Arm® Compiler

Version 6.6

armasm User Guide

arm

DUI0801I

Arm® Compiler

armasm User Guide

Release Information

Document History

Issue	Date	Confidentiality	Change
A	14 March 2014	Non-Confidential	Arm Compiler v6.00 Release
B	15 December 2014	Non-Confidential	Arm Compiler v6.01 Release
C	30 June 2015	Non-Confidential	Arm Compiler v6.02 Release
D	18 November 2015	Non-Confidential	Arm Compiler v6.3 Release
E	24 February 2016	Non-Confidential	Arm Compiler v6.4 Release
F	29 June 2016	Non-Confidential	Arm Compiler v6.5 Release
G	04 November 2016	Non-Confidential	Arm Compiler v6.6 Release
H	08 May 2017	Non-Confidential	Arm Compiler v6.6.1 Release
I	29 November 2017	Non-Confidential	Arm Compiler v6.6.2 Release

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

http://www.arm.com

Contents
Arm® Compiler armasm User Guide

Chapter 18 A64 Floating-point Instructions

List of Figures
Arm® Compiler armasm User Guide

List of Tables
Arm® Compiler armasm User Guide

Preface

This preface introduces the *Arm® Compiler armasm User Guide*.

It contains the following:

- *About this book* on page 43.

About this book

Arm® Compiler armasm User Guide. This document provides topic based documentation for using the Arm assembler (armasm). It contains information on command line options, A32, T32, and A64 instruction sets, Advanced SIMD and floating-point instructions, assembler directives, and supports the Armv6-M, Armv7, and Armv8 architectures.

Using this book

This book is organized into the following chapters:

Chapter 1 Overview of the Assembler
Gives an overview of the assemblers provided with Arm Compiler toolchain.

Chapter 2 Overview of the Arm®v8 Architecture
Gives an overview of the Armv8 architecture.

Chapter 3 Overview of AArch32 state
Gives an overview of the AArch32 state of Armv8.

Chapter 4 Overview of AArch64 state
Gives an overview of the AArch64 state of Armv8.

Chapter 5 Structure of Assembly Language Modules
Describes the structure of assembly language source files.

Chapter 6 Writing A32/T32 Assembly Language
Describes the use of a few basic A32 and T32 instructions and the use of macros.

Chapter 7 Condition Codes
Describes condition codes and conditional execution of A64, A32, and T32 code.

Chapter 8 Using armasm
Describes how to use `armasm`.

Chapter 9 Advanced SIMD Programming
Describes Advanced SIMD assembly language programming.

Chapter 10 Floating-point Programming
Describes floating-point assembly language programming.

Chapter 11 armasm Command-line Options
Describes the `armasm` command-line syntax and command-line options.

Chapter 12 Symbols, Literals, Expressions, and Operators
Describes how you can use symbols to represent variables, addresses, and constants in code, and how you can combine these with operators to create numeric or string expressions.

Chapter 13 A32 and T32 Instructions
Describes the A32 and T32 instructions supported in AArch32 state.

Chapter 14 Advanced SIMD Instructions (32-bit)
Describes Advanced SIMD assembly language instructions.

Chapter 15 Floating-point Instructions (32-bit)
Describes floating-point assembly language instructions.

Chapter 16 A64 General Instructions
Describes the A64 general instructions.

Chapter 17 A64 Data Transfer Instructions
Describes the A64 data transfer instructions.

Chapter 18 A64 Floating-point Instructions
Describes the A64 floating-point instructions.

Chapter 19 A64 SIMD Scalar Instructions
Describes the A64 SIMD scalar instructions.

Chapter 20 A64 SIMD Vector Instructions
Describes the A64 SIMD vector instructions.

Chapter 21 Directives Reference
Describes the directives that are provided by the Arm assembler, `armasm`.

Chapter 22 Via File Syntax
Describes the syntax of via files accepted by `armasm`.

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 *Arm Compiler armasm User Guide*.
- The number DUI0801I.
- 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*.

Chapter 1
Overview of the Assembler

Gives an overview of the assemblers provided with Arm Compiler toolchain.

It contains the following sections:

1.1 About the Arm® Compiler toolchain assemblers

The Arm Compiler toolchain provides different assemblers.

They are:

- The freestanding legacy assembler, `armasm`. Use `armasm` to assemble existing A64, A32, and T32 assembly language code written in armasm syntax.
- The `armclang` integrated assembler. Use this to assemble assembly language code written in GNU syntax.
- An optimizing inline assembler built into `armclang`. Use this to assemble assembly language code written in GNU syntax that is used inline in C or C++ source code.

Note

This book only applies to `armasm`. For information on `armclang`, see the *armclang Reference Guide*.

Note

Be aware of the following:

- Generated code might be different between two Arm Compiler releases.
- For a feature release, there might be significant code generation differences.

Note

The command-line option descriptions and related information in the individual Arm Compiler tools documents describe all the features that Arm Compiler supports. Any features not documented are not supported and are used at your own risk. You are responsible for making sure that any generated code using *community features* on page 1-53 is operating correctly.

Related information

Arm Compiler armclang Reference Guide.
Mixing Assembly Code with C or C++ Code.
Assembling armasm and GNU syntax assembly code.

1.2 Key features of the armasm assembler

The `armasm` assembler supports instructions, directives, and user-defined macros.

It supports:

- *Unified Assembly Language* (UAL) for both A32 and T32 code.
- Assembly language for A64 code.
- Advanced SIMD instructions in A64, A32, and T32 code.
- Floating-point instructions in A64, A32, and T32 code.
- Directives in assembly source code.
- Processing of user-defined macros.

Related concepts

1.3 How the assembler works on page 1-49.
6.1 About the Unified Assembler Language on page 6-102.
9.1 Architecture support for Advanced SIMD on page 9-182.
6.22 Use of macros on page 6-130.

Related references

Chapter 9 Advanced SIMD Programming on page 9-181.
Chapter 21 Directives Reference on page 21-1642.

1.3 How the assembler works

`armasm` reads the assembly language source code twice before it outputs object code. Each read of the source code is called a pass.

This is because assembly language source code often contains forward references. A forward reference occurs when a label is used as an operand, for example as a branch target, earlier in the code than the definition of the label. The assembler cannot know the address of the forward reference label until it reads the definition of the label.

During each pass, the assembler performs different functions. In the first pass, the assembler:

- Checks the syntax of the instruction or directive. It faults if there is an error in the syntax, for example if a label is specified on a directive that does not accept one.
- Determines the size of the instruction and data being assembled and reserves space.
- Determines offsets of labels within sections.
- Creates a symbol table containing label definitions and their memory addresses.

In the second pass, the assembler:

- Faults if an undefined reference is specified in an instruction operand or directive.
- Encodes the instructions using the label offsets from pass 1, where applicable.
- Generates relocations.
- Generates debug information if requested.
- Outputs the object file.

Memory addresses of labels are determined and finalized in the first pass. Therefore, the assembly code must not change during the second pass. All instructions must be seen in both passes. Therefore you must not define a symbol after a `:DEF:` test for the symbol. The assembler faults if it sees code in pass 2 that was not seen in pass 1.

Line not seen in pass 1

The following example shows that `num EQU 42` is not seen in pass 1 but is seen in pass 2:

```
    AREA x,CODE
    [ :DEF: foo
num EQU 42
    ]
foo DCD num
    END
```

Assembling this code generates the error:

```
A1903E: Line not seen in first pass; cannot be assembled.
```

Line not seen in pass 2

The following example shows that `MOV r1,r2` is seen in pass 1 but not in pass 2:

```
    AREA x,CODE
    [ :LNOT: :DEF: foo
    MOV r1, r2
    ]
foo MOV r3, r4
    END
```

Assembling this code generates the error:

```
A1909E: Line not seen in second pass; cannot be assembled.
```

Related concepts

8.13 Two pass assembler diagnostics on page 8-175.
6.25 Instruction and directive relocations on page 6-134.

Related references

1.4 Directives that can be omitted in pass 2 of the assembler on page 1-51.
11.17 --diag_error=tag[,tag,...] on page 11-245.
11.14 --debug on page 11-242.

1.4 Directives that can be omitted in pass 2 of the assembler

Most directives must appear in both passes of the assembly process. You can omit some directives from the second pass over the source code by the assembler, but doing this is strongly discouraged.

Directives that can be omitted from pass 2 are:

- `GBLA`, `GBLL`, `GBLS`.
- `LCLA`, `LCLL`, `LCLS`.
- `SETA`, `SETL`, `SETS`.
- `RN`, `RLIST`.
- `CN`, `CP`.
- `SN`, `DN`, `QN`.
- `EQU`.
- `MAP`, `FIELD`.
- `GET`, `INCLUDE`.
- `IF`, `ELSE`, `ELIF`, `ENDIF`.
- `WHILE`, `WEND`.
- `ASSERT`.
- `ATTR`.
- `COMMON`.
- `EXPORTAS`.
- `IMPORT`.
- `EXTERN`.
- `KEEP`.
- `MACRO`, `MEND`, `MEXIT`.
- `REQUIRE8`.
- `PRESERVE8`.

Note

Macros that appear only in pass 1 and not in pass 2 must contain only these directives.

ASSERT directive appears in pass 1 only

The code in the following example assembles without error although the `ASSERT` directive does not appear in pass 2:

```
    AREA ||.text||,CODE
x   EQU 42
    IF :LNOT: :DEF: sym
        ASSERT x == 42
    ENDIF
sym EQU 1
    END
```

Use of ELSE and ELIF directives

Directives that appear in pass 2 but do not appear in pass 1 cause an assembly error. However, this does not cause an assembly error when using the `ELSE` and `ELIF` directives if their matching `IF` directive appears in pass 1. The following example assembles without error because the `IF` directive appears in pass 1:

```
    AREA ||.text||,CODE
x   EQU 42
    IF :DEF: sym
    ELSE
        ASSERT x == 42
    ENDIF
sym EQU 1
    END
```

Related concepts

1.3 How the assembler works on page 1-49.
8.13 Two pass assembler diagnostics on page 8-175.

1.5 Support level definitions

This describes the levels of support for various Arm Compiler 6 features.

Arm Compiler 6 is built on Clang and LLVM technology. Therefore it has more functionality than the set of product features described in the documentation. The following definitions clarify the levels of support and guarantees on functionality that are expected from these features.

Arm welcomes feedback regarding the use of all Arm Compiler 6 features, and endeavors to support users to a level that is appropriate for that feature. You can contact support at *https://developer.arm.com/support*.

Identification in the documentation

All features that are documented in the Arm Compiler 6 documentation are product features, except where explicitly stated. The limitations of non-product features are explicitly stated.

Product features

Product features are suitable for use in a production environment. The functionality is well-tested, and is expected to be stable across feature and update releases.

- Arm endeavors to give advance notice of significant functionality changes to product features.
- If you have a support and maintenance contract, Arm provides full support for use of all product features.
- Arm welcomes feedback on product features.
- Any issues with product features that Arm encounters or is made aware of are considered for fixing in future versions of Arm Compiler.

In addition to fully supported product features, some product features are only alpha or beta quality.

Beta product features

Beta product features are implementation complete, but have not been sufficiently tested to be regarded as suitable for use in production environments.
Beta product features are indicated with [BETA].

- Arm endeavors to document known limitations on beta product features.
- Beta product features are expected to eventually become product features in a future release of Arm Compiler 6.
- Arm encourages the use of beta product features, and welcomes feedback on them.
- Any issues with beta product features that Arm encounters or is made aware of are considered for fixing in future versions of Arm Compiler.

Alpha product features

Alpha product features are not implementation complete, and are subject to change in future releases, therefore the stability level is lower than in beta product features.
Alpha product features are indicated with [ALPHA].

- Arm endeavors to document known limitations of alpha product features.
- Arm encourages the use of alpha product features, and welcomes feedback on them.
- Any issues with alpha product features that Arm encounters or is made aware of are considered for fixing in future versions of Arm Compiler.

Community features

Arm Compiler 6 is built on LLVM technology and preserves the functionality of that technology where possible. This means that there are additional features available in Arm Compiler that are not listed in the documentation. These additional features are known as community features. For information on these community features, see the *documentation for the Clang/LLVM project*.

Where community features are referenced in the documentation, they are indicated with [COMMUNITY].

- Arm makes no claims about the quality level or the degree of functionality of these features, except when explicitly stated in this documentation.
- Functionality might change significantly between feature releases.
- Arm makes no guarantees that community features will remain functional across update releases, although changes are expected to be unlikely.

Some community features might become product features in the future, but Arm provides no roadmap for this. Arm is interested in understanding your use of these features, and welcomes feedback on them. Arm supports customers using these features on a best-effort basis, unless the features are unsupported. Arm accepts defect reports on these features, but does not guarantee that these issues will be fixed in future releases.

Guidance on use of community features

There are several factors to consider when assessing the likelihood of a community feature being functional:

- The following figure shows the structure of the Arm Compiler 6 toolchain:

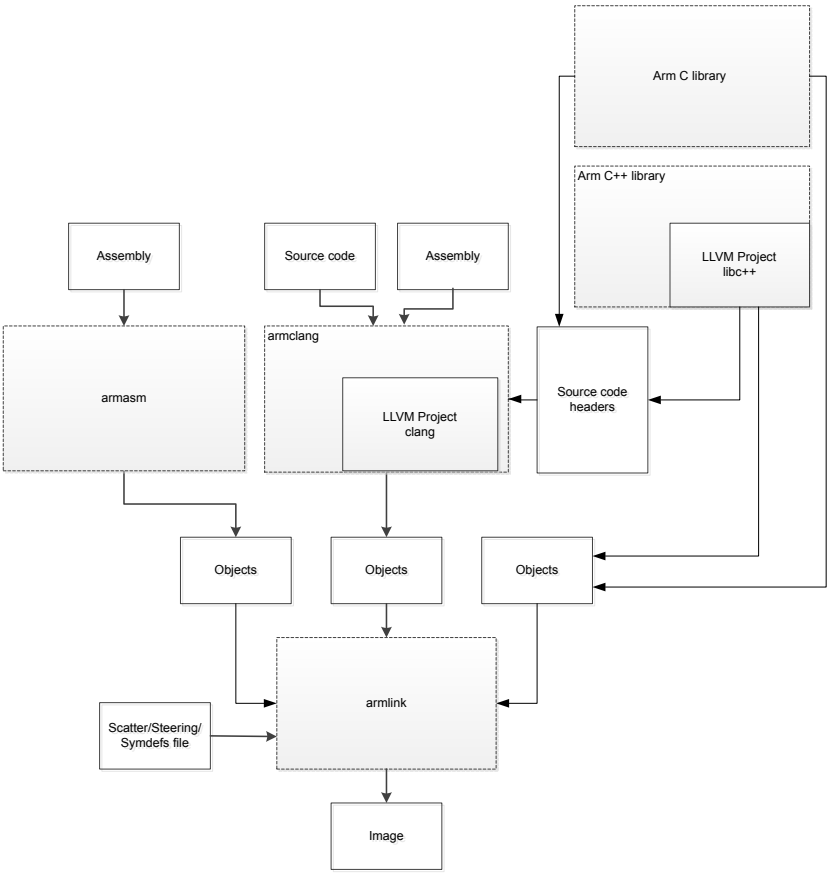


Figure 1-1 Integration boundaries in Arm Compiler 6.

