. For example {`D31`,`D0`,`D1`} is not permitted. If *list* contains `Q` registers, they disassemble to the equivalent `D` registers.

Dm

specifies the index vector.

Operation

`VTBL` 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 zero.

`VTBX` works in the same way, except that indexes out of range leave the destination element unchanged.

Related references

C1.9 Condition code suffixes on page C1-92

C3.141 VTRN

Vector Transpose.

Syntax

`VTRN{cond}.size Qd, Qm`

`VTRN{cond}.size Dd, Dm`

where:

cond

is an optional condition code.

size

must be one of `8`, `16`, or `32`.

Qd, Qm

specifies the vectors, for a quadword operation.

Dd, Dm

specifies the vectors, for a doubleword operation.

Operation

`VTRN` treats the elements of its operand vectors as elements of 2 x 2 matrices, and transposes the matrices. The following figures show examples of the operation of `VTRN`:

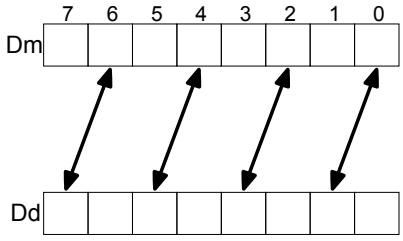


Figure C3-9 Operation of doubleword VTRN.8

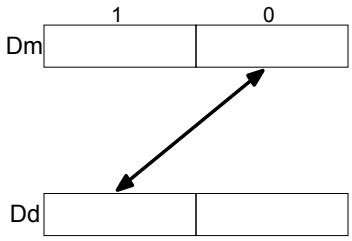


Figure C3-10 Operation of doubleword VTRN.32

Related references

C1.9 Condition code suffixes on page C1-92

C3.142 VTST

Vector Test bits.

Syntax

`VTST{cond}.size {Qd}, Qn, Qm`

`VTST{cond}.size {Dd}, Dn, Dm`

where:

cond

is an optional condition code.

size

must be one of `8`, `16`, or `32`.

Qd, Qn, Qm

specifies the destination register, the first operand register, and the second operand register, for a quadword operation.

Dd, Dn, Dm

specifies the destination register, the first operand register, and the second operand register, for a doubleword operation.

Operation

`VTST` takes each element in a vector, and bitwise logical ANDs them 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.

Related references

C1.9 Condition code suffixes on page C1-92

C3.143 VUDOT (vector)

Dot Product vector form with unsigned integers.

Syntax

```
VUDOT{q}.U8 Dd, Dn, Dm ; 64-bit SIMD vector
VUDOT{q}.U8 Qd, Qn, Qm ; A1 128-bit SIMD vector FP/SIMD registers (A32)
```

Where:

q
Is an optional instruction width specifier. See *C2.2 Instruction width specifiers* on page C2-111.

Dd
Is the 64-bit name of the SIMD and FP destination register.

Dn
Is the 64-bit name of the first SIMD and FP source register.

Dm
Is the 64-bit name of the second SIMD and FP source register.

Qd
Is the 128-bit name of the SIMD and FP destination register.

Qn
Is the 128-bit name of the first SIMD and FP source register.

Qm
Is the 128-bit name of the second SIMD and FP source register.

Architectures supported

Supported in Armv8.2 and later.

For Armv8.2 and Armv8.3, this is an OPTIONAL instruction.

Usage

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.

Note

ID_ISAR6.DP indicates whether this instruction is supported in the T32 and A32 instruction sets.

Related references

C3.1 Summary of Advanced SIMD instructions on page C3-391

C3.144 VUDOT (by element)

Dot Product index form with unsigned integers.

Syntax

```
VUDOT{q}.U8 Dd, Dn, Dm[index] ; 64-bit SIMD vector
VUDOT{q}.U8 Qd, Qn, Dm[index] ; A1 128-bit SIMD vector FP/SIMD registers (A32)
```

Where:

q
Is an optional instruction width specifier. See *C2.2 Instruction width specifiers* on page C2-111.

Dd
Is the 64-bit name of the SIMD and FP destination register.

Dn
Is the 64-bit name of the first SIMD and FP source register.

Dm
Is the 64-bit name of the second SIMD and FP source register.

index
Is the element index in the range 0 to 1.

Qd
Is the 128-bit name of the SIMD and FP destination register.

Qn
Is the 128-bit name of the first SIMD and FP source register.

Architectures supported

Supported in Armv8.2 and later.

For Armv8.2 and Armv8.3, this is an OPTIONAL instruction.

Usage

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.

Note

ID_ISAR6.DP indicates whether this instruction is supported in the T32 and A32 instruction sets.

Related references

C3.1 Summary of Advanced SIMD instructions on page C3-391

C3.145 VUZP

Vector Unzip.

Syntax

`VUZP{`*cond*`}.`*size* *Qd*`,` *Qm*

`VUZP{`*cond*`}.`*size* *Dd*`,` *Dm*

where:

cond
: is an optional condition code.

size
: must be one of `8`, `16`, or `32`.

Qd, *Qm*
: specifies the vectors, for a quadword operation.

Dd, *Dm*
: specifies the vectors, for a doubleword operation.

Note

The following are all the same instruction:

- `VZIP.32` *Dd*`,` *Dm*.
- `VUZP.32` *Dd*`,` *Dm*.
- `VTRN.32` *Dd*`,` *Dm*.

The instruction is disassembled as `VTRN.32` *Dd*`,` *Dm*.

Operation

`VUZP` de-interleaves the elements of two vectors.

De-interleaving is the inverse process of interleaving.

Table C3-27 Operation of doubleword VUZP.8

	Register state before operation								Register state after operation							
Dd	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	B_6	B_4	B_2	B_0	A_6	A_4	A_2	A_0
Dm	B_7	B_6	B_5	B_4	B_3	B_2	B_1	B_0	B_7	B_5	B_3	B_1	A_7	A_5	A_3	A_1

Table C3-28 Operation of quadword VUZP.32

	Register state before operation				Register state after operation			
Qd	A_3	A_2	A_1	A_0	B_2	B_0	A_2	A_0
Qm	B_3	B_2	B_1	B_0	B_3	B_1	A_3	A_1

Related concepts

C3.3 Interleaving provided by load and store element and structure instructions on page C3-395

Related references

C3.141 VTRN on page C3-538
C1.9 Condition code suffixes on page C1-92

C3.146 VZIP

Vector Zip.