The dashed boxes are toolchain components, and any interaction between these components is an integration boundary. Community features that span an integration boundary might have significant limitations in functionality. The exception to this is if the interaction is codified in one of the standards supported by Arm Compiler 6. See *Application Binary Interface (ABI) for the Arm®*

Architecture. Community features that do not span integration boundaries are more likely to work as expected.
- Features primarily used when targeting hosted environments such as Linux or BSD might have significant limitations, or might not be applicable, when targeting bare-metal environments.
- The Clang implementations of compiler features, particularly those that have been present for a long time in other toolchains, are likely to be mature. The functionality of new features, such as support for new language features, is likely to be less mature and therefore more likely to have limited functionality.

Unsupported features

With both the product and community feature categories, specific features and use-cases are known not to function correctly, or are not intended for use with Arm Compiler 6.

Limitations of product features are stated in the documentation. Arm cannot provide an exhaustive list of unsupported features or use-cases for community features. The known limitations on community features are listed in *Community features* on page 1-53.

List of known unsupported features

The following is an incomplete list of unsupported features, and might change over time:
- The Clang option `-stdlib=libstdc++` is not supported.
- C++ static initialization of local variables is not thread-safe when linked against the standard C++ libraries. For thread-safety, you must provide your own implementation of thread-safe functions as described in *Standard C++ library implementation definition*.

  Note

  This restriction does not apply to the [ALPHA]-supported multi-threaded C++ libraries.

- Use of C11 library features is unsupported.
- Any community feature that exclusively pertains to non-Arm architectures is not supported.
- Compilation for targets that implement architectures older than Armv7 or Armv6-M is not supported.
- The **`long double`** data type is not supported for AArch64 state because of limitations in the current Arm C library.
- Complex numbers are not supported because of limitations in the current Arm C library.

Chapter 2
Overview of the Arm®v8 Architecture

Gives an overview of the Armv8 architecture.

It contains the following sections:

2.1 About the Arm® architecture

The Arm architecture is a load-store architecture. The addressing range depends on whether you are using the 32-bit or the 64-bit architecture.

Arm processors are typical of RISC processors in that only load and store instructions can access memory. Data processing instructions operate on register contents only.

Armv8 is the next major architectural update after Armv7. It introduces a 64-bit architecture, but maintains compatibility with existing 32-bit architectures. It uses two execution states:

AArch32
In AArch32 state, code has access to 32-bit general purpose registers.

Code executing in AArch32 state can only use the A32 and T32 instruction sets. This state is broadly compatible with the Armv7-A architecture.

AArch64
In AArch64 state, code has access to 64-bit general purpose registers. The AArch64 state exists only in the Armv8 architecture.

Code executing in AArch64 state can only use the A64 instruction set.

In the AArch32 execution state, there are the following instruction set states:

A32 state
The state that executes A32 instructions.

T32 state
The state that executes T32 instructions.

Related information

Arm Architecture Reference Manual.

2.2 A32 and T32 instruction sets

A32 instructions are 32 bits wide. T32 instructions are 32-bits wide with 16-bit instructions in some architectures.

The A32 instruction set provides a comprehensive range of operations.

Most of the functionality of the 32-bit A32 instruction set is available, but some operations require more instructions. The T32 instruction set provides better code density, at the expense of performance.

The 32-bit and 16-bit T32 instructions together provide almost exactly the same functionality as the A32 instruction set. The T32 instruction set achieves the high performance of A32 code along with the benefits of better code density.

Armv6-M, Armv7-M, Armv8-M.baseline, and Armv8-M.mainline do not support the A32 instruction set. On these architectures, instructions must not attempt to change to A32 state. Armv7-A, Armv7-R, Armv8-A, and Armv8-R support both A32 and T32 instruction sets.

Note

With the exception of Armv6-M and Armv6S-M, assembling code for architectures earlier than Armv7 is not supported in Arm Compiler 6.

In Armv8, the A32 and T32 instruction sets are largely unchanged from Armv7. They are only available when the processor is in AArch32 state. The main changes in Armv8 are the addition of a few new instructions and the deprecation of some behavior, including many uses of the `IT` instruction.

Armv8 also defines an optional Crypto Extension. This extension provides cryptographic and hash instructions in the A32 instruction set.

Note

- The term A32 is an alias for the Arm instruction set.
- The term T32 is an alias for the Thumb® instruction set.

Related references

3.14 A32 and T32 instruction set overview on page 3-78.

2.3 A64 instruction set

A64 instructions are 32 bits wide.

Armv8 introduces a new set of 32-bit instructions called A64, with new encodings and assembly language. A64 is only available when the processor is in AArch64 state. It provides similar functionality to the A32 and T32 instruction sets, but gives access to a larger virtual address space, and has some other changes, including reduced conditionality.

Armv8 also defines an optional Crypto Extension. This extension provides cryptographic and hash instructions in the A64 instruction set.

Related references

4.12 A64 instruction set overview on page 4-92.

2.4 Changing between AArch64 and AArch32 states

The processor must be in the correct execution state for the instructions it is executing.

A processor that is executing A64 instructions is operating in AArch64 state. In this state, the instructions can access both the 64-bit and 32-bit registers.

A processor that is executing A32 or T32 instructions is operating in AArch32 state. In this state, the instructions can only access the 32-bit registers, and not the 64-bit registers.

A processor based on the Armv8 architecture can run applications built for AArch32 and AArch64 states but a change between AArch32 and AArch64 states can only happen at exception boundaries.

Arm Compiler toolchain builds images for either the AArch32 state or AArch64 state. Therefore, an image built with Arm Compiler toolchain can either contain only A32 and T32 instructions or only A64 instructions.

A processor can only execute instructions from the instruction set that matches its current execution state. A processor in AArch32 state cannot execute A64 instructions, and a processor in AArch64 state cannot execute A32 or T32 instructions. You must ensure that the processor never receives instructions from the wrong instruction set for the current execution state.

Related references

13.21 BLX, BLXNS on page 13-368.
13.22 BX, BXNS on page 13-370.
21.7 ARM or CODE32 directive on page 21-1653.
21.11 CODE16 directive on page 21-1657.
21.65 THUMB directive on page 21-1719.

2.5 Advanced SIMD

Advanced SIMD is a 64-bit and 128-bit hybrid *Single Instruction Multiple Data* (SIMD) technology targeted at advanced media and signal processing applications and embedded processors.

Advanced SIMD is implemented as part of an Arm-based processor, but has its own execution pipelines and a register bank that is distinct from the general-purpose register bank.

Advanced SIMD instructions are available in both A32 and A64. The A64 Advanced SIMD instructions are based on those in A32. The main differences are the following:

- Different instruction mnemonics and syntax.
- Thirty-two 128-bit vector registers, increased from sixteen in A32.
- A different register packing scheme:
  - In A64, smaller registers occupy the low order bits of larger registers. For example, S31 maps to bits[31:0] of D31.
  - In A32, smaller registers are packed into larger registers. For example, S31 maps to bits[63:32] of D15.
- A64 Advanced SIMD instructions support both single-precision and double-precision floating-point data types and arithmetic.
- A32 Advanced SIMD instructions support only single-precision floating-point data types.

Related concepts

9.1 Architecture support for Advanced SIMD on page 9-182.
9.4 Views of the Advanced SIMD register bank in AArch32 state on page 9-187.
9.5 Views of the Advanced SIMD register bank in AArch64 state on page 9-188.

Related references

Chapter 9 Advanced SIMD Programming on page 9-181.

2.6 Floating-point hardware

There are several floating-point architecture versions and variants.

The floating-point hardware, together with associated support code, provides single-precision and double-precision floating-point arithmetic, as defined by *IEEE Std. 754-2008 IEEE Standard for Floating-Point Arithmetic*. This document is referred to as the IEEE 754 standard.

The floating-point hardware uses a register bank that is distinct from the Arm core register bank.

Note

The floating-point register bank is shared with the SIMD register bank.

In AArch32 state, floating-point support is largely unchanged from VFPv4, apart from the addition of a few instructions for compliance with the IEEE 754 standard.

The floating-point architecture in AArch64 state is also based on VFPv4. The main differences are the following:

- In AArch64 state, the number of 128-bit SIMD and floating-point registers increases from sixteen to thirty-two.
- Single-precision registers are no longer packed into double-precision registers, so register S*x* is D*x*[31:0].
- The presence of floating-point hardware is mandated, so software floating-point linkage is not supported.
- Earlier versions of the floating-point architecture, for instance VFPv2, VFPv3, and VFPv4, are not supported in AArch64 state.
- VFP vector mode is not supported in either AArch32 or AArch64 state. Use Advanced SIMD instructions for vector floating-point.
- Some new instructions have been added, including:
  - Direct conversion between half-precision and double-precision.
  - Load and store pair, replacing load and store multiple.
  - Fused multiply-add and multiply-subtract.
  - Instructions for IEEE 754-2008 compatibility.

Related concepts

10.1 Architecture support for floating-point on page 10-207.
10.4 Views of the floating-point extension register bank in AArch32 state on page 10-211.
10.5 Views of the floating-point extension register bank in AArch64 state on page 10-212.

Related references

Chapter 10 Floating-point Programming on page 10-206.

Chapter 3
Overview of AArch32 state

Gives an overview of the AArch32 state of Armv8.

It contains the following sections:

3.1 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

13.21 BLX, BLXNS on page 13-368.
13.22 BX, BXNS on page 13-370.
21.7 ARM or CODE32 directive on page 21-1653.
21.11 CODE16 directive on page 21-1657.
21.65 THUMB directive on page 21-1719.

3.2 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 3-1 Arm 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

3.3 Processor modes in Arm®v6-M, Arm®v7-M, and Arm®v8-M on page 3-66.

Related information

Arm Architecture Reference Manual.

3.3 Processor modes in Arm®v6-M, Arm®v7-M, and Arm®v8-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

3.2 Processor modes, and privileged and unprivileged software execution on page 3-65.

Related information

Arm Architecture Reference Manual.

3.4 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.

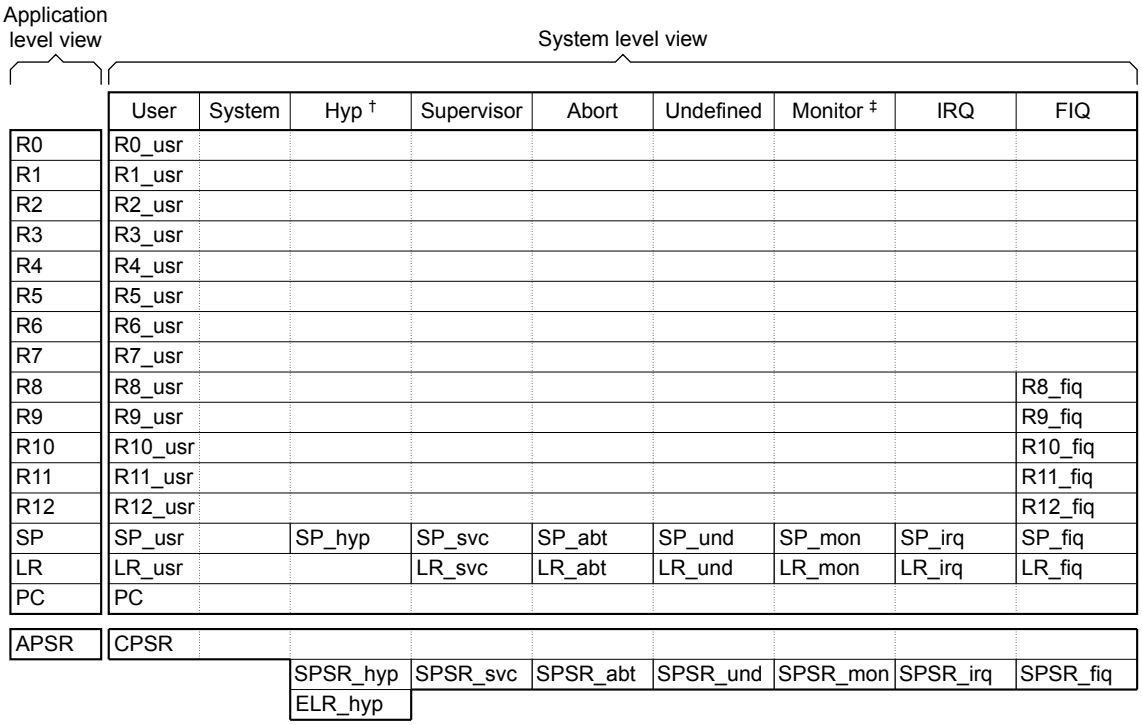
The additional registers in Arm processors are:

- 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 3-1 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

3.5 General-purpose registers in AArch32 state on page 3-69.

3.9 Program Counter in AArch32 state on page 3-73.

3.11 Application Program Status Register on page 3-75.

3.13 Saved Program Status Registers in AArch32 state on page 3-77.

3.12 Current Program Status Register in AArch32 state on page 3-76.

3.2 Processor modes, and privileged and unprivileged software execution on page 3-65.

Related information

Arm Architecture Reference Manual.

3.5 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 13 A32 and T32 Instructions* on page 13-327 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

3.9 Program Counter in AArch32 state on page 3-73.

3.6 Register accesses in AArch32 state on page 3-70.

Related references

3.7 Predeclared core register names in AArch32 state on page 3-71.

13.68 MRS (PSR to general-purpose register) on page 13-437.

13.71 MSR (general-purpose register to PSR) on page 13-441.

3.6 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

3.5 General-purpose registers in AArch32 state on page 3-69.

3.9 Program Counter in AArch32 state on page 3-73.

3.11 Application Program Status Register on page 3-75.

3.12 Current Program Status Register in AArch32 state on page 3-76.

3.13 Saved Program Status Registers in AArch32 state on page 3-77.

6.20 The Read-Modify-Write operation on page 6-128.

Related references

3.7 Predeclared core register names in AArch32 state on page 3-71.

13.68 MRS (PSR to general-purpose register) on page 13-437.

13.71 MSR (general-purpose register to PSR) on page 13-441.

3.7 Predeclared core register names in AArch32 state

Many of the core register names have synonyms.

The following table shows the predeclared core registers:

Table 3-2 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

3.5 General-purpose registers in AArch32 state on page 3-69.

3.8 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 3-3 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.

Related concepts

9.2 Extension register bank mapping for Advanced SIMD in AArch32 state on page 9-183.

3.9 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 concepts

12.5 Register-relative and PC-relative expressions on page 12-302.

Related references

13.15 B on page 13-359.
13.22 BX, BXNS on page 13-370.
13.24 CBZ and CBNZ on page 13-373.
13.161 TBB and TBH on page 13-554.

3.10 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 concepts

6.20 The Read-Modify-Write operation on page 6-128.

Related references

13.68 MRS (PSR to general-purpose register) on page 13-437.
13.71 MSR (general-purpose register to PSR) on page 13-441.
13.82 QADD on page 13-457.
13.132 SMULxy on page 13-514.
13.134 SMULWy on page 13-516.

3.11 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

7.1 Conditional instructions on page 7-140.

Related references

7.6 Updates to the condition flags in A32/T32 code on page 7-145.

13.68 MRS (PSR to general-purpose register) on page 13-437.

13.71 MSR (general-purpose register to PSR) on page 13-441.

13.107 SEL on page 13-487.

3.12 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.
- Either:
  - The instruction set state for the Armv8 architecture (A32 or T32).
  - The instruction set state for the Armv7 architecture (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 in A32 and T32 and has no equivalent in A64.

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

3.13 Saved Program Status Registers in AArch32 state on page 3-77.

Related references

13.45 IT on page 13-399.

13.68 MRS (PSR to general-purpose register) on page 13-437.

13.71 MSR (general-purpose register to PSR) on page 13-441.

13.108 SETEND on page 13-489.

7.6 Updates to the condition flags in A32/T32 code on page 7-145.

3.13 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

3.12 Current Program Status Register in AArch32 state on page 3-76.

3.14 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 3-4 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.

Related concepts

6.14 Load and store multiple register instructions on page 6-120.

3.15 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.

Related concepts

6.4 Load immediate values on page 6-105.

6.5 Load immediate values using MOV and MVN on page 6-106.

Chapter 4
Overview of AArch64 state

Gives an overview of the AArch64 state of Armv8.

It contains the following sections:

4.1 Registers in AArch64 state

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

In AArch64 state, the following registers are available:

- Thirty-one 64-bit general-purpose registers X0-X30, the bottom halves of which are accessible as W0-W30.
- Four stack pointer registers SP_EL0, SP_EL1, SP_EL2, SP_EL3.
- Three exception link registers ELR_EL1, ELR_EL2, ELR_EL3.
- Three saved program status registers SPSR_EL1, SPSR_EL2, SPSR_EL3.
- One program counter.

All these registers are 64 bits wide except SPSR_EL1, SPSR_EL2, and SPSR_EL3, which are 32 bits wide.

Most A64 integer instructions can operate on either 32-bit or 64-bit registers. The register width is determined by the register identifier, where W means 32-bit and X means 64-bit. The names W*n* and X*n*, where *n* is in the range 0-30, refer to the same register. When you use the 32-bit form of an instruction, the upper 32 bits of the source registers are ignored and the upper 32 bits of the destination register are set to zero.

There is no register named W31 or X31. Depending on the instruction, register 31 is either the stack pointer or the zero register. When used as the stack pointer, you refer to it as SP. When used as the zero register, you refer to it as WZR in a 32-bit context or XZR in a 64-bit context.

Related concepts

4.2 Exception levels on page 4-82.
4.3 Link registers on page 4-83.
4.4 Stack Pointer register on page 4-84.
4.7 Program Counter in AArch64 state on page 4-87.
4.8 Conditional execution in AArch64 state on page 4-88.
4.11 Saved Program Status Registers in AArch64 state on page 4-91.

4.2 Exception levels

The Armv8 architecture defines four exception levels, EL0 to EL3, where EL3 is the highest exception level with the most execution privilege. When taking an exception, the exception level can either increase or remain the same, and when returning from an exception, it can either decrease or remain the same.

The following is a common usage model for the exception levels:

EL0
: Applications.

EL1
: OS kernels and associated functions that are typically described as privileged.

EL2
: Hypervisor.

EL3
: Secure monitor.

When taking an exception to a higher exception level, the execution state can either remain the same, or change from AArch32 to AArch64.

When returning to a lower exception level, the execution state can either remain the same or change from AArch64 to AArch32.

The only way the execution state can change is by taking or returning from an exception. It is not possible to change between execution states in the same way as changing between A32 and T32 code in AArch32 state.

On powerup and on reset, the processor enters the highest implemented exception level. The execution state for this exception level is a property of the implementation, and might be determined by a configuration input signal.

For exception levels other than EL0, the execution state is determined by one or more control register configuration bits. These bits can be set only in a higher exception level.

For EL0, the execution state is determined as part of the exception return to EL0, under the control of the exception level that the execution is returning from.

Related concepts

4.3 Link registers on page 4-83.
4.11 Saved Program Status Registers in AArch64 state on page 4-91.
2.4 Changing between AArch64 and AArch32 states on page 2-60.
4.10 Process State on page 4-90.

4.3 Link registers

In AArch64 state, the Link Register (LR) stores 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. The LR maps to register 30. Unlike in AArch32 state, the LR is distinct from the Exception Link Registers (ELRs) and is therefore unbanked.

There are three Exception Link Registers, ELR_EL1, ELR_EL2, and ELR_EL3, that correspond to each of the exception levels. When an exception is taken, the Exception Link Register for the target exception level stores the return address to jump to after the handling of that exception completes. If the exception was taken from AArch32 state, the top 32 bits in the ELR are all set to zero. Subroutine calls within the exception level use the LR to store the return address from the subroutine.

For example when the exception level changes from EL0 to EL1, the return address is stored in ELR_EL1.

When in an exception level, if you enable interrupts that use the same exception level, you must ensure you store the ELR on the stack because it will be overwritten with a new return address when the interrupt is taken.

Related concepts

4.7 Program Counter in AArch64 state on page 4-87.

3.5 General-purpose registers in AArch32 state on page 3-69.

Related references

4.5 Predeclared core register names in AArch64 state on page 4-85.

4.4 Stack Pointer register

In AArch64 state, SP represents the 64-bit Stack Pointer. SP_EL0 is an alias for SP. Do not use SP as a general purpose register.

You can only use SP as an operand in the following instructions:

- As the base register for loads and stores. In this case it must be quadword-aligned before adding any offset, or a stack alignment exception occurs.
- As a source or destination for arithmetic instructions, but it cannot be used as the destination in instructions that set the condition flags.
- In logical instructions, for example in order to align it.

There is a separate stack pointer for each of the three exception levels, SP_EL1, SP_EL2, and SP_EL3. Within an exception level you can either use the dedicated stack pointer for that exception level or you can use SP_EL0, the stack pointer associated with EL0. You can use the SPSel register to select which stack pointer to use in the exception level.

The choice of stack pointer is indicated by the letter t or h appended to the exception level name, for example EL0t or EL3h. The t suffix indicates that the exception level uses SP_EL0 and the h suffix indicates it uses SP_EL*x*, where *x* is the current exception level number. EL0 always uses SP_EL0 so cannot have an h suffix.

Related concepts

3.5 General-purpose registers in AArch32 state on page 3-69.

4.2 Exception levels on page 4-82.

4.10 Process State on page 4-90.

Related references

4.1 Registers in AArch64 state on page 4-81.

4.5 Predeclared core register names in AArch64 state

In AArch64 state, the predeclared core registers are different from those in AArch32 state.

The following table shows the predeclared core registers in AArch64 state:

Table 4-1 Predeclared core registers in AArch64 state

Register names	Meaning
`W0-W30`	32-bit general purpose registers.
`X0-X30`	64-bit general purpose registers.
`WZR`	32-bit RAZ/WI register. This is the name for register 31 when it is used as the zero register in a 32-bit context.
`XZR`	64-bit RAZ/WI register. This is the name for register 31 when it is used as the zero register in a 64-bit context.
`WSP`	32-bit stack pointer. This is the name for register 31 when it is used as the stack pointer in a 32-bit context.
`SP`	64-bit stack pointer. This is the name for register 31 when it is used as the stack pointer in a 64-bit context.
`LR`	Link register. This is a synonym for X30.

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

Note

In AArch64 state, the PC is not a general purpose register and you cannot access it by name.

Related concepts

4.3 Link registers on page 4-83.
4.4 Stack Pointer register on page 4-84.
4.7 Program Counter in AArch64 state on page 4-87.

Related references

3.7 Predeclared core register names in AArch32 state on page 3-71.
4.1 Registers in AArch64 state on page 4-81.

4.6 Predeclared extension register names in AArch64 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 in AArch64 state:

Table 4-2 Predeclared extension registers in AArch64 state

Register names	Meaning
`V0-V31`	Advanced SIMD 128-bit vector registers.
`Q0-Q31`	Advanced SIMD registers holding a 128-bit scalar.
`D0-D31`	Advanced SIMD registers holding a 64-bit scalar, floating-point double-precision registers.
`S0-S31`	Advanced SIMD registers holding a 32-bit scalar, floating-point single-precision registers.
`H0-H31`	Advanced SIMD registers holding a 16-bit scalar, floating-point half-precision registers.
`B0-B31`	Advanced SIMD registers holding an 8-bit scalar.

Related concepts

9.3 Extension register bank mapping for Advanced SIMD in AArch64 state on page 9-185.

Related references

3.8 Predeclared extension register names in AArch32 state on page 3-72.

4.1 Registers in AArch64 state on page 4-81.

4.7 Program Counter in AArch64 state

In AArch64 state, the *Program Counter* (PC) contains the address of the currently executing instruction. It is incremented by the size of the instruction executed, which is always four bytes.

In AArch64 state, the PC is not a general purpose register and you cannot access it explicitly. The following types of instructions read it implicitly:

- Instructions that compute a PC-relative address.
- PC-relative literal loads.
- Direct branches to a PC-relative label.
- Branch and link instructions, which store it in the procedure link register.

The only types of instructions that can write to the PC are:

- Conditional and unconditional branches.
- Exception generation and exception returns.

Branch instructions load the destination address into the PC.

Related concepts

3.9 Program Counter in AArch32 state on page 3-73.

12.5 Register-relative and PC-relative expressions on page 12-302.

Related references

13.15 B on page 13-359.

13.20 BL on page 13-366.

13.21 BLX, BLXNS on page 13-368.

13.22 BX, BXNS on page 13-370.

4.8 Conditional execution in AArch64 state

In AArch64 state, the NZCV register 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 NZCV register contains the flags in bits[31:28].

The condition flags are accessible in all exception levels, using the `MSR` and `MRS` instructions.

A64 makes less use of conditionality than A32. For example, in A64:

- Only a few instructions can set or test the condition flags.
- There is no equivalent of the T32 `IT` instruction.
- The only conditionally executed instruction, which behaves as a NOP if the condition is false, is the conditional branch, `B.`*`cond`*.

Related concepts

3.11 Application Program Status Register on page 3-75.

7.1 Conditional instructions on page 7-140.

Related references

7.6 Updates to the condition flags in A32/T32 code on page 7-145.

7.7 Updates to the condition flags in A64 code on page 7-146.

13.68 MRS (PSR to general-purpose register) on page 13-437.

13.71 MSR (general-purpose register to PSR) on page 13-441.

4.9 The Q flag in AArch64 state

In AArch64 state, you cannot read or write to the Q flag because in A64 there are no saturating arithmetic instructions that operate on the general purpose registers.

The Advanced SIMD saturating arithmetic instructions set the QC bit in the floating-point status register (FPSR) to indicate that saturation has occurred. You can identify such instructions by the Q mnemonic modifier, for example `SQADD`.

Related references

Chapter 19 A64 SIMD Scalar Instructions on page 19-1212.
Chapter 20 A64 SIMD Vector Instructions on page 20-1335.

4.10 Process State

In AArch64 state, there is no *Current Program Status Register* (CPSR). You can access the different components of the traditional CPSR independently as *Process State* fields.

The Process State fields are:

- N, Z, C, and V condition flags (NZCV).
- Current register width (nRW).
- Stack pointer selection bit (SPSel).
- Interrupt disable flags (DAIF).
- Current exception level (EL).
- Single step process state bit (SS).
- Illegal exception return state bit (IL).

You can use `MSR` to write to:

- The N, Z, C, and V flags in the NZCV register.
- The interrupt disable flags in the DAIF register.
- The SP selection bit in the SPSel register, in EL1 or higher.

You can use `MRS` to read:

- The N, Z, C, and V flags in the NZCV register.
- The interrupt disable flags in the DAIF register.
- The exception level bits in the CurrentEL register, in EL1 or higher.
- The SP selection bit in the SPSel register, in EL1 or higher.

When an exception occurs, all Process State fields associated with the current exception level are stored in a single register associated with the target exception level, the SPSR. You can access the SS, IL, and nRW bits only from the SPSR.

Related concepts

3.12 Current Program Status Register in AArch32 state on page 3-76.
3.13 Saved Program Status Registers in AArch32 state on page 3-77.
4.11 Saved Program Status Registers in AArch64 state on page 4-91.