Syntax

`VZIP{`*cond*`}.`*size* *Qd*`,` *Qm*

`VZIP{`*cond*`}.`*size* *Dd*`,` *Dm*

where:

cond

is an optional condition code.

size

must be one of `8`, `16`, or `32`.

Qd`,` *Qm*

specifies the vectors, for a quadword operation.

Dd`,` *Dm*

specifies the vectors, for a doubleword operation.

Note

The following are all the same instruction:

- `VZIP.32` *Dd*`,` *Dm*.
- `VUZP.32` *Dd*`,` *Dm*.
- `VTRN.32` *Dd*`,` *Dm*.

The instruction is disassembled as `VTRN.32` *Dd*`,` *Dm*.

Operation

`VZIP` interleaves the elements of two vectors.

Table C3-29 Operation of doubleword VZIP.8

	Register state before operation								Register state after operation							
Dd	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	B_3	A_3	B_2	A_2	B_1	A_1	B_0	A_0
Dm	B_7	B_6	B_5	B_4	B_3	B_2	B_1	B_0	B_7	A_7	B_6	A_6	B_5	A_5	B_4	A_4

Table C3-30 Operation of quadword VZIP.32

	Register state before operation				Register state after operation			
Qd	A_3	A_2	A_1	A_0	B_1	A_1	B_0	A_0
Qm	B_3	B_2	B_1	B_0	B_3	A_3	B_2	A_2

Related concepts

C3.3 Interleaving provided by load and store element and structure instructions on page C3-395

Related references

C3.141 VTRN on page C3-538
C1.9 Condition code suffixes on page C1-92

Chapter C4
Floating-point Instructions (32-bit)

Describes floating-point assembly language instructions.

It contains the following sections:

C4.1 Summary of floating-point instructions

A summary of the floating-point instructions. Not all of these instructions are available in all floating-point versions.

The following table shows a summary of floating-point instructions that are not available in Advanced SIMD.

Note

Floating-point vector mode is not supported in Armv8. Use Advanced SIMD instructions for vector floating-point.

Table C4-1 Summary of floating-point instructions

Mnemonic	Brief description
VABS	Absolute value
VADD	Add
VCMP, VCMPE	Compare
VCVT	Convert between single-precision and double-precision
	Convert between floating-point and integer
	Convert between floating-point and fixed-point
	Convert floating-point to integer with directed rounding modes
VCVTB, VCVTT	Convert between half-precision and single-precision floating-point
	Convert between half-precision and double-precision
VDIV	Divide
VFMA, VFMS	Fused multiply accumulate, Fused multiply subtract
VFNMA, VFNMS	Fused multiply accumulate with negation, Fused multiply subtract with negation
VJCVT	Javascript Convert to signed fixed-point, rounding toward Zero
VLDM	Extension register load multiple
VLDR	Extension register load
VLLDM	Floating-point Lazy Load Multiple
VLSTM	Floating-point Lazy Store Multiple
VMAXNM, VMINNM	Maximum, Minimum, consistent with IEEE 754-2008
VMLA	Multiply accumulate
VMLS	Multiply subtract
VMOV	Insert floating-point immediate in single-precision or double-precision register, or copy one FP register into another FP register of the same width
VMRS	Transfer contents from a floating-point system register to a general-purpose register
VMSR	Transfer contents from a general-purpose register to a floating-point system register
VMUL	Multiply
VNEG	Negate

Table C4-1 Summary of floating-point instructions (continued)

Mnemonic	Brief description
VNMLA	Negated multiply accumulate
VNMLS	Negated multiply subtract
VNMUL	Negated multiply
VPOP	Extension register load multiple
VPUSH	Extension register store multiple
VRINT	Round to integer
VSEL	Select
VSQRT	Square Root
VSTM	Extension register store multiple
VSTR	Extension register store
VSUB	Subtract

C4.2 VABS (floating-point)

Floating-point absolute value.

Syntax

`VABS{cond}.F32 Sd, Sm`

`VABS{cond}.F64 Dd, Dm`

where:

cond

is an optional condition code.

Sd, Sm

are the single-precision registers for the result and operand.

Dd, Dm

are the double-precision registers for the result and operand.

Operation

The `VABS` instruction takes the contents of *Sm* or *Dm*, clears the sign bit, and places the result in *Sd* or *Dd*. This gives the absolute value.

If the operand is a NaN, the sign bit is cleared, but no exception is produced.

Floating-point exceptions

`VABS` instructions do not produce any exceptions.

Related references

C1.9 Condition code suffixes on page C1-92

C4.3 VADD (floating-point)

Floating-point add.

Syntax

`VADD{cond}.F32 {Sd}, Sn, Sm`

`VADD{cond}.F64 {Dd}, Dn, Dm`

where:

cond

is an optional condition code.

Sd, Sn, Sm

are the single-precision registers for the result and operands.

Dd, Dn, Dm

are the double-precision registers for the result and operands.

Operation

The VADD instruction adds the values in the operand registers and places the result in the destination register.

Floating-point exceptions

The VADD instruction can produce Invalid Operation, Overflow, or Inexact exceptions.

Related references

C1.9 Condition code suffixes on page C1-92

C4.4 VCMP, VCMPE

Floating-point compare.

Syntax

`VCMP{E}{cond}.F32 Sd, Sm`

`VCMP{E}{cond}.F32 Sd, #0`

`VCMP{E}{cond}.F64 Dd, Dm`

`VCMP{E}{cond}.F64 Dd, #0`

where:

E

if present, indicates that the instruction raises an Invalid Operation exception if either operand is a quiet or signaling NaN. Otherwise, it raises the exception only if either operand is a signaling NaN.

cond

is an optional condition code.

Sd, Sm

are the single-precision registers holding the operands.

Dd, Dm

are the double-precision registers holding the operands.

Operation

The `VCMP{E}` instruction subtracts the value in the second operand register (or 0 if the second operand is #0) from the value in the first operand register, and sets the VFP condition flags based on the result.

Floating-point exceptions

`VCMP{E}` instructions can produce Invalid Operation exceptions.

Related references

C1.9 Condition code suffixes on page C1-92

C4.5 VCVT (between single-precision and double-precision)

Convert between single-precision and double-precision numbers.

Syntax

`VCVT{cond}.F64.F32 Dd, Sm`

`VCVT{cond}.F32.F64 Sd, Dm`

where:

cond

is an optional condition code.

Dd

is a double-precision register for the result.

Sm

is a single-precision register holding the operand.

Sd

is a single-precision register for the result.

Dm

is a double-precision register holding the operand.

Operation

These instructions convert the single-precision value in *Sm* to double-precision, placing the result in *Dd*, or the double-precision value in *Dm* to single-precision, placing the result in *Sd*.

Floating-point exceptions

These instructions can produce Invalid Operation, Input Denormal, Overflow, Underflow, or Inexact exceptions.

Related references

C1.9 Condition code suffixes on page C1-92

C4.6 VCVT (between floating-point and integer)

Convert between floating-point numbers and integers.

Syntax

`VCVT{R}{cond}.type.F64 Sd, Dm`

`VCVT{R}{cond}.type.F32 Sd, Sm`

`VCVT{cond}.F64.type Dd, Sm`

`VCVT{cond}.F32.type Sd, Sm`

where:

`R`

makes the operation use the rounding mode specified by the FPSCR. Otherwise, the operation rounds towards zero.

`cond`

is an optional condition code.

`type`

can be either `U32` (unsigned 32-bit integer) or `S32` (signed 32-bit integer).

`Sd`

is a single-precision register for the result.

`Dd`

is a double-precision register for the result.

`Sm`

is a single-precision register holding the operand.

`Dm`

is a double-precision register holding the operand.

Operation

The first two forms of this instruction convert from floating-point to integer.

The third and fourth forms convert from integer to floating-point.

Floating-point exceptions

These instructions can produce Input Denormal, Invalid Operation, or Inexact exceptions.

Related references

C1.9 Condition code suffixes on page C1-92

C4.7 VCVT (from floating-point to integer with directed rounding modes)

Convert from floating-point to signed or unsigned integer with directed rounding modes.

Note

This instruction is supported only in Armv8.

Syntax

`VCVT`*mode*`.S32.F64` *Sd, Dm*

`VCVT`*mode*`.S32.F32` *Sd, Sm*

`VCVT`*mode*`.U32.F64` *Sd, Dm*

`VCVT`*mode*`.U32.F32` *Sd, Sm*

where:

mode

must be one of:

A

meaning round to nearest, ties away from zero

N

meaning round to nearest, ties to even

P

meaning round towards plus infinity

M

meaning round towards minus infinity.

Sd, Sm

specifies the single-precision registers for the operand and result.

Sd, Dm

specifies a single-precision register for the result and double-precision register holding the operand.

Notes

You cannot use `VCVT` with a directed rounding mode inside an IT block.

Floating-point exceptions

These instructions can produce Input Denormal, Invalid Operation, or Inexact exceptions.

C4.8 VCVT (between floating-point and fixed-point)

Convert between floating-point and fixed-point numbers.

Syntax

`VCVT{cond}.type.F64 Dd, Dd, #fbits`

`VCVT{cond}.type.F32 Sd, Sd, #fbits`

`VCVT{cond}.F64.type Dd, Dd, #fbits`

`VCVT{cond}.F32.type Sd, Sd, #fbits`

where:

cond

is an optional condition code.

type

can be any one of:

S16

16-bit signed fixed-point number.

U16

16-bit unsigned fixed-point number.

S32

32-bit signed fixed-point number.

U32

32-bit unsigned fixed-point number.

Sd

is a single-precision register for the operand and result.

Dd

is a double-precision register for the operand and result.

fbits

is the number of fraction bits in the fixed-point number, in the range 0-16 if *type* is S16 or U16, or in the range 1-32 if *type* is S32 or U32.

Operation

The first two forms of this instruction convert from floating-point to fixed-point.

The third and fourth forms convert from fixed-point to floating-point.

In all cases the fixed-point number is contained in the least significant 16 or 32 bits of the register.

Floating-point exceptions

These instructions can produce Input Denormal, Invalid Operation, or Inexact exceptions.

Related references

C1.9 Condition code suffixes on page C1-92

C4.9 VCVTB, VCVTT (half-precision extension)

Convert between half-precision and single-precision floating-point numbers.

Syntax

`VCVTB{cond}.type Sd, Sm`

`VCVTT{cond}.type Sd, Sm`

where:

cond

is an optional condition code.

type

can be any one of:

F32.F16

Convert from half-precision to single-precision.

F16.F32

Convert from single-precision to half-precision.

Sd

is a single word register for the result.

Sm

is a single word register for the operand.

Operation

`VCVTB` uses the bottom half (bits[15:0]) of the single word register to obtain or store the half-precision value

`VCVTT` uses the top half (bits[31:16]) of the single word register to obtain or store the half-precision value.

Architectures

The instructions are only available in VFPv3 systems with the half-precision extension, and VFPv4.

Floating-point exceptions

These instructions can produce Input Denormal, Invalid Operation, Overflow, Underflow, or Inexact exceptions.

Related references

C1.9 Condition code suffixes on page C1-92

C4.10 VCVTB, VCVTT (between half-precision and double-precision)

These instructions convert between half-precision and double-precision floating-point numbers.

The conversion can be done in either of the following ways:

- From half-precision floating-point to double-precision floating-point (`F64.F16`).
- From double-precision floating-point to half-precision floating-point (`F16.F64`).

`VCVTB` uses the bottom half (bits[15:0]) of the single word register to obtain or store the half-precision value.

`VCVTT` uses the top half (bits[31:16]) of the single word register to obtain or store the half-precision value.

Note

These instructions are supported only in Armv8.

Syntax

`VCVTB{`*cond*`}.F64.F16` *Dd, Sm*

`VCVTB{`*cond*`}.F16.F64` *Sd, Dm*

`VCVTT{`*cond*`}.F64.F16` *Dd, Sm*

`VCVTT{`*cond*`}.F16.F64` *Sd, Dm*

where:

cond
: is an optional condition code.