Related references

7.6 Updates to the condition flags in A32/T32 code on page 7-145.
7.7 Updates to the condition flags in A64 code on page 7-146.
13.68 MRS (PSR to general-purpose register) on page 13-437.
13.71 MSR (general-purpose register to PSR) on page 13-441.

4.11 Saved Program Status Registers in AArch64 state

The *Saved Program Status Registers* (SPSRs) are 32-bit registers that store the process state of the current exception level when an exception is taken to an exception level that uses AArch64 state. This allows the process state to be restored after the exception has been handled.

In AArch64 state, each target exception level has its own SPSR:

- SPSR_EL1.
- SPSR_EL2.
- SPSR_EL3.

When taking an exception, the process state of the current exception level is stored in the SPSR of the target exception level. On returning from an exception, the exception handler uses the SPSR of the exception level that is being returned from to restore the process state of the exception level that is being returned to.

Note

On returning from an exception, the preferred return address is restored from the ELR associated with the exception level that is being returned from.

The SPSRs store the following information:

- N, Z, C, and V flags.
- D, A, I, and F interrupt disable bits.
- The register width.
- The execution mode.
- The IL and SS bits.

Related concepts

4.4 Stack Pointer register on page 4-84.

4.10 Process State on page 4-90.

3.13 Saved Program Status Registers in AArch32 state on page 3-77.

4.12 A64 instruction set overview

A64 instructions can be grouped by functional area.

The following table describes some of the functional groupings of the instructions in A64.

Table 4-3 A64 instruction groups

Instruction group	Description
Branch and control	These instructions do the following: • Branch to and return from subroutines. • Branch backwards to form loops. • Branch forward in conditional structures. • Generate and return from exceptions.
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. The addition and subtraction instructions can optionally left shift the immediate operand, or can sign or zero-extend and shift the final source operand register. A64 includes signed and unsigned 32-bit and 64-bit multiply and divide instructions.
Register load and store	These instructions load or store the value of a single register or pair of registers from or to memory. You can load or store a single 64-bit doubleword, 32-bit word, 16-bit halfword, or 8-bit byte, or a pair of words or doublewords. Byte and halfword loads can either be sign-extended or zero-extended to fill the 32-bit register. You can also load and sign-extend a signed byte, halfword or word into a 64-bit register, or load a pair of signed words into two 64-bit registers.
System register access	These instructions move the contents of a system register to or from a general-purpose register.

Related references

3.14 A32 and T32 instruction set overview on page 3-78.
Chapter 16 A64 General Instructions on page 16-794.
Chapter 17 A64 Data Transfer Instructions on page 17-990.

Chapter 5
Structure of Assembly Language Modules

Describes the structure of assembly language source files.

It contains the following sections:

5.1 Syntax of source lines in assembly language

The assembler parses and assembles assembly language to produce object code.

Syntax

Each line of assembly language source code has this general form:

`{symbol} {instruction|directive|pseudo-instruction} {;comment}`

All three sections of the source line are optional.

symbol is usually a label. In instructions and pseudo-instructions it is always a label. In some directives it is a symbol for a variable or a constant. The description of the directive makes this clear in each case.

symbol must begin in the first column. It cannot contain any white space character such as a space or a tab unless it is enclosed by bars (|).

Labels are symbolic representations of addresses. You can use labels to mark specific addresses that you want to refer to from other parts of the code. Numeric local labels are a subclass of labels that begin with a number in the range 0-99. Unlike other labels, a numeric local label can be defined many times. This makes them useful when generating labels with a macro.

Directives provide important information to the assembler that either affects the assembly process or affects the final output image.

Instructions and pseudo-instructions make up the code a processor uses to perform tasks.

Note

Instructions, pseudo-instructions, and directives must be preceded by white space, such as a space or a tab, irrespective of whether there is a preceding label or not.

Some directives do not allow the use of a label.

A comment is the final part of a source line. The first semicolon on a line marks the beginning of a comment except where the semicolon appears inside a string literal. The end of the line is the end of the comment. A comment alone is a valid line. The assembler ignores all comments. You can use blank lines to make your code more readable.

Considerations when writing assembly language source code

You must write instruction mnemonics, pseudo-instructions, directives, and symbolic register names (except `a1-a4` and `v1-v8` in A32 or T32 instructions) in either all uppercase or all lowercase. You must not use mixed case. Labels and comments can be in uppercase, lowercase, or mixed case.

```
        AREA     A32ex, CODE, READONLY
        ; Name this block of code A32ex

        ENTRY                   ; Mark first instruction to execute
start
        MOV      r0, #10        ; Set up parameters
        MOV      r1, #3
        ADD      r0, r0, r1     ; r0 = r0 + r1
stop
        MOV      r0, #0x18      ; angel_SWIreason_ReportException
        LDR      r1, =0x20026   ; ADP_Stopped_ApplicationExit
        SVC      #0x123456      ; A32 semihosting (formerly SWI)
        END                     ; Mark end of file
```

To make source files easier to read, you can split a long line of source into several lines by placing a backslash character (\) at the end of the line. The backslash must not be followed by any other

characters, including spaces and tabs. The assembler treats the backslash followed by end-of-line sequence as white space. You can also use blank lines to make your code more readable.

Note

Do not use the backslash followed by end-of-line sequence within quoted strings.

The limit on the length of lines, including any extensions using backslashes, is 4095 characters.

Related concepts

12.6 Labels on page 12-303.
12.10 Numeric local labels on page 12-307.
12.13 String literals on page 12-310.

Related references

5.2 Literals on page 5-96.
12.1 Symbol naming rules on page 12-298.
12.15 Syntax of numeric literals on page 12-312.

5.2 Literals

Assembly language source code can contain numeric, string, Boolean, and single character literals.

Literals can be expressed as:

- Decimal numbers, for example `123`.
- Hexadecimal numbers, for example `0x7B`.
- Numbers in any base from `2` to `9`, for example `5_204` is a number in base `5`.
- Floating point numbers, for example `123.4`.
- Boolean values `{TRUE}` or `{FALSE}`.
- Single character values enclosed by single quotes, for example `'w'`.
- Strings enclosed in double quotes, for example `"This is a string"`.

Note

In most cases, a string containing a single character is accepted as a single character value. For example `ADD r0,r1,#"a"` is accepted, but `ADD r0,r1,#"ab"` is faulted.

You can also use variables and names to represent literals.

Related references

5.1 Syntax of source lines in assembly language on page 5-94.

5.3 ELF sections and the AREA directive

Object files produced by the assembler are divided into sections. In assembly source code, you use the `AREA` directive to mark the start of a section.

ELF sections are independent, named, indivisible sequences of code or data. A single code section is the minimum required to produce an application.

The output of an assembly or compilation can include:

- One or more code sections. These are usually read-only sections.
- One or more data sections. These are usually read-write sections. They might be *zero-initialized* (ZI).

The linker places each section in a program image according to section placement rules. Sections that are adjacent in source files are not necessarily adjacent in the application image

Use the `AREA` directive to name the section and set its attributes. The attributes are placed after the name, separated by commas.

You can choose any name for your sections. However, names starting with any non-alphabetic character must be enclosed in bars, or an `AREA name missing` error is generated. For example, `|1_DataArea|`.

The following example defines a single read-only section called `A32ex` that contains code:

```
        AREA  A32ex, CODE, READONLY ; Name this block of code A32ex
```

Related concepts

5.4 An example armasm syntax assembly language module on page 5-98.

Related references

21.6 AREA on page 21-1650.

Related information

Information about scatter files.

5.4 An example armasm syntax assembly language module

An armasm syntax assembly language module has several constituent parts.

These are:

- ELF sections (defined by the AREA directive).
- Application entry (defined by the ENTRY directive).
- Application execution.
- Application termination.
- Program end (defined by the END directive).

Constituents of an A32 assembly language module

The following example defines a single section called A32ex that contains code and is marked as being READONLY. This example uses the A32 instruction set.

```
        AREA     A32ex, CODE, READONLY
                                ; Name this block of code A32ex
        ENTRY                   ; Mark first instruction to execute
start
        MOV      r0, #10        ; Set up parameters
        MOV      r1, #3
        ADD      r0, r0, r1     ; r0 = r0 + r1
stop
        MOV      r0, #0x18      ; angel_SWIreason_ReportException
        LDR      r1, =0x20026   ; ADP_Stopped_ApplicationExit
        SVC      #0x123456      ; A32 semihosting (formerly SWI)
        END                     ; Mark end of file
```

Constituents of an A64 assembly language module

The following example defines a single section called A64ex that contains code and is marked as being READONLY. This example uses the A64 instruction set.

```
        AREA     A64ex, CODE, READONLY
                                ; Name this block of code A64ex
        ENTRY                   ; Mark first instruction to execute
start
        MOV      w0, #10        ; Set up parameters
        MOV      w1, #3
        ADD      w0, w0, w1     ; w0 = w0 + w1
stop
        MOV      x1, #0x26
        MOVK     x1, #2, LSL #16
        STR      x1, [sp,#0]    ; ADP_Stopped_ApplicationExit
        MOV      x0, #0
        STR      x0, [sp,#8]    ; Exit status code
        MOV      x1, sp         ; x1 contains the address of parameter block
        MOV      w0, #0x18      ; angel_SWIreason_ReportException
        HLT      0xf000         ; A64 semihosting
        END                     ; Mark end of file
```

Constituents of a T32 assembly language module

The following example defines a single section called T32ex that contains code and is marked as being READONLY. This example uses the T32 instruction set.

```
        AREA     T32ex, CODE, READONLY
                                ; Name this block of code T32ex
        ENTRY                   ; Mark first instruction to execute
        THUMB
start
        MOV      r0, #10        ; Set up parameters
        MOV      r1, #3
        ADD      r0, r0, r1     ; r0 = r0 + r1
stop
        MOV      r0, #0x18      ; angel_SWIreason_ReportException
        LDR      r1, =0x20026   ; ADP_Stopped_ApplicationExit
        SVC      #0xab          ; T32 semihosting (formerly SWI)
        ALIGN    4              ; Aligned on 4-byte boundary
        END                     ; Mark end of file
```

Application entry

The `ENTRY` directive declares an entry point to the program. It marks the first instruction to be executed. In applications using the C library, an entry point is also contained within the C library initialization code. Initialization code and exception handlers also contain entry points.

Application execution in A32 or T32 code

The application code begins executing at the label `start`, where it loads the decimal values 10 and 3 into registers `R0` and `R1`. These registers are added together and the result placed in `R0`.

Application execution in A64 code

The application code begins executing at the label `start`, where it loads the decimal values 10 and 3 into registers `W0` and `W1`. These registers are added together and the result placed in `W0`.

Application termination

After executing the main code, the application terminates by returning control to the debugger.

A32 and T32 code

You do this in A32 and T32 code using the semihosting `SVC` instruction:

- In A32 code, the semihosting `SVC` instruction is `0x123456` by default.
- In T32 code, use the semihosting `SVC` instruction is `0xAB` by default.

A32 and T32 code uses the following parameters:

- R0 equal to `angel_SWIreason_ReportException` (`0x18`).
- R1 equal to `ADP_Stopped_ApplicationExit` (`0x20026`).

A64 code

In A64 code, use `HLT` instruction `0xF000` to invoke the semihosting interface.
A64 code uses the following parameters:

- W0 equal to `angel_SWIreason_ReportException` (`0x18`).
- X1 is the address of a block of two parameters. The first is the exception type, `ADP_Stopped_ApplicationExit` (`0x20026`) and the second is the exit status code.

Program end

The `END` directive instructs the assembler to stop processing this source file. Every assembly language source module must finish with an `END` directive on a line by itself. Any lines following the `END` directive are ignored by the assembler.

Related concepts

5.3 ELF sections and the AREA directive on page 5-97.

Related references

21.23 END on page 21-1669.
21.25 ENTRY on page 21-1671.

Related information

Semihosting for AArch32 and AArch64.

Chapter 6
Writing A32/T32 Assembly Language

Describes the use of a few basic A32 and T32 instructions and the use of macros.

It contains the following sections:

6.1 About the Unified Assembler Language

Unified Assembler Language (UAL) is a common syntax for A32 and T32 instructions. It supersedes earlier versions of both the A32 and T32 assembler languages.

Code that is written using UAL can be assembled for A32 or T32 for any Arm processor. `armasm` faults the use of unavailable instructions.

`armasm` can assemble code that is written in pre-UAL and UAL syntax.

By default, `armasm` expects source code to be written in UAL. `armasm` accepts UAL syntax if any of the directives `CODE32`, `ARM`, or `THUMB` is used or if you assemble with any of the `--32`, `--arm`, or `--thumb` command-line options. `armasm` also accepts source code that is written in pre-UAL A32 assembly language when you assemble with the `CODE32` or `ARM` directive.

`armasm` accepts source code that is written in pre-UAL T32 assembly language when you assemble using the `--16` command-line option, or the `CODE16` directive in the source code.

Note

The pre-UAL T32 assembly language does not support 32-bit T32 instructions.

Related references

11.1 --16 on page 11-226.
21.7 ARM or CODE32 directive on page 21-1653.
21.11 CODE16 directive on page 21-1657.
21.65 THUMB directive on page 21-1719.
11.2 --32 on page 11-227.
11.4 --arm on page 11-230.
11.59 --thumb on page 11-287.

6.2 Syntax differences between UAL and A64 assembly language

UAL is the assembler syntax that is used by the A32 and T32 instruction sets. A64 assembly language is the assembler syntax that is used by the A64 instruction set.

UAL in Armv8 is unchanged from Armv7.