Dd
: is a double-precision register for the result.

Sm
: is a single word register holding the operand.

Sd
: is a single word register for the result.

Dm
: is a double-precision register holding the operand.

Usage

These instructions convert the half-precision value in *Sm* to double-precision and place the result in *Dd*, or the double-precision value in *Dm* to half-precision and place the result in *Sd*.

Floating-point exceptions

These instructions can produce Input Denormal, Invalid Operation, Overflow, Underflow, or Inexact exceptions.

C4.11 VDIV

Floating-point divide.

Syntax

`VDIV{cond}.F32 {Sd}, Sn, Sm`

`VDIV{cond}.F64 {Dd}, Dn, Dm`

where:

cond

is an optional condition code.

Sd, Sn, Sm

are the single-precision registers for the result and operands.

Dd, Dn, Dm

are the double-precision registers for the result and operands.

Operation

The `VDIV` instruction divides the value in the first operand register by the value in the second operand register, and places the result in the destination register.

Floating-point exceptions

`VDIV` operations can produce Division by Zero, Invalid Operation, Overflow, Underflow, or Inexact exceptions.

Related references

C1.9 Condition code suffixes on page C1-92

C4.12 VFMA, VFMS, VFNMA, VFNMS (floating-point)

Fused floating-point multiply accumulate and fused floating-point multiply subtract, with optional negation.

Syntax

`VF{N}op{cond}.F64 {Dd}, Dn, Dm`

`VF{N}op{cond}.F32 {Sd}, Sn, Sm`

where:

op

is one of `MA` or `MS`.

N

negates the final result.

cond

is an optional condition code.

Sd, Sn, Sm

are the single-precision registers for the result and operands.

Dd, Dn, Dm

are the double-precision registers for the result and operands.

Operation

`VFMA` multiplies the values in the operand registers, adds the value in the destination register, and places the final result in the destination register. The result of the multiply is not rounded before the accumulation.

`VFMS` multiplies the values in the operand registers, subtracts the product from the value in the destination register, and places the final result in the destination register. The result of the multiply is not rounded before the subtraction.

In each case, the final result is negated if the `N` option is used.

Floating-point exceptions

These instructions can produce Input Denormal, Invalid Operation, Overflow, Underflow, or Inexact exceptions.

Related references

C4.28 VMUL (floating-point) on page C4-575
C1.9 Condition code suffixes on page C1-92

C4.13 VJCVT

Javascript Convert to signed fixed-point, rounding toward Zero.

Syntax

```
VJCVT{q}.S32.F64 Sd, Dm ; A1 FP/SIMD registers (A32)
VJCVT{q}.S32.F64 Sd, Dm ; T1 FP/SIMD registers (T32)
```

Where:

q
Is an optional instruction width specifier. See *C2.2 Instruction width specifiers* on page C2-111.

Sd
Is the 32-bit name of the SIMD and FP destination register.

Dm
Is the 64-bit name of the SIMD and FP source register.

Architectures supported

Supported in the Armv8.3-A architecture and later.

Usage

Javascript Convert to signed fixed-point, rounding toward Zero. This instruction converts the double-precision floating-point value in the SIMD and FP source register to a 32-bit signed integer using the Round towards Zero rounding mode, and write the result to the general-purpose destination register. If the result is too large to be held as a 32-bit signed integer, then the result is the integer modulo 2^{32}, as held in a 32-bit signed integer.

Depending on settings in the *CPACR*, *NSACR*, *HCPTR*, and *FPEXC* registers, and the security state and mode in which the instruction is executed, an 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 the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

C4.1 Summary of floating-point instructions on page C4-547

C4.14 VLDM (floating-point)

Extension register load multiple.

Syntax

`VLDM`*`mode`*`{`*`cond`*`} `*`Rn`*`{!}, `*`Registers`*

where:

`mode`

must be one of:

`IA`

meaning Increment address After each transfer. `IA` is the default, and can be omitted.

`DB`

meaning Decrement address Before each transfer.

`EA`

meaning Empty Ascending stack operation. This is the same as `DB` for loads.

`FD`

meaning Full Descending stack operation. This is the same as `IA` for loads.

`cond`

is an optional condition code.

`Rn`

is the general-purpose register holding the base address for the transfer.

`!`

is optional. `!` specifies that the updated base address must be written back to *`Rn`*. If `!` is not specified, *`mode`* must be `IA`.

`Registers`

is a list of consecutive extension registers enclosed in braces, `{` and `}`. The list can be comma-separated, or in range format. There must be at least one register in the list.

You can specify `S` or `D` registers, but they must not be mixed. The number of registers must not exceed 16 `D` registers.

Note

`VPOP` *`Registers`* is equivalent to `VLDM sp!,` *`Registers`*.

You can use either form of this instruction. They both disassemble to `VPOP`.

Related references

C1.9 Condition code suffixes on page C1-92

C4.15 VLDR (floating-point)

Extension register load.

Syntax

`VLDR{cond}{.size} Fd, [Rn{, #offset}]`

`VLDR{cond}{.size} Fd, label`

where:

cond

is an optional condition code.

size

is an optional data size specifier. Must be 32 if *Fd* is an `S` register, or 64 otherwise.

Fd

is the extension register to be loaded, and can be either a `D` or `S` register.

Rn

is the general-purpose register holding the base address for the transfer.

offset

is an optional numeric expression. It must evaluate to a numeric value at assembly time. The value must be a multiple of 4, and lie in the range -1020 to +1020. The value is added to the base address to form the address used for the transfer.

label

is a PC-relative expression.

label must be aligned on a word boundary within ±1KB of the current instruction.

Operation

The `VLDR` instruction loads an extension register from memory.

One word is transferred if *Fd* is an `S` register. Two words are transferred otherwise.

There is also a `VLDR` pseudo-instruction.

Related references

C1.9 Condition code suffixes on page C1-92

C4.16 VLDR (post-increment and pre-decrement, floating-point)

Pseudo-instruction that loads extension registers, with post-increment and pre-decrement forms.

Note

There are also `VLDR` and `VSTR` instructions without post-increment and pre-decrement.

Syntax

`VLDR{`*cond*`}{.`*size*`} `*Fd*`, [`*Rn*`], #`*offset* ; post-increment

`VLDR{`*cond*`}{.`*size*`} `*Fd*`, [`*Rn*`, #-`*offset*`]! ; pre-decrement

where:

cond

is an optional condition code.

size

is an optional data size specifier. Must be 32 if *Fd* is an `S` register, or 64 if *Fd* is a `D` register.

Fd

is the extension register to load. It can be either a double precision (*Dd*) or a single precision (*Sd*) register.

Rn

is the general-purpose register holding the base address for the transfer.

offset

is a numeric expression that must evaluate to a numeric value at assembly time. The value must be `4` if *Fd* is an `S` register, or `8` if *Fd* is a `D` register.

Operation

The post-increment instruction increments the base address in the register by the offset value, after the transfer. The pre-decrement instruction decrements the base address in the register by the offset value, and then performs the transfer using the new address in the register. This pseudo-instruction assembles to a `VLDM` instruction.

Related references

C4.14 VLDM (floating-point) on page C4-561

C4.15 VLDR (floating-point) on page C4-562

C1.9 Condition code suffixes on page C1-92

C4.17 VLLDM

Floating-point Lazy Load Multiple.

Syntax

`VLLDM{c}{q} Rn`

Where:

c
Is an optional condition code. See *Chapter C1 Condition Codes* on page C1-83.

q
Is an optional instruction width specifier. See *C2.2 Instruction width specifiers* on page C2-111.

Rn
Is the general-purpose base register.

Architectures supported

Supported in Armv8-M Main extension only.

Usage

Floating-point Lazy Load Multiple restores the contents of the Secure floating-point registers that were protected by a VLSTM instruction, and marks the floating-point context as active.

If the lazy state preservation set up by a previous VLSTM instruction is active (FPCCR.LSPACT == 1), this instruction deactivates lazy state preservation and enables access to the Secure floating-point registers.

If lazy state preservation is inactive (FPCCR.LSPACT == 0), either because lazy state preservation was not enabled (FPCCR.LSPEN == 0) or because a floating-point instruction caused the Secure floating-point register contents to be stored to memory, this instruction loads the stored Secure floating-point register contents back into the floating-point registers.

If Secure floating-point is not in use (CONTROL_S.SFPA == 0), this instruction behaves as a NOP.

This instruction is only available in Secure state, and is UNDEFINED in Non-secure state.

If the Floating-point Extension is not implemented, this instruction is available in Secure state, but behaves as a NOP.

Related references
C4.1 Summary of floating-point instructions on page C4-547

C4.18 VLSTM

Floating-point Lazy Store Multiple.

Syntax

`VLSTM{c}{q} Rn`

Where:

c
Is an optional condition code. See *Chapter C1 Condition Codes* on page C1-83.

q
Is an optional instruction width specifier. See *C2.2 Instruction width specifiers* on page C2-111.

Rn
Is the general-purpose base register.

Architectures supported

Supported in Armv8-M Main extension only.

Usage

Floating-point Lazy Store Multiple stores the contents of Secure floating-point registers to a prepared stack frame, and clears the Secure floating-point registers.

If floating-point lazy preservation is enabled (FPCCR.LSPEN == 1), then the next time a floating-point instruction other than VLSTM or VLLDM is executed:

- The contents of Secure floating-point registers are stored to memory.
- The Secure floating-point registers are cleared.

If Secure floating-point is not in use (CONTROL_S.SFPA == 0), this instruction behaves as a NOP.

This instruction is only available in Secure state, and is UNDEFINED in Non-secure state.

If the Floating-point extension is not implemented, this instruction is available in Secure state, but behaves as a NOP.

Related references
C4.1 Summary of floating-point instructions on page C4-547

C4.19 VMAXNM, VMINNM (floating-point)

Vector Minimum, Vector Maximum.

Note

These instructions are supported only in Armv8.

Syntax

`Vop.F32 Sd, Sn, Sm`

`Vop.F64 Dd, Dn, Dm`

where:

op
must be either `MAXNM` or `MINNM`.

Sd, Sn, Sm
are the single-precision destination register, first operand register, and second operand register.

Dd, Dn, Dm
are the double-precision destination register, first operand register, and second operand register.

Operation

`VMAXNM` compares the values in the operand registers, and copies the larger value into the destination operand register.

`VMINNM` compares the values in the operand registers, and copies the smaller value into the destination operand register.

If one of the values being compared is a number and the other value is NaN, the number is copied into the destination operand register. This is consistent with the IEEE 754-2008 standard.

Notes

You cannot use `VMAXNM` or `VMINNM` inside an IT block.

Floating-point exceptions

These instructions can produce Input Denormal, Invalid Operation, Overflow, Underflow, or Inexact exceptions.

C4.20 VMLA (floating-point)

Floating-point multiply accumulate.

Syntax

`VMLA{cond}.F32 Sd, Sn, Sm`

`VMLA{cond}.F64 Dd, Dn, Dm`

where:

cond

is an optional condition code.

Sd, Sn, Sm

are the single-precision registers for the result and operands.

Dd, Dn, Dm

are the double-precision registers for the result and operands.

Operation

The `VMLA` instruction multiplies the values in the operand registers, adds the value in the destination register, and places the final result in the destination register.

Floating-point exceptions

This instruction can produce Invalid Operation, Overflow, Underflow, Inexact, or Input Denormal exceptions.

Related references

C1.9 Condition code suffixes on page C1-92

C4.21 VMLS (floating-point)

Floating-point multiply subtract.

Syntax

`VMLS{cond}.F32 Sd, Sn, Sm`

`VMLS{cond}.F64 Dd, Dn, Dm`

where:

cond

is an optional condition code.

Sd, Sn, Sm

are the single-precision registers for the result and operands.

Dd, Dn, Dm

are the double-precision registers for the result and operands.

Operation

The `VMLS` instruction multiplies the values in the operand registers, subtracts the result from the value in the destination register, and places the final result in the destination register.

Floating-point exceptions

This instruction can produce Invalid Operation, Overflow, Underflow, Inexact, or Input Denormal exceptions.

Related references

C1.9 Condition code suffixes on page C1-92

C4.22 VMOV (floating-point)

Insert a floating-point immediate value into a single-precision or double-precision register, or copy one register into another register. This instruction is always scalar.