The general statement format and operand order of A64 assembly language is the same as UAL, but there are some differences between them. The following table describes the main differences:

Table 6-1 Syntax differences between UAL and A64 assembly language

UAL	A64
You make an instruction conditional by appending a condition code suffix directly to the mnemonic, with no delimiter. For example: `BEQ label`	For conditionally executed instructions, you separate the condition code suffix from the mnemonic using a `.` delimiter. For example: `B.EQ label`
Apart from the `IT` instruction, there are no unconditionally executed integer instructions that use a condition code as an operand.	A64 provides several unconditionally executed instructions that use a condition code as an operand. For these instructions, you specify the condition code to test for in the final operand position. For example: `CSEL w1,w2,w3,EQ`
The `.W` and `.N` instruction width specifiers control whether the assembler generates a 32-bit or 16-bit encoding for a T32 instruction.	A64 is a fixed width 32-bit instruction set so does not support `.W` and `.N` qualifiers.
The core register names are R0-R15.	Qualify register names to indicate the operand data size, either 32-bit (W0-W31) or 64-bit (X0-X31).
You can refer to registers R13, R14, and R15 as synonyms for SP, LR, and PC respectively.	In AArch64, there is no register that is named W31 or X31. Instead, you can refer to register 31 as SP, WZR, or XZR, depending on the context. You cannot refer to PC either by name or number. LR is an alias for register 30.
A32 has no equivalent of the extend operators.	You can specify an extend operator in several instructions to control how a portion of the second source register value is sign or zero extended. For example, in the following instruction, `UXTB` is the extend type (zero extend, byte) and `#2` is an optional left shift amount: `ADD X1, X2, W3, UXTB #2`

6.3 Register usage in subroutine calls

You use branch instructions to call and return from subroutines. The Procedure Call Standard for the Arm Architecture defines how to use registers in subroutine calls.

A subroutine is a block of code that performs a task based on some arguments and optionally returns a result. By convention, you use registers R0 to R3 to pass arguments to subroutines, and R0 to pass a result back to the callers. A subroutine that requires more than four inputs uses the stack for the additional inputs.

To call subroutines, use a branch and link instruction. The syntax is:

```
BL  destination
```

where *destination* is usually the label on the first instruction of the subroutine.

destination can also be a PC-relative expression.

The BL instruction:

- Places the return address in the link register.
- Sets the PC to the address of the subroutine.

After the subroutine code has executed you can use a BX LR instruction to return.

Note

Calls between separately assembled or compiled modules must comply with the restrictions and conventions defined by the *Procedure Call Standard for the Arm® Architecture*.

Example

The following example shows a subroutine, doadd, that adds the values of two arguments and returns a result in R0:

```
        AREA    subrout, CODE, READONLY     ; Name this block of code
        ENTRY                               ; Mark first instruction to execute
start   MOV     r0, #10                     ; Set up parameters
        MOV     r1, #3
        BL      doadd                       ; Call subroutine
stop    MOV     r0, #0x18                   ; angel_SWIreason_ReportException
        LDR     r1, =0x20026                ; ADP_Stopped_ApplicationExit
        SVC     #0x123456                   ; A32 semihosting (formerly SWI)
doadd   ADD     r0, r0, r1                  ; Subroutine code
        BX      lr                          ; Return from subroutine
        END                                 ; Mark end of file
```

Related concepts

6.17 Stack operations for nested subroutines on page 6-124.

Related references

13.20 BL on page 13-366.
13.22 BX, BXNS on page 13-370.

Related information

Procedure Call Standard for the Arm Architecture.
Procedure Call Standard for the Arm 64-bit Architecture (AArch64).

6.4 Load immediate values

To represent some immediate values, you might have to use a sequence of instructions rather than a single instruction.

A32 and T32 instructions can only be 32 bits wide. You can use a `MOV` or `MVN` instruction to load a register with an immediate value from a range that depends on the instruction set. Certain 32-bit values cannot be represented as an immediate operand to a single 32-bit instruction, although you can load these values from memory in a single instruction.

You can load any 32-bit immediate value into a register with two instructions, a `MOV` followed by a `MOVT`. Or, you can use a pseudo-instruction, `MOV32`, to construct the instruction sequence for you.

You can also use the `LDR` pseudo-instruction to load immediate values into a register.

You can include many commonly-used immediate values directly as operands within data processing instructions, without a separate load operation. The range of immediate values that you can include as operands in 16-bit T32 instructions is much smaller.

Related concepts

6.5 Load immediate values using MOV and MVN on page 6-106.
6.6 Load immediate values using MOV32 on page 6-109.
6.7 Load immediate values using LDR Rd, =const on page 6-110.

Related references

13.54 LDR pseudo-instruction on page 13-417.

6.5 Load immediate values using MOV and MVN

The MOV and MVN instructions can write a range of immediate values to a register.

In A32:

- MOV can load any 8-bit immediate value, giving a range of 0x0-0xFF (0-255).

  It can also rotate these values by any even number.

  These values are also available as immediate operands in many data processing operations, without being loaded in a separate instruction.
- MVN can load the bitwise complements of these values. The numerical values are -(*n*+1), where *n* is the value available in MOV.
- MOV can load any 16-bit number, giving a range of 0x0-0xFFFF (0-65535).

The following table shows the range of 8-bit values that can be loaded in a single A32 MOV or MVN instruction (for data processing operations). The value to load must be a multiple of the value shown in the Step column.

Table 6-2 A32 state immediate values (8-bit)

Binary	Decimal	Step	Hexadecimal	MVN value[a]	Notes
00000000000000000000000abcdefgh	0-255	1	0-0xFF	-1 to -256	-
00000000000000000000abcdefgh00	0-1020	4	0-0x3FC	-4 to -1024	-
00000000000000000000abcdefgh0000	0-4080	16	0-0xFF0	-16 to -4096	-
00000000000000000000abcdefgh000000	0-16320	64	0-0x3FC0	-64 to -16384	-
...	...	...	...	...	-
abcdefgh000000000000000000000000	0-255 x 2^{24}	2^{24}	0-0xFF000000	1-256 x -2^{24}	-
cdefgh000000000000000000000000ab	(bit pattern)	-	-	(bit pattern)	See b in Note
efgh000000000000000000000000abcd	(bit pattern)	-	-	(bit pattern)	See b in Note
gh000000000000000000000000abcdef	(bit pattern)	-	-	(bit pattern)	See b in Note

The following table shows the range of 16-bit values that can be loaded in a single MOV A32 instruction:

Table 6-3 A32 state immediate values in MOV instructions

Binary	Decimal	Step	Hexadecimal	MVN value	Notes
0000000000000000abcdefghijklmnop	0-65535	1	0-0xFFFF	-	See c in Note

Note

These notes give extra information on both tables.

a

The MVN values are only available directly as operands in MVN instructions.

b

These values are available in A32 only. All the other values in this table are also available in 32-bit T32 instructions.

c

These values are not available directly as operands in other instructions.

In T32:

- The 32-bit MOV instruction can load:
  - Any 8-bit immediate value, giving a range of 0x0-0xFF (0-255).
  - Any 8-bit immediate value, shifted left by any number.
  - Any 8-bit pattern duplicated in all four bytes of a register.
  - Any 8-bit pattern duplicated in bytes 0 and 2, with bytes 1 and 3 set to 0.
  - Any 8-bit pattern duplicated in bytes 1 and 3, with bytes 0 and 2 set to 0.

  These values are also available as immediate operands in many data processing operations, without being loaded in a separate instruction.
- The 32-bit MVN instruction can load the bitwise complements of these values. The numerical values are -(*n*+1), where *n* is the value available in MOV.
- The 32-bit MOV instruction can load any 16-bit number, giving a range of 0x0-0xFFFF (0-65535). These values are not available as immediate operands in data processing operations.

In architectures with T32, the 16-bit T32 MOV instruction can load any immediate value in the range 0-255.

The following table shows the range of values that can be loaded in a single 32-bit T32 MOV or MVN instruction (for data processing operations). The value to load must be a multiple of the value shown in the Step column.

Table 6-4 32-bit T32 immediate values

Binary	Decimal	Step	Hexadecimal	MVN value[a]	Notes
000000000000000000000000abcdefgh	0-255	1	0x0-0xFF	-1 to -256	-
000000000000000000000000abcdefgh0	0-510	2	0x0-0x1FE	-2 to -512	-
000000000000000000000000abcdefgh00	0-1020	4	0x0-0x3FC	-4 to -1024	-
...	...	...	...	...	-
0abcdefgh00000000000000000000000	0-255 x 2^{23}	2^{23}	0x0-0x7F800000	1-256 x -2^{23}	-
abcdefgh00000000000000000000000	0-255 x 2^{24}	2^{24}	0x0-0xFF000000	1-256 x -2^{24}	-
abcdefghabcdefghabcdefghabcdefgh	(bit pattern)	-	0xXYXYXYXY	0xXYXYXYXY	-
00000000abcdefgh00000000abcdefgh	(bit pattern)	-	0x00XY00XY	0xFFXYFFXY	-
abcdefgh00000000abcdefgh00000000	(bit pattern)	-	0xXY00XY00	0xXYFFXYFF	-
000000000000000000000abcdefghijkl	0-4095	1	0x0-0xFFF	-	See b in Note

The following table shows the range of 16-bit values that can be loaded by the MOV 32-bit T32 instruction:

Table 6-5 32-bit T32 immediate values in MOV instructions

Binary	Decimal	Step	Hexadecimal	MVN value	Notes
0000000000000000abcdefghijklmnop	0-65535	1	0x0-0xFFFF	-	See c in Note

Note

These notes give extra information on the tables.

a

The MVN values are only available directly as operands in MVN instructions.

b

These values are available directly as operands in ADD, SUB, and MOV instructions, but not in MVN or any other data processing instructions.

c

These values are only available in `MOV` instructions.

In both A32 and T32, you do not have to decide whether to use `MOV` or `MVN`. The assembler uses whichever is appropriate. This is useful if the value is an assembly-time variable.

If you write an instruction with an immediate value that is not available, the assembler reports the error: `Immediate` *n* `out of range for this operation.`

Related concepts

6.4 Load immediate values on page 6-105.

6.6 Load immediate values using MOV32

To load any 32-bit immediate value, a pair of `MOV` and `MOVT` instructions is equivalent to a `MOV32` pseudo-instruction.

Both A32 and T32 instruction sets include:

- A `MOV` instruction that can load any value in the range `0x00000000` to `0x0000FFFF` into a register.
- A `MOVT` instruction that can load any value in the range `0x0000` to `0xFFFF` into the most significant half of a register, without altering the contents of the least significant half.

You can use these two instructions to construct any 32-bit immediate value in a register. Alternatively, you can use the `MOV32` pseudo-instruction. The assembler generates the `MOV`, `MOVT` instruction pair for you.

You can also use the `MOV32` instruction to load addresses into registers by using a label or any PC-relative expression in place of an immediate value. The assembler puts a relocation directive into the object file for the linker to resolve the address at link-time.

Related concepts

12.5 Register-relative and PC-relative expressions on page 12-302.

Related references

13.64 MOV32 pseudo-instruction on page 13-433.

6.7 Load immediate values using LDR Rd, =const

The `LDR Rd,=const` pseudo-instruction generates the most efficient single instruction to load any 32-bit number.

You can use this pseudo-instruction to generate constants that are out of range of the `MOV` and `MVN` instructions.

The `LDR` pseudo-instruction generates the most efficient single instruction for the specified immediate value:

- If the immediate value can be constructed with a single `MOV` or `MVN` instruction, the assembler generates the appropriate instruction.
- If the immediate value cannot be constructed with a single `MOV` or `MVN` instruction, the assembler:
  - Places the value in a *literal pool* (a portion of memory embedded in the code to hold constant values).
  - Generates an `LDR` instruction with a PC-relative address that reads the constant from the literal pool.

  For example:

```
LDR      rn, [pc, #offset to literal pool]
                       ; load register n with one word
                       ; from the address [pc + offset]
```

  You must ensure that there is a literal pool within range of the `LDR` instruction generated by the assembler.

Related concepts

6.8 Literal pools on page 6-111.

Related references

13.54 LDR pseudo-instruction on page 13-417.

6.8 Literal pools

The assembler uses literal pools to store some constant data in code sections. You can use the LTORG directive to ensure a literal pool is within range.

The assembler places a literal pool at the end of each section. The end of a section is defined either by the END directive at the end of the assembly or by the AREA directive at the start of the following section. The END directive at the end of an included file does not signal the end of a section.

In large sections the default literal pool can be out of range of one or more LDR instructions. The offset from the PC to the constant must be:

- Less than 4KB in A32 or T32 code when the 32-bit LDR instruction is available, but can be in either direction.
- Forward and less than 1KB when only the 16-bit T32 LDR instruction is available.

When an LDR Rd,=const pseudo-instruction requires the immediate value to be placed in a literal pool, the assembler:

- Checks if the value is available and addressable in any previous literal pools. If so, it addresses the existing constant.
- Attempts to place the value in the next literal pool if it is not already available.

If the next literal pool is out of range, the assembler generates an error message. In this case you must use the LTORG directive to place an additional literal pool in the code. Place the LTORG directive after the failed LDR pseudo-instruction, and within the valid range for an LDR instruction.

You must place literal pools where the processor does not attempt to execute them as instructions. Place them after unconditional branch instructions, or after the return instruction at the end of a subroutine.

Example of placing literal pools

The following example shows the placement of literal pools. The instructions listed as comments are the A32 instructions generated by the assembler.

```
        AREA     Loadcon, CODE, READONLY
        ENTRY                       ; Mark first instruction to execute
start
        BL       func1              ; Branch to first subroutine
        BL       func2              ; Branch to second subroutine
stop
        MOV      r0, #0x18          ; angel_SWIreason_ReportException
        LDR      r1, =0x20026       ; ADP_Stopped_ApplicationExit
        SVC      #0x123456          ; A32 semihosting (formerly SWI)
func1
        LDR      r0, =42            ; => MOV R0, #42
        LDR      r1, =0x55555555    ; => LDR R1, [PC, #offset to
                                    ; Literal Pool 1]
        LDR      r2, =0xFFFFFFFF    ; => MVN R2, #0
        BX       lr
        LTORG                       ; Literal Pool 1 contains
                                    ; literal Ox55555555
func2
        LDR      r3, =0x55555555    ; => LDR R3, [PC, #offset to
                                    ; Literal Pool 1]
        ; LDR r4, =0x66666666       ; If this is uncommented it
                                    ; fails, because Literal Pool 2
                                    ; is out of reach
        BX       lr
LargeTable
        SPACE    4200               ; Starting at the current location,
                                    ; clears a 4200 byte area of memory
                                    ; to zero
        END                         ; Literal Pool 2 is inserted here,
                                    ; but is out of range of the LDR
                                    ; pseudo-instruction that needs it
```

Related concepts

6.7 Load immediate values using LDR Rd, =const on page 6-110.

Related references

21.50 LTORG on page 21-1699.

6.9 Load addresses into registers

It is often necessary to load an address into a register. There are several ways to do this.