Syntax

`VMOV{cond}.F32 Sd, #imm`

`VMOV{cond}.F64 Dd, #imm`

`VMOV{cond}.F32 Sd, Sm`

`VMOV{cond}.F64 Dd, Dm`

where:

cond

is an optional condition code.

Sd

is the single-precision destination register.

Dd

is the double-precision destination register.

imm

is the floating-point immediate value.

Sm

is the single-precision source register.

Dm

is the double-precision source register.

Immediate values

Any number that can be expressed as $\pm n * 2^{-r}$,where n and r are integers, $16 <= n <= 31$, $0 <= r <= 7$.

Architectures

The instructions that copy immediate constants are available in VFPv3 and above.

The instructions that copy from registers are available in all VFP systems.

Related references

C1.9 Condition code suffixes on page C1-92

C4.23 VMOV (between one general-purpose register and single precision floating-point register)

Transfer contents between a single-precision floating-point register and a general-purpose register.

Syntax

`VMOV{cond} Rd, Sn`

`VMOV{cond} Sn, Rd`

where:

cond

is an optional condition code.

Sn

is the floating-point single-precision register.

Rd

is the general-purpose register. *Rd* must not be PC.

Operation

`VMOV Rd, Sn` transfers the contents of *Sn* into *Rd*.

`VMOV Sn, Rd` transfers the contents of *Rd* into *Sn*.

Related references

C1.9 Condition code suffixes on page C1-92

C4.24 VMOV (between two general-purpose registers and one or two extension registers)

Transfer contents between two general-purpose registers and either one 64-bit register or two consecutive 32-bit registers.

Syntax

`VMOV{cond} Dm, Rd, Rn`

`VMOV{cond} Rd, Rn, Dm`

`VMOV{cond} Sm, Sm1, Rd, Rn`

`VMOV{cond} Rd, Rn, Sm, Sm1`

where:

cond

is an optional condition code.

Dm

is a 64-bit extension register.

Sm

is a VFP 32-bit register.

Sm1

is the next consecutive VFP 32-bit register after *Sm*.

Rd, Rn

are the general-purpose registers. R*d* and R*n* must not be PC.

Operation

`VMOV Dm, Rd, Rn` transfers the contents of *Rd* into the low half of *Dm*, and the contents of *Rn* into the high half of *Dm*.

`VMOV Rd, Rn, Dm` transfers the contents of the low half of *Dm* into *Rd*, and the contents of the high half of *Dm* into *Rn*.

`VMOV Rd, Rn, Sm, Sm1` transfers the contents of *Sm* into *Rd*, and the contents of *Sm1* into *Rn*.

`VMOV Sm, Sm1, Rd, Rn` transfers the contents of *Rd* into *Sm*, and the contents of *Rn* into *Sm1*.

Architectures

The instructions are available in VFPv2 and above.

Related references

C1.9 Condition code suffixes on page C1-92

C4.25 VMOV (between a general-purpose register and half a double precision floating-point register)

Transfer contents between a general-purpose register and half a double precision floating-point register.

Syntax

`VMOV{cond}{.size} Dn[x], Rd`

`VMOV{cond}{.size} Rd, Dn[x]`

where:

cond

is an optional condition code.

size

the data size. Must be either `32` or omitted. If omitted, *size* is `32`.

Dn[*x*]

is the upper or lower half of a double precision floating-point register.

Rd

is the general-purpose register. *Rd* must not be PC.

Operation

`VMOV` *Dn*`[x], ` *Rd* transfers the contents of *Rd* into *Dn*[x].

`VMOV` *Rd*`, ` *Dn*`[x]` transfers the contents of *Dn*[x] into *Rd*.

Related references

C1.9 Condition code suffixes on page C1-92

C4.26 VMRS (floating-point)

Transfer contents from an floating-point system register to a general-purpose register.

Syntax

`VMRS{`*`cond`*`} `*`Rd, extsysreg`*

where:

`cond`
is an optional condition code.

`extsysreg`
is the floating-point system register, usually `FPSCR`, `FPSID`, or `FPEXC`.

`Rd`
is the general-purpose register. *`Rd`* must not be PC.

It can be `APSR_nzcv`, if *`extsysreg`* is `FPSCR`. In this case, the floating-point status flags are transferred into the corresponding flags in the special-purpose APSR.

Usage

The `VMRS` instruction transfers the contents of *`extsysreg`* into *`Rd`*.

Note

The instruction stalls the processor until all current floating-point operations complete.

Examples

```
    VMRS    r2,FPCID
    VMRS    APSR_nzcv, FPSCR     ; transfer FP status register to the
                                 ; special-purpose APSR
```

Related references

C1.9 Condition code suffixes on page C1-92

C4.27 VMSR (floating-point)

Transfer contents of a general-purpose register to a floating-point system register.

Syntax

`VMSR{`*`cond`*`} `*`extsysreg, Rd`

where:

`cond`
is an optional condition code.

`extsysreg`
is the floating-point system register, usually `FPSCR`, `FPSID`, or `FPEXC`.

`Rd`
is the general-purpose register. *`Rd`* must not be PC.

It can be `APSR_nzcv`, if *`extsysreg`* is `FPSCR`. In this case, the floating-point status flags are transferred into the corresponding flags in the special-purpose APSR.

Usage

The `VMSR` instruction transfers the contents of *`Rd`* into *`extsysreg`*.

Note

The instruction stalls the processor until all current floating-point operations complete.

Example

```
    VMSR    FPSCR, r4
```

Related references

C1.9 Condition code suffixes on page C1-92

C4.28 VMUL (floating-point)

Floating-point multiply.

Syntax

`VMUL{cond}.F32 {Sd,} Sn, Sm`

`VMUL{cond}.F64 {Dd,} Dn, Dm`

where:

cond

is an optional condition code.

Sd, Sn, Sm

are the single-precision registers for the result and operands.

Dd, Dn, Dm

are the double-precision registers for the result and operands.

Operation

The `VMUL` operation multiplies the values in the operand registers and places the result in the destination register.

Floating-point exceptions

This instruction can produce Invalid Operation, Overflow, Underflow, Inexact, or Input Denormal exceptions.

Related references

C1.9 Condition code suffixes on page C1-92

C4.29 VNEG (floating-point)

Floating-point negate.

Syntax

`VNEG{cond}.F32 Sd, Sm`

`VNEG{cond}.F64 Dd, Dm`

where:

cond

is an optional condition code.

Sd, Sm

are the single-precision registers for the result and operand.

Dd, Dm

are the double-precision registers for the result and operand.

Operation

The `VNEG` instruction takes the contents of *Sm* or *Dm*, changes the sign bit, and places the result in *Sd* or *Dd*. This gives the negation of the value.

If the operand is a NaN, the sign bit is changed, but no exception is produced.

Floating-point exceptions

`VNEG` instructions do not produce any exceptions.

Related references

C1.9 Condition code suffixes on page C1-92

C4.30 VNMLA (floating-point)

Floating-point multiply accumulate with negation.

Syntax

`VNMLA{cond}.F32 Sd, Sn, Sm`

`VNMLA{cond}.F64 Dd, Dn, Dm`

where:

cond

is an optional condition code.

Sd, Sn, Sm

are the single-precision registers for the result and operands.

Dd, Dn, Dm

are the double-precision registers for the result and operands.

Operation

The `VNMLA` instruction multiplies the values in the operand registers, adds the value to the destination register, and places the negated final result in the destination register.

Floating-point exceptions

This instruction can produce Invalid Operation, Overflow, Underflow, Inexact, or Input Denormal exceptions.

Related references

C1.9 Condition code suffixes on page C1-92

C4.31 VNMLS (floating-point)

Floating-point multiply subtract with negation.

Syntax

`VNMLS{cond}.F32 Sd, Sn, Sm`

`VNMLS{cond}.F64 Dd, Dn, Dm`

where:

cond

is an optional condition code.

Sd, Sn, Sm

are the single-precision registers for the result and operands.

Dd, Dn, Dm

are the double-precision registers for the result and operands.

Operation

The `VNMLS` instruction multiplies the values in the operand registers, subtracts the result from the value in the destination register, and places the negated final result in the destination register.

Floating-point exceptions

This instruction can produce Invalid Operation, Overflow, Underflow, Inexact, or Input Denormal exceptions.

Related references

C1.9 Condition code suffixes on page C1-92

C4.32 VNMUL (floating-point)

Floating-point multiply with negation.

Syntax

`VNMUL{cond}.F32 {Sd,} Sn, Sm`

`VNMUL{cond}.F64 {Dd,} Dn, Dm`

where:

cond

is an optional condition code.

Sd, Sn, Sm

are the single-precision registers for the result and operands.

Dd, Dn, Dm

are the double-precision registers for the result and operands.

Operation

The `VNMUL` instruction multiplies the values in the operand registers and places the negated result in the destination register.

Floating-point exceptions

This instruction can produce Invalid Operation, Overflow, Underflow, Inexact, or Input Denormal exceptions.

Related references

C1.9 Condition code suffixes on page C1-92

C4.33 VPOP (floating-point)

Pop extension registers from the stack.

Syntax

`VPOP{cond} Registers`

where:

cond

is an optional condition code.

Registers

is a list of consecutive extension registers enclosed in braces, `{` and `}`. The list can be comma-separated, or in range format. There must be at least one register in the list.

You can specify `S` or `D` registers, but they must not be mixed. The number of registers must not exceed 16 `D` registers.

Note

`VPOP Registers` is equivalent to `VLDM sp!, Registers`.

You can use either form of this instruction. They both disassemble to `VPOP`.

Related references

C1.9 Condition code suffixes on page C1-92

C4.34 VPUSH (floating-point) on page C4-581

C4.34 VPUSH (floating-point)

Push extension registers onto the stack.

Syntax

`VPUSH{cond} Registers`

where:

cond

is an optional condition code.

Registers

is a list of consecutive extension registers enclosed in braces, `{` and `}`. The list can be comma-separated, or in range format. There must be at least one register in the list.

You can specify `S` or `D` registers, but they must not be mixed. The number of registers must not exceed 16 `D` registers.

Note

`VPUSH Registers` is equivalent to `VSTMDB sp!, Registers`.

You can use either form of this instruction. They both disassemble to `VPUSH`.

Related references

C1.9 Condition code suffixes on page C1-92

C4.33 VPOP (floating-point) on page C4-580

C4.35 VRINT (floating-point)

Rounds a floating-point number to integer and places the result in the destination register. The resulting integer is represented in floating-point format.

Note

This instruction is supported only in Armv8.

Syntax

`VRINT`*mode*`{`*cond*`}.F64.F64` *Dd, Dm*

`VRINT`*mode*`{`*cond*`}.F32.F32` *Sd, Sm*

where:

mode

must be one of:

Z
meaning round towards zero.

R
meaning use the rounding mode specified in the FPSCR.

X
meaning use the rounding mode specified in the FPSCR, generating an Inexact exception if the result is not exact.

A
meaning round to nearest, ties away from zero.

N
meaning round to nearest, ties to even.

P
meaning round towards plus infinity.

M
meaning round towards minus infinity.

cond
is an optional condition code. This can only be used when *mode* is `Z`, `R` or `X`.

Sd, Sm
specifies the destination and operand registers, for a word operation.

Dd, Dm
specifies the destination and operand registers, for a doubleword operation.

Notes

You cannot use `VRINT` with a rounding mode of `A`, `N`, `P` or `M` inside an IT block.

Floating-point exceptions

These instructions cannot produce any exceptions, except `VRINTX` which can generate an Inexact exception.

Related references

C1.9 Condition code suffixes on page C1-92

C4.36 VSEL

Floating-point select.

Note

This instruction is supported only in Armv8.

Syntax

`VSEL`*`cond`*`.F32` *`Sd, Sn, Sm`*

`VSEL`*`cond`*`.F64` *`Dd, Dn, Dm`*

where:

`cond`
: must be one of `GE`, `GT`, `EQ`, `VS`.

`Sd, Sn, Sm`
: are the single-precision registers for the result and operands.

`Dd, Dn, Dm`
: are the double-precision registers for the result and operands.