For example, you might have to load the address of a variable, a string literal, or the start location of a jump table.

Addresses are normally expressed as offsets from a label, or from the current PC or other register.

You can load an address into a register either:

- Using the instruction `ADR`.
- Using the pseudo-instruction `ADRL`.
- Using the pseudo-instruction `MOV32`.
- From a literal pool using the pseudo-instruction `LDR Rd,=Label`.

Related concepts

6.10 Load addresses to a register using ADR on page 6-114.
6.11 Load addresses to a register using ADRL on page 6-116.
6.6 Load immediate values using MOV32 on page 6-109.
6.12 Load addresses to a register using LDR Rd, =label on page 6-117.

6.10 Load addresses to a register using ADR

The ADR instruction loads an address within a certain range, without performing a data load.

ADR accepts a PC-relative expression, that is, a label with an optional offset where the address of the label is relative to the PC.

Note

The label used with ADR must be within the same code section. The assembler faults references to labels that are out of range in the same section.

The available range of addresses for the ADR instruction depends on the instruction set and encoding:

A32
: Any value that can be produced by rotating an 8-bit value right by any even number of bits within a 32-bit word. The range is relative to the PC.

32-bit T32 encoding
: ±4095 bytes to a byte, halfword, or word-aligned address.

16-bit T32 encoding
: 0 to 1020 bytes. *label* must be word-aligned. You can use the ALIGN directive to ensure this.

Example of a jump table implementation with ADR

This example shows A32 code that implements a jump table. Here, the ADR instruction loads the address of the jump table.

```
        AREA   Jump, CODE, READONLY ; Name this block of code
        ARM                         ; Following code is A32 code
num     EQU    2                    ; Number of entries in jump table
        ENTRY                       ; Mark first instruction to execute
start                               ; First instruction to call
        MOV    r0, #0               ; Set up the three arguments
        MOV    r1, #3
        MOV    r2, #2
        BL     arithfunc            ; Call the function
stop
        MOV    r0, #0x18            ; angel_SWIreason_ReportException
        LDR    r1, =0x20026         ; ADP_Stopped_ApplicationExit
        SVC    #0x123456            ; A32 semihosting (formerly SWI)
arithfunc                           ; Label the function
        CMP    r0, #num             ; Treat function code as unsigned
                                    ; integer
        BXHS   lr                   ; If code is >= num then return
        ADR    r3, JumpTable        ; Load address of jump table
        LDR    pc, [r3,r0,LSL#2]    ; Jump to the appropriate routine
JumpTable
        DCD    DoAdd
        DCD    DoSub
DoAdd
        ADD    r0, r1, r2           ; Operation 0
        BX     lr                   ; Return
DoSub
        SUB    r0, r1, r2           ; Operation 1
        BX     lr                   ; Return
        END                         ; Mark the end of this file
```

In this example, the function arithfunc takes three arguments and returns a result in R0. The first argument determines the operation to be carried out on the second and third arguments:

argument1=0
: Result = argument2 + argument3.

argument1=1
: Result = argument2 – argument3.

The jump table is implemented with the following instructions and assembler directives:

`EQU`

Is an assembler directive. You use it to give a value to a symbol. In this example, it assigns the value 2 to *num*. When *num* is used elsewhere in the code, the value 2 is substituted. Using `EQU` in this way is similar to using `#define` to define a constant in C.

`DCD`

Declares one or more words of store. In this example, each `DCD` stores the address of a routine that handles a particular clause of the jump table.

`LDR`

The `LDR PC,[R3,R0,LSL#2]` instruction loads the address of the required clause of the jump table into the PC. It:

- Multiplies the clause number in `R0` by 4 to give a word offset.
- Adds the result to the address of the jump table.
- Loads the contents of the combined address into the PC.

Related concepts

6.12 Load addresses to a register using LDR Rd, =label on page 6-117.

6.11 Load addresses to a register using ADRL on page 6-116.

Related references

13.10 ADR (PC-relative) on page 13-349.

6.11 Load addresses to a register using ADRL

The ADRL pseudo-instruction loads an address within a certain range, without performing a data load. The range is wider than that of the ADR instruction.

ADRL accepts a PC-relative expression, that is, a label with an optional offset where the address of the label is relative to the current PC.

Note

The label used with ADRL must be within the same code section. The assembler faults references to labels that are out of range in the same section.

The assembler converts an ADRL *rn,label* pseudo-instruction by generating:

- Two data processing instructions that load the address, if it is in range.
- An error message if the address cannot be constructed in two instructions.

The available range depends on the instruction set and encoding.

A32
: Any value that can be generated by two ADD or two SUB instructions. That is, any value that can be produced by the addition of two values, each of which is 8 bits rotated right by any even number of bits within a 32-bit word. The range is relative to the PC.

32-bit T32 encoding
: ±1MB to a byte, halfword, or word-aligned address.

16-bit T32 encoding
: ADRL is not available.

Related concepts

6.10 Load addresses to a register using ADR on page 6-114.

6.12 Load addresses to a register using LDR Rd, =label on page 6-117.

6.12 Load addresses to a register using LDR Rd, =label

The `LDR Rd,=label` pseudo-instruction places an address in a literal pool and then loads the address into a register.

`LDR Rd,=label` can load any 32-bit numeric value into a register. It also accepts PC-relative expressions such as labels, and labels with offsets.

The assembler converts an `LDR Rd,=label` pseudo-instruction by:

- Placing the address of *label* in a literal pool (a portion of memory embedded in the code to hold constant values).
- Generating a PC-relative `LDR` instruction that reads the address from the literal pool, for example:

```
LDR rn [pc, #offset_to_literal_pool]
                    ; load register n with one word
                    ; from the address [pc + offset]
```

  You must ensure that the literal pool is within range of the `LDR` pseudo-instruction that needs to access it.

Example of loading using LDR Rd, =label

The following example shows a section with two literal pools. The final `LDR` pseudo-instruction needs to access the second literal pool, but it is out of range. Uncommenting this line causes the assembler to generate an error.

The instructions listed in the comments are the A32 instructions generated by the assembler.

```
        AREA   LDRlabel, CODE, READONLY
        ENTRY                    ; Mark first instruction to execute
start
        BL     func1             ; Branch to first subroutine
        BL     func2             ; Branch to second subroutine
stop
        MOV    r0, #0x18         ; angel_SWIreason_ReportException
        LDR    r1, =0x20026      ; ADP_Stopped_ApplicationExit
        SVC    #0x123456         ; A32 semihosting (formerly SWI)
func1
        LDR    r0, =start        ; => LDR r0,[PC, #offset into Literal Pool 1]
        LDR    r1, =Darea + 12   ; => LDR r1,[PC, #offset into Literal Pool 1]
        LDR    r2, =Darea + 6000 ; => LDR r2,[PC, #offset into Literal Pool 1]
        BX     lr                ; Return
        LTORG                    ; Literal Pool 1
func2
        LDR    r3, =Darea + 6000 ; => LDR r3,[PC, #offset into Literal Pool 1]
                                 ; (sharing with previous literal)
        ; LDR   r4, =Darea + 6004 ; If uncommented, produces an error because
                                 ; Literal Pool 2 is out of range.
        BX     lr                ; Return
Darea   SPACE  8000              ; Starting at the current location, clears
                                 ; a 8000 byte area of memory to zero.
        END                      ; Literal Pool 2 is automatically inserted
                                 ; after the END directive.
                                 ; It is out of range of all the LDR
                                 ; pseudo-instructions in this example.
```

Example of string copy

The following example shows an A32 code routine that overwrites one string with another. It uses the `LDR` pseudo-instruction to load the addresses of the two strings from a data section. The following are particularly significant:

`DCB`

The `DCB` directive defines one or more bytes of store. In addition to integer values, `DCB` accepts quoted strings. Each character of the string is placed in a consecutive byte.

`LDR, STR`

The `LDR` and `STR` instructions use post-indexed addressing to update their address registers. For example, the instruction:

```
LDRB    r2,[r1],#1
```

loads `R2` with the contents of the address pointed to by `R1` and then increments `R1` by 1.

The example also shows how, unlike the `ADR` and `ADRL` pseudo-instructions, you can use the `LDR` pseudo-instruction with labels that are outside the current section. The assembler places a relocation directive in the object code when the source file is assembled. The relocation directive instructs the linker to resolve the address at link time. The address remains valid wherever the linker places the section containing the `LDR` and the literal pool.

```
        AREA    StrCopy, CODE, READONLY
        ENTRY                       ; Mark first instruction to execute
start
        LDR     r1, =srcstr         ; Pointer to first string
        LDR     r0, =dststr         ; Pointer to second string
        BL      strcopy             ; Call subroutine to do copy
stop
        MOV     r0, #0x18           ; angel_SWIreason_ReportException
        LDR     r1, =0x20026        ; ADP_Stopped_ApplicationExit
        SVC     #0x123456           ; A32 semihosting (formerly SWI)
strcopy
        LDRB    r2, [r1],#1         ; Load byte and update address
        STRB    r2, [r0],#1         ; Store byte and update address
        CMP     r2, #0              ; Check for zero terminator
        BNE     strcopy             ; Keep going if not
        MOV     pc,lr               ; Return
        AREA    Strings, DATA, READWRITE
srcstr  DCB     "First string - source",0
dststr  DCB     "Second string - destination",0
        END
```

Related concepts

6.11 Load addresses to a register using ADRL on page 6-116.
6.7 Load immediate values using LDR Rd, =const on page 6-110.

Related references

13.54 LDR pseudo-instruction on page 13-417.
21.15 DCB on page 21-1661.

6.13 Other ways to load and store registers

You can load and store registers using `LDR`, `STR` and `MOV` (register) instructions.

You can load any 32-bit value from memory into a register with an `LDR` data load instruction. To store registers into memory you can use the `STR` data store instruction.

You can use the `MOV` instruction to move any 32-bit data from one register to another.

Related concepts

6.14 Load and store multiple register instructions on page 6-120.

6.15 Load and store multiple register instructions in A32 and T32 on page 6-121.

Related references

13.63 MOV on page 13-431.

6.14 Load and store multiple register instructions

The A32 and T32 instruction sets include instructions that load and store multiple registers. These instructions can provide a more efficient way of transferring the contents of several registers to and from memory than using single register loads and stores.

Multiple register transfer instructions are most often used for block copy and for stack operations at subroutine entry and exit. The advantages of using a multiple register transfer instruction instead of a series of single data transfer instructions include:

- Smaller code size.
- A single instruction fetch overhead, rather than many instruction fetches.
- On uncached Arm processors, the first word of data transferred by a load or store multiple is always a nonsequential memory cycle, but all subsequent words transferred can be sequential memory cycles. Sequential memory cycles are faster in most systems.

Note

The lowest numbered register is transferred to or from the lowest memory address accessed, and the highest numbered register to or from the highest address accessed. The order of the registers in the register list in the instructions makes no difference.

You can use the `--diag_warning 1206` assembler command line option to check that registers in register lists are specified in increasing order.

Related concepts

6.15 Load and store multiple register instructions in A32 and T32 on page 6-121.
6.16 Stack implementation using LDM and STM on page 6-122.
6.17 Stack operations for nested subroutines on page 6-124.
6.18 Block copy with LDM and STM on page 6-125.

6.15 Load and store multiple register instructions in A32 and T32

Instructions are available in both the A32 and T32 instruction sets to load and store multiple registers.

They are:

LDM
: Load Multiple registers.

STM
: Store Multiple registers.

PUSH
: Store multiple registers onto the stack and update the stack pointer.

POP
: Load multiple registers off the stack, and update the stack pointer.

In `LDM` and `STM` instructions:

- The list of registers loaded or stored can include:
  - In A32 instructions, any or all of R0-R12, SP, LR, and PC.
  - In 32-bit T32 instructions, any or all of R0-R12, and optionally LR or PC (`LDM` only) with some restrictions.
  - In 16-bit T32 instructions, any or all of R0-R7.
- The address must be word-aligned. It can be:
  - Incremented after each transfer.
  - Incremented before each transfer (A32 instructions only).
  - Decremented after each transfer (A32 instructions only).
  - Decremented before each transfer (not in 16-bit encoded T32 instructions).
- The base register can be either:
  - Updated to point to the next block of data in memory.
  - Left as it was before the instruction.

When the base register is updated to point to the next block in memory, this is called writeback, that is, the adjusted address is written back to the base register.

In `PUSH` and `POP` instructions:

- The stack pointer (SP) is the base register, and is always updated.
- The address is incremented after each transfer in `POP` instructions, and decremented before each transfer in `PUSH` instructions.
- The list of registers loaded or stored can include:
  - In A32 instructions, any or all of R0-R12, SP, LR, and PC.
  - In 32-bit T32 instructions, any or all of R0-R12, and optionally LR or PC (`POP` only) with some restrictions.
  - In 16-bit T32 instructions, any or all of R0-R7, and optionally LR (`PUSH` only) or PC (`POP` only).

Note

Use of SP in the list of registers in these A32 instructions is deprecated.

A32 `STM` and `PUSH` instructions that use PC in the list of registers, and A32 `LDM` and `POP` instructions that use both PC and LR in the list of registers are deprecated.

Related concepts

6.14 Load and store multiple register instructions on page 6-120.

6.16 Stack implementation using LDM and STM

You can use the `LDM` and `STM` instructions to implement pop and push operations respectively. You use a suffix to indicate the stack type.

The load and store multiple instructions can update the base register. For stack operations, the base register is usually the stack pointer, SP. This means that you can use these instructions to implement push and pop operations for any number of registers in a single instruction.

The load and store multiple instructions can be used with several types of stack:

Descending or ascending
: The stack grows downwards, starting with a high address and progressing to a lower one (a *descending* stack), or upwards, starting from a low address and progressing to a higher address (an *ascending* stack).

Full or empty
: The stack pointer can either point to the last item in the stack (a *full* stack), or the next free space on the stack (an *empty* stack).