Usage

The `VSEL` instruction compares the values in the operand registers. If the condition is true, it copies the value in the first operand register into the destination operand register. Otherwise, it copies the value in the second operand register.

You cannot use `VSEL` inside an IT block.

Floating-point exceptions

`VSEL` instructions cannot produce any exceptions.

Related references

C1.11 Comparison of condition code meanings in integer and floating-point code on page C1-94

C1.9 Condition code suffixes on page C1-92

C4.37 VSQRT

Floating-point square root.

Syntax

`VSQRT{cond}.F32 Sd, Sm`

`VSQRT{cond}.F64 Dd, Dm`

where:

cond

is an optional condition code.

Sd, Sm

are the single-precision registers for the result and operand.

Dd, Dm

are the double-precision registers for the result and operand.

Operation

The `VSQRT` instruction takes the square root of the contents of *Sm* or *Dm*, and places the result in *Sd* or *Dd*.

Floating-point exceptions

`VSQRT` instructions can produce Invalid Operation or Inexact exceptions.

Related references

C1.9 Condition code suffixes on page C1-92

C4.38 VSTM (floating-point)

Extension register store multiple.

Syntax

`VSTM`*`mode`*`{`*`cond`*`} `*`Rn`*`{!}, `*`Registers`*

where:

`mode`

must be one of:

`IA`

meaning Increment address After each transfer. `IA` is the default, and can be omitted.

`DB`

meaning Decrement address Before each transfer.

`EA`

meaning Empty Ascending stack operation. This is the same as `IA` for stores.

`FD`

meaning Full Descending stack operation. This is the same as `DB` for stores.

`cond`

is an optional condition code.

`Rn`

is the general-purpose register holding the base address for the transfer.

`!`

is optional. `!` specifies that the updated base address must be written back to *`Rn`*. If `!` is not specified, *`mode`* must be `IA`.

`Registers`

is a list of consecutive extension registers enclosed in braces, `{` and `}`. The list can be comma-separated, or in range format. There must be at least one register in the list.

You can specify `S` or `D` registers, but they must not be mixed. The number of registers must not exceed 16 `D` registers.

Note

`VPUSH` *`Registers`* is equivalent to `VSTMDB sp!,` *`Registers`*.

You can use either form of this instruction. They both disassemble to `VPUSH`.

Related references

C1.9 Condition code suffixes on page C1-92

C4.39 VSTR (floating-point)

Extension register store.

Syntax

`VSTR{cond}{.size} Fd, [Rn{, #offset}]`

where:

cond

is an optional condition code.

size

is an optional data size specifier. Must be 32 if *Fd* is an `S` register, or 64 otherwise.

Fd

is the extension register to be saved. It can be either a `D` or `S` register.

Rn

is the general-purpose register holding the base address for the transfer.

offset

is an optional numeric expression. It must evaluate to a numeric value at assembly time. The value must be a multiple of 4, and lie in the range -1020 to +1020. The value is added to the base address to form the address used for the transfer.

Operation

The `VSTR` instruction saves the contents of an extension register to memory.

One word is transferred if *Fd* is an `S` register. Two words are transferred otherwise.

Related references

C1.9 Condition code suffixes on page C1-92

C4.40 VSTR (post-increment and pre-decrement, floating-point)

Pseudo-instruction that stores extension registers with post-increment and pre-decrement forms.

Note

There are also `VLDR` and `VSTR` instructions without post-increment and pre-decrement.

Syntax

`VSTR{`*cond*`}{.`*size*`} Fd, [Rn], #`*offset* `; post-increment`

`VSTR{`*cond*`}{.`*size*`} Fd, [Rn, #-`*offset*`]! ; pre-decrement`

where:

cond

is an optional condition code.

size

is an optional data size specifier. Must be 32 if *Fd* is an `S` register, or 64 if *Fd* is a `D` register.

Fd

is the extension register to be saved. It can be either a double precision (*Dd*) or a single precision (*Sd*) register.

Rn

is the general-purpose register holding the base address for the transfer.

offset

is a numeric expression that must evaluate to a numeric value at assembly time. The value must be `4` if *Fd* is an `S` register, or `8` if *Fd* is a `D` register.

Operation

The post-increment instruction increments the base address in the register by the offset value, after the transfer. The pre-decrement instruction decrements the base address in the register by the offset value, and then performs the transfer using the new address in the register. This pseudo-instruction assembles to a `VSTM` instruction.

Related references

C4.39 VSTR (floating-point) on page C4-586
C4.38 VSTM (floating-point) on page C4-585
C1.9 Condition code suffixes on page C1-92

C4.41 VSUB (floating-point)

Floating-point subtract.

Syntax

`VSUB{cond}.F32 {Sd}, Sn, Sm`

`VSUB{cond}.F64 {Dd}, Dn, Dm`

where:

cond

is an optional condition code.

Sd, Sn, Sm

are the single-precision registers for the result and operands.

Dd, Dn, Dm

are the double-precision registers for the result and operands.

Operation

The `VSUB` instruction subtracts the value in the second operand register from the value in the first operand register, and places the result in the destination register.

Floating-point exceptions

The `VSUB` instruction can produce Invalid Operation, Overflow, or Inexact exceptions.

Related references

C1.9 Condition code suffixes on page C1-92

Chapter C5
A32/T32 Cryptographic Algorithms

Lists the cryptographic algorithms that A32 and T32 SIMD instructions support.

It contains the following section:

C5.1 A32/T32 Cryptographic instructions

A set of A32 and T32 cryptographic instructions is available in the Armv8 architecture.

These instructions use the 128-bit Advanced SIMD registers and support the acceleration of the following cryptographic and hash algorithms:

- AES.
- SHA1.
- SHA256.

Summary of A32/T32 cryptographic instructions

The following table lists the A32/T32 cryptographic instructions that are supported:

Table C5-1 Summary of A32/T32 cryptographic instructions

Mnemonic	Brief description
AESD	AES single round decryption
AESE	AES single round encryption
AESIMC	AES inverse mix columns
AESMC	AES mix columns
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
SHA256H2	SHA256 hash update part 2
SHA256H	SHA256 hash update part 1
SHA256SU0	SHA256 schedule update 0
SHA256SU1	SHA256 schedule update 1