To make it easier for the programmer, stack-oriented suffixes can be used instead of the increment or decrement, and before or after suffixes. The following table shows the stack-oriented suffixes and their equivalent addressing mode suffixes for load and store instructions:

Table 6-6 Stack-oriented suffixes and equivalent addressing mode suffixes

Stack-oriented suffix	For store or push instructions	For load or pop instructions
FD (Full Descending stack)	DB (Decrement Before)	IA (Increment After)
FA (Full Ascending stack)	IB (Increment Before)	DA (Decrement After)
ED (Empty Descending stack)	DA (Decrement After)	IB (Increment Before)
EA (Empty Ascending stack)	IA (Increment After)	DB (Decrement Before)

The following table shows the load and store multiple instructions with the stack-oriented suffixes for the various stack types:

Table 6-7 Suffixes for load and store multiple instructions

Stack type	Store	Load
Full descending	`STMFD` (`STMDB`, Decrement Before)	`LDMFD` (`LDM`, increment after)
Full ascending	`STMFA` (`STMIB`, Increment Before)	`LDMFA` (`LDMDA`, Decrement After)
Empty descending	`STMED` (`STMDA`, Decrement After)	`LDMED` (`LDMIB`, Increment Before)
Empty ascending	`STMEA` (`STM`, increment after)	`LDMEA` (`LDMDB`, Decrement Before)

For example:

```
STMFD    sp!, {r0-r5}  ; Push onto a Full Descending Stack
LDMFD    sp!, {r0-r5}  ; Pop from a Full Descending Stack
```

Note

The *Procedure Call Standard for the Arm® Architecture* (AAPCS), and `armclang` always use a full descending stack.

The `PUSH` and `POP` instructions assume a full descending stack. They are the preferred synonyms for `STMDB` and `LDM` with writeback.

Related concepts

6.14 Load and store multiple register instructions on page 6-120.

Related references

13.49 LDM on page 13-407.

Related information

Procedure Call Standard for the Arm Architecture.

6.17 Stack operations for nested subroutines

Stack operations can be very useful at subroutine entry and exit to avoid losing register contents if other subroutines are called.

At the start of a subroutine, any working registers required can be stored on the stack, and at exit they can be popped off again.

In addition, if the link register is pushed onto the stack at entry, additional subroutine calls can be made safely without causing the return address to be lost. If you do this, you can also return from a subroutine by popping the PC off the stack at exit, instead of popping the LR and then moving that value into the PC. For example:

```
subroutine  PUSH    {r5-r7,lr} ; Push work registers and lr
            ; code
            BL      somewhere_else
            ; code
            POP     {r5-r7,pc} ; Pop work registers and pc
```

Related concepts

6.3 Register usage in subroutine calls on page 6-104.

6.14 Load and store multiple register instructions on page 6-120.

Related information

Procedure Call Standard for the Arm Architecture.

Procedure Call Standard for the Arm 64-bit Architecture (AArch64).

6.18 Block copy with LDM and STM

You can sometimes make code more efficient by using LDM and STM instead of LDR and STR instructions.

Example of block copy without LDM and STM

The following example is an A32 code routine that copies a set of words from a source location to a destination a single word at a time:

```
        AREA  Word, CODE, READONLY  ; name the block of code
num     EQU   20                    ; set number of words to be copied
        ENTRY                       ; mark the first instruction called
start
        LDR   r0, =src              ; r0 = pointer to source block
        LDR   r1, =dst              ; r1 = pointer to destination block
        MOV   r2, #num              ; r2 = number of words to copy
wordcopy
        LDR   r3, [r0], #4          ; load a word from the source and
        STR   r3, [r1], #4          ; store it to the destination
        SUBS  r2, r2, #1            ; decrement the counter
        BNE   wordcopy              ; ... copy more
stop
        MOV   r0, #0x18             ; angel_SWIreason_ReportException
        LDR   r1, =0x20026          ; ADP_Stopped_ApplicationExit
        SVC   #0x123456             ; A32 semihosting (formerly SWI)
        AREA  BlockData, DATA, READWRITE
src     DCD   1,2,3,4,5,6,7,8,1,2,3,4,5,6,7,8,1,2,3,4
dst     DCD   0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
        END
```

You can make this module more efficient by using LDM and STM for as much of the copying as possible. Eight is a sensible number of words to transfer at a time, given the number of available registers. You can find the number of eight-word multiples in the block to be copied (if R2 = number of words to be copied) using:

```
    MOVS   r3, r2, LSR #3    ; number of eight word multiples
```

You can use this value to control the number of iterations through a loop that copies eight words per iteration. When there are fewer than eight words left, you can find the number of words left (assuming that R2 has not been corrupted) using:

```
    ANDS   r2, r2, #7
```

Example of block copy using LDM and STM

The following example lists the block copy module rewritten to use LDM and STM for copying:

```
      AREA   Block, CODE, READONLY ; name this block of code
num   EQU    20                    ; set number of words to be copied
      ENTRY                       ; mark the first instruction called
start
      LDR    r0, =src              ; r0 = pointer to source block
      LDR    r1, =dst              ; r1 = pointer to destination block
      MOV    r2, #num              ; r2 = number of words to copy
      MOV    sp, #0x400            ; Set up stack pointer (sp)
blockcopy
      MOVS   r3,r2, LSR #3         ; Number of eight word multiples
      BEQ    copywords             ; Fewer than eight words to move?
      PUSH   {r4-r11}              ; Save some working registers
octcopy
      LDM    r0!, {r4-r11}         ; Load 8 words from the source
      STM    r1!, {r4-r11}         ; and put them at the destination
      SUBS   r3, r3, #1            ; Decrement the counter
      BNE    octcopy               ; ... copy more
      POP    {r4-r11}              ; Don't require these now - restore
                                   ; originals
copywords
      ANDS   r2, r2, #7            ; Number of odd words to copy
      BEQ    stop                  ; No words left to copy?
wordcopy
      LDR    r3, [r0], #4          ; Load a word from the source and
      STR    r3, [r1], #4          ; store it to the destination
      SUBS   r2, r2, #1            ; Decrement the counter
      BNE    wordcopy              ; ... copy more
stop
      MOV    r0, #0x18             ; angel_SWIreason_ReportException
```

```
      LDR    r1, =0x20026          ; ADP_Stopped_ApplicationExit
      SVC   #0x123456              ; A32 semihosting (formerly SWI)
      AREA   BlockData, DATA, READWRITE
src   DCD    1,2,3,4,5,6,7,8,1,2,3,4,5,6,7,8,1,2,3,4
dst   DCD    0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
      END
```

Note

The purpose of this example is to show the use of the `LDM` and `STM` instructions. There are other ways to perform bulk copy operations, the most efficient of which depends on many factors and is outside the scope of this document.

Related information

What is the fastest way to copy memory on a Cortex-A8?.

6.19 Memory accesses

Many load and store instructions support different addressing modes.

Offset addressing

The offset value is applied to an address obtained from the base register. The result is used as the address for the memory access. The base register is unchanged. The assembly language syntax for this mode is:

```
[Rn, offset]
```

Pre-indexed addressing

The offset value is applied to an address obtained from the base register. The result is used as the address for the memory access, and written back into the base register. The assembly language syntax for this mode is:

```
[Rn, offset]!
```

Post-indexed addressing

The address obtained from the base register is used, unchanged, as the address for the memory access. The offset value is applied to the address, and written back into the base register. The assembly language syntax for this mode is:

```
[Rn], offset
```

In each case, `Rn` is the base register and `offset` can be:

- An immediate constant.
- An index register, `Rm`.
- A shifted index register, such as `Rm, LSL #shift`.

Related concepts

8.15 Address alignment in A32/T32 code on page 8-178.

Related references

3.4 Registers in AArch32 state on page 3-67.

6.20 The Read-Modify-Write operation

The read-modify-write operation ensures that you modify only the specific bits in a system register that you want to change.

Individual bits in a system register control different system functionality. Modifying the wrong bits in a system register might cause your program to behave incorrectly.

```
VMRS    r10,FPSCR             ; copy FPSCR into the general-purpose r10
BIC     r10,r10,#0x00370000   ; clear STRIDE bits[21:20] and LEN bits[18:16]
ORR     r10,r10,#0x00030000   ; set bits[17:16] (STRIDE =1 and LEN = 4)
VMSR    FPSCR,r10             ; copy r10 back into FPSCR
```

To read-modify-write a system register, the instruction sequence is:

1. The first instruction copies the value from the target system register to a temporary general-purpose register.
2. The next one or more instructions modify the required bits in the general-purpose register. This can be one or both of:
   - `BIC` to clear to 0 only the bits that must be cleared.
   - `ORR` to set to 1 only the bits that must be set.
3. The final instruction writes the value from the general-purpose register to the target system register.

Related concepts

3.6 Register accesses in AArch32 state on page 3-70.

Related references

3.10 The Q flag in AArch32 state on page 3-74.

13.68 MRS (PSR to general-purpose register) on page 13-437.

13.71 MSR (general-purpose register to PSR) on page 13-441.

14.72 VMRS on page 14-679.

6.21 Optional hash with immediate constants

You do not have to specify a hash before an immediate constant in any instruction syntax.

This applies to A32, T32, Advanced SIMD, and floating-point instructions. For example, the following are valid instructions:

```
BKPT 100
MOVT R1, 256
VCEQ.I8 Q1, Q2, 0
```

By default, the assembler warns if you do not specify a hash:

```
WARNING: A1865W: '#' not seen before constant expression.
```

You can suppressed this with `--diag_suppress=1865`.

If you use the assembly code with another assembler, you are advised to use the # before all immediates. The disassembler always shows the # for clarity.

Related references

Chapter 13 A32 and T32 Instructions on page 13-327.
Chapter 14 Advanced SIMD Instructions (32-bit) on page 14-599.

6.22 Use of macros

A macro definition is a block of code enclosed between `MACRO` and `MEND` directives. It defines a name that you can use as a convenient alternative to repeating the block of code.

The main uses for a macro are:

- To make it easier to follow the logic of the source code by replacing a block of code with a single meaningful name.
- To avoid repeating a block of code several times.

Related concepts

6.23 Test-and-branch macro example on page 6-131.

6.24 Unsigned integer division macro example on page 6-132.

Related references

21.51 MACRO and MEND on page 21-1700.

6.23 Test-and-branch macro example

You can use a macro to perform a test-and-branch operation.

In A32 code, a test-and-branch operation requires two instructions to implement.

You can define a macro such as this:

```
        MACRO
$label  TestAndBranch  $dest, $reg, $cc
$label  CMP     $reg, #0
        B$cc    $dest
        MEND
```

The line after the `MACRO` directive is the *macro prototype statement*. This defines the name (`TestAndBranch`) you use to invoke the macro. It also defines parameters (`$label`, `$dest`, `$reg`, and `$cc`). Unspecified parameters are substituted with an empty string. For this macro you must give values for `$dest`, `$reg` and `$cc` to avoid syntax errors. The assembler substitutes the values you give into the code.

This macro can be invoked as follows:

```
test    TestAndBranch    NonZero, r0, NE
         ...
         ...
NonZero
```

After substitution this becomes:

```
test    CMP     r0, #0
        BNE     NonZero
         ...
         ...
NonZero
```

Related concepts

6.22 Use of macros on page 6-130.
6.24 Unsigned integer division macro example on page 6-132.
12.10 Numeric local labels on page 12-307.

6.24 Unsigned integer division macro example

You can use a macro to perform unsigned integer division.

The macro takes the following parameters:

`$Bot`
: The register that holds the divisor.

`$Top`
: The register that holds the dividend before the instructions are executed. After the instructions are executed, it holds the remainder.

`$Div`
: The register where the quotient of the division is placed. It can be `NULL` (`""`) if only the remainder is required.

`$Temp`
: A temporary register used during the calculation.

Example unsigned integer division with a macro

```
        MACRO
$Lab    DivMod  $Div,$Top,$Bot,$Temp
        ASSERT  $Top <> $Bot        ; Produce an error message if the
        ASSERT  $Top <> $Temp       ; registers supplied are
        ASSERT  $Bot <> $Temp       ; not all different
        IF      "$Div" <> ""
            ASSERT  $Div <> $Top    ; These three only matter if $Div
            ASSERT  $Div <> $Bot    ; is not null ("")
            ASSERT  $Div <> $Temp   ;
        ENDIF
$Lab
        MOV     $Temp, $Bot           ; Put divisor in $Temp
        CMP     $Temp, $Top, LSR #1   ; double it until
90      MOVLS   $Temp, $Temp, LSL #1  ; 2 * $Temp > $Top
        CMP     $Temp, $Top, LSR #1
        BLS     %b90                  ; The b means search backwards
        IF      "$Div" <> ""          ; Omit next instruction if $Div
                                      ; is null
            MOV     $Div, #0          ; Initialize quotient
        ENDIF
91      CMP     $Top, $Temp           ; Can we subtract $Temp?
        SUBCS   $Top, $Top,$Temp      ; If we can, do so
        IF      "$Div" <> ""          ; Omit next instruction if $Div
                                      ; is null
            ADC     $Div, $Div, $Div  ; Double $Div
        ENDIF
        MOV     $Temp, $Temp, LSR #1  ; Halve $Temp,
        CMP     $Temp, $Bot           ; and loop until
        BHS     %b91                  ; less than divisor
        MEND
```

The macro checks that no two parameters use the same register. It also optimizes the code produced if only the remainder is required.

To avoid multiple definitions of labels if `DivMod` is used more than once in the assembler source, the macro uses numeric local labels (90, 91).

The following example shows the code that this macro produces if it is invoked as follows:

```
ratio  DivMod  R0,R5,R4,R2
```

Output from the example division macro

```
        ASSERT  r5 <> r4          ; Produce an error if the
        ASSERT  r5 <> r2          ; registers supplied are
        ASSERT  r4 <> r2          ; not all different
        ASSERT  r0 <> r5          ; These three only matter if $Div
        ASSERT  r0 <> r4          ; is not null ("")
        ASSERT  r0 <> r2          ;
ratio
        MOV     r2, r4            ; Put divisor in $Temp
        CMP     r2, r5, LSR #1    ; double it until
90      MOVLS   r2, r2, LSL #1    ; 2 * r2 > r5
        CMP     r2, r5, LSR #1
```

```
        BLS     %b90            ; The b means search backwards
        MOV     r0, #0          ; Initialize quotient
91      CMP     r5, r2          ; Can we subtract r2?
        SUBCS   r5, r5, r2      ; If we can, do so
        ADC     r0, r0, r0      ; Double r0
        MOV     r2, r2, LSR #1  ; Halve r2,
        CMP     r2, r4          ; and loop until
        BHS     %b91            ; less than divisor
```

Related concepts

6.22 Use of macros on page 6-130.

6.23 Test-and-branch macro example on page 6-131.

12.10 Numeric local labels on page 12-307.

6.25 Instruction and directive relocations

The assembler can embed relocation directives in object files to indicate labels with addresses that are unknown at assembly time. The assembler can relocate several types of instruction.

A relocation is a directive embedded in the object file that enables source code to refer to a label whose target address is unknown or cannot be calculated at assembly time. The assembler emits a relocation in the object file, and the linker resolves this to the address where the target is placed.

The assembler relocates the data directives `DCB`, `DCW`, `DCWU`, `DCD`, and `DCDU` if their syntax contains an external symbol, that is a symbol declared using `IMPORT` or `EXTERN`. This causes the bottom 8, 16, or 32 bits of the address to be used at link-time.

The `REQUIRE` directive emits a relocation to signal to the linker that the target label must be present if the current section is present.

The assembler is permitted to emit a relocation for these instructions:

`LDR` (PC-relative)
: All A32 and T32 instructions, except the T32 doubleword instruction, can be relocated.

`PLD`, `PLDW`, and `PLI`
: All A32 and T32 instructions can be relocated.

`B`, `BL`, and `BLX`
: All A32 and T32 instructions can be relocated.

`CBZ` and `CBNZ`
: All T32 instructions can be relocated but this is discouraged because of the limited branch range of these instructions.

`LDC` and `LDC2`
: Only A32 instructions can be relocated.

`VLDR`
: Only A32 instructions can be relocated.

The assembler emits a relocation for these instructions if the label used meets any of the following requirements, as appropriate for the instruction type:

- The label is `WEAK`.
- The label is not in the same `AREA`.
- The label is external to the object (`IMPORT` or `EXTERN`).

For `B`, `BL`, and `BX` instructions, the assembler emits a relocation also if:

- The label is a function.
- The label is exported using `EXPORT` or `GLOBAL`.

Note

You can use the `RELOC` directive to control the relocation at a finer level, but this requires knowledge of the ABI.

Example

```
IMPORT sym   ; sym is an external symbol
DCW sym      ; Because DCW only outputs 16 bits, only the lower
             ; 16 bits of the address of sym are inserted at
             ; link-time.
```

Related references

21.6 AREA on page 21-1650.
21.27 EXPORT or GLOBAL on page 21-1673.
21.45 IMPORT and EXTERN on page 21-1693.
21.58 REQUIRE on page 21-1711.

21.57 RELOC on page 21-1710.
21.15 DCB on page 21-1661.
21.16 DCD and DCDU on page 21-1662.
21.22 DCW and DCWU on page 21-1668.
13.51 LDR (PC-relative) on page 13-411.
13.10 ADR (PC-relative) on page 13-349.
13.79 PLD, PLDW, and PLI on page 13-453.
13.15 B on page 13-359.
13.24 CBZ and CBNZ on page 13-373.
13.48 LDC and LDC2 on page 13-405.
14.52 VLDR on page 14-659.

Related information

ELF for the Arm Architecture.

6.26 Symbol versions

The Arm linker conforms to the Base Platform ABI for the Arm Architecture (BPABI) and supports the GNU-extended symbol versioning model.

To add a symbol version to an existing symbol, you must define a version symbol at the same address. A version symbol is of the form:

- *name@ver* if *ver* is a non default version of *name*.
- *name@@ver* if *ver* is the default version of *name*.

The version symbols must be enclosed in vertical bars.

For example, to define a default version:

```
|my_versioned_symbol@@ver2|   ; Default version
my_asm_function PROC
                ...
                BX lr
                ENDP
```

To define a non default version:

```
|my_versioned_symbol@ver1|    ; Non default version
my_old_asm_function     PROC
                        ...
                        BX lr
                        ENDP
```

Related information

Base Platform ABI for the Arm Architecture.

Accessing and managing symbols with armlink.

6.27 Frame directives

Frame directives provide information in object files that enables debugging and profiling of assembly language functions.

You must use frame directives to describe the way that your code uses the stack if you want to be able to do either of the following:

- Debug your application using stack unwinding.
- Use either flat or call-graph profiling.

The assembler uses frame directives to insert DWARF debug frame information into the object file in ELF format that it produces. This information is required by a debugger for stack unwinding and for profiling.

Be aware of the following:

- Frame directives do not affect the code produced by the assembler.
- The assembler does not validate the information in frame directives against the instructions emitted.

Related concepts

6.28 Exception tables and Unwind tables on page 6-138.

Related references

21.3 About frame directives on page 21-1646.

Related information

Procedure Call Standard for the Arm Architecture.

6.28 Exception tables and Unwind tables

You use FRAME directives to enable the assembler to generate *unwind* tables.

Note

Not supported for AArch64 state.

Exception tables are necessary to handle exceptions thrown by functions in high-level languages such as C++. Unwind tables contain debug frame information which is also necessary for the handling of such exceptions. An exception can only propagate through a function with an unwind table.

An assembly language function is code enclosed by either PROC and ENDP or FUNC and ENDFUNC directives. Functions written in C++ have unwind information by default. However, for assembly language functions that are called from C++ code, you must ensure that there are exception tables and unwind tables to enable the exceptions to propagate through them.

An exception cannot propagate through a function with a *nounwind* table. The exception handling runtime environment terminates the program if it encounters a nounwind table during exception processing.

The assembler can generate nounwind table entries for all functions and non-functions. The assembler can generate an unwind table for a function only if the function contains sufficient FRAME directives to describe the use of the stack within the function. To be able to create an unwind table for a function, each POP or PUSH instruction must be followed by a FRAME POP or FRAME PUSH directive respectively. Functions must conform to the conditions set out in the *Exception Handling ABI for the Arm® Architecture* (EHABI), section 9.1 *Constraints on Use*. If the assembler cannot generate an unwind table it generates a nounwind table.

Related concepts

6.27 Frame directives on page 6-137.

Related references

21.3 About frame directives on page 21-1646.
11.26 --exceptions, --no_exceptions on page 11-254.
11.27 --exceptions_unwind, --no_exceptions_unwind on page 11-255.
21.39 FRAME UNWIND ON on page 21-1686.
21.40 FRAME UNWIND OFF on page 21-1687.
21.41 FUNCTION or PROC on page 21-1688.
21.24 ENDFUNC or ENDP on page 21-1670.

Related information

Exception Handling ABI for the Arm Architecture.

Chapter 7
Condition Codes

Describes condition codes and conditional execution of A64, A32, and T32 code.

It contains the following sections:

7.1 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. Few A64 instructions can be conditionally executed.

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

7.2 Conditional execution in A32 code on page 7-141.
7.3 Conditional execution in T32 code on page 7-142.

Related references

7.12 Condition code suffixes and related flags on page 7-151.
7.6 Updates to the condition flags in A32/T32 code on page 7-145.
7.7 Updates to the condition flags in A64 code on page 7-146.

7.2 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

7.3 Conditional execution in T32 code on page 7-142.

7.3 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

7.2 Conditional execution in A32 code on page 7-141.

Related references

13.45 IT on page 13-399.

13.24 CBZ and CBNZ on page 13-373.

7.4 Conditional execution in A64 code

In the A64 instruction set, there are a few instructions that are truly conditional. Truly conditional means that when the condition is false, the instruction advances the program counter but has no other effect.

The conditional branch, `B.cond` is a truly conditional instruction. The condition code is appended to the instruction with a '.' delimiter, for example `B.EQ`.

There are other truly conditional branch instructions that execute depending on the value of the Zero condition flag. You cannot append any condition code suffix to them. These instructions are:

- `CBNZ`.
- `CBZ`.
- `TBNZ`.
- `TBZ`.

There are a few A64 instructions that are unconditionally executed but use the condition code as a source operand. These instructions always execute but the operation depends on the value of the condition code. These instructions can be categorized as:

- Conditional data processing instructions, for example `CSEL`.
- Conditional comparison instructions, `CCMN` and `CCMP`.

In these instructions, you specify the condition code in the final operand position, for example `CSEL Wd,Wm,Wn,NE`.

Related concepts

7.3 Conditional execution in T32 code on page 7-142.
7.2 Conditional execution in A32 code on page 7-141.

7.5 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

7.6 Updates to the condition flags in A32/T32 code on page 7-145.
7.7 Updates to the condition flags in A64 code on page 7-146.
7.12 Condition code suffixes and related flags on page 7-151.

7.6 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

7.1 Conditional instructions on page 7-140.

Related references

7.5 Condition flags on page 7-144.

7.7 Updates to the condition flags in A64 code on page 7-146.

7.12 Condition code suffixes and related flags on page 7-151.

Chapter 13 A32 and T32 Instructions on page 13-327.

7.7 Updates to the condition flags in A64 code

In AArch64 state, the N, Z, C, and V condition flags are held in the NZCV system register, which is part of the process state. You can access the flags using the `MSR` and `MRS` instructions.

Note

An instruction updates the condition flags only if the S suffix is specified, except the instructions `CMP`, `CMN`, `CCMP`, `CCMN`, and `TST`, which always update the condition flags. The instruction also determines which flags get updated. If a conditional instruction does not execute, it does not affect the flags.

Example

This example shows the read-modify-write procedure to change some of the condition flags in A64 code.

```
MRS  x1, NZCV          ; copy N, Z, C, and V flags into general-purpose x1
MOV  x2, #0x30000000
BIC  x1,x1,x2          ; clears the C and V flags (bits 29,28)
ORR  x1,x1,#0xC0000000 ; sets the N and Z flags (bits 31,30)
MSR  NZCV, x1          ; copy x1 back into NZCV register to update the condition flags
```

Related concepts

7.1 Conditional instructions on page 7-140.

Related references

7.5 Condition flags on page 7-144.

7.6 Updates to the condition flags in A32/T32 code on page 7-145.

7.12 Condition code suffixes and related flags on page 7-151.

7.8 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
```

All A64 floating-point comparison instructions can update the condition flags. These instructions update the flags directly in the NZCV register.

Related concepts

6.20 The Read-Modify-Write operation on page 6-128.
7.9 Carry flag on page 7-148.
7.10 Overflow flag on page 7-149.

Related references

7.7 Updates to the condition flags in A64 code on page 7-146.
15.4 VCMP, VCMPE on page 15-755.
14.72 VMRS on page 14-679.
15.27 VMRS (floating-point) on page 15-778.

Related information

Arm Architecture Reference Manual.

7.9 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.

In A64 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` and the negate instructions `NEGS` and `NGCS`, C is set to 0 if the subtraction produced a borrow (that is, an unsigned underflow), and to 1 otherwise.
- For the integer and floating-point conditional compare instructions `CCMP`, `CCMN`, `FCCMP`, and `FCCMPE`, C and the other condition flags are set either to the result of the comparison, or directly from an immediate value.
- For the floating-point compare instructions, `FCMP` and `FCMPE`, C and the other condition flags are set to the result of the comparison.
- For other instructions, C is normally left unchanged, but see the individual instruction descriptions for any special cases.

Related concepts

7.10 Overflow flag on page 7-149.

Related references

3.7 Predeclared core register names in AArch32 state on page 3-71.
4.5 Predeclared core register names in AArch64 state on page 4-85.
7.12 Condition code suffixes and related flags on page 7-151.
7.6 Updates to the condition flags in A32/T32 code on page 7-145.
7.7 Updates to the condition flags in A64 code on page 7-146.

7.10 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}.

In A64 instructions that use the 64-bit X registers, overflow occurs if the result of the operation is greater than or equal to 2^{63}, or less than -2^{63}.

In A64 instructions that use the 32-bit W registers, overflow occurs if the result of the operation is greater than or equal to 2^{31}, or less than -2^{31}.

Related concepts

7.9 Carry flag on page 7-148.

Related references

3.7 Predeclared core register names in AArch32 state on page 3-71.
7.6 Updates to the condition flags in A32/T32 code on page 7-145.
7.7 Updates to the condition flags in A64 code on page 7-146.

7.11 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 7-1 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 concepts

9.8 Conditional execution of A32/T32 Advanced SIMD instructions on page 9-192.

10.8 Conditional execution of A32/T32 floating-point instructions on page 10-215.

Related references

7.13 Comparison of condition code meanings in integer and floating-point code on page 7-152.

13.45 IT on page 13-399.

14.72 VMRS on page 14-679.

15.27 VMRS (floating-point) on page 15-778.

7.12 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 7-2 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 and in A64 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

7.1 Conditional instructions on page 7-140.

Related references

7.5 Condition flags on page 7-144.

7.13 Comparison of condition code meanings in integer and floating-point code on page 7-152.

7.6 Updates to the condition flags in A32/T32 code on page 7-145.

7.7 Updates to the condition flags in A64 code on page 7-146.

Chapter 13 A32 and T32 Instructions on page 13-327.

7.13 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 7-3 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

7.1 Conditional instructions on page 7-140.

Related references

7.12 Condition code suffixes and related flags on page 7-151.
7.6 Updates to the condition flags in A32/T32 code on page 7-145.
7.7 Updates to the condition flags in A64 code on page 7-146.
15.4 VCMP, VCMPE on page 15-755.

14.72 VMRS on page 14-679.
15.27 VMRS (floating-point) on page 15-778.

Related information

Arm Architecture Reference Manual.

7.14 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

7.15 Example showing the benefits of conditional instructions in A32 and T32 code on page 7-155.

7.15 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 7-4 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 7-5 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

7.14 Benefits of using conditional execution in A32 and T32 code on page 7-154.
7.16 Optimization for execution speed on page 7-158.

Related references

13.45 IT on page 13-399.
7.12 Condition code suffixes and related flags on page 7-151.

Related information

Arm Architecture Reference Manual.

7.16 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 8
Using armasm

Describes how to use `armasm`.

It contains the following sections:

8.1 armasm command-line syntax

You can use a command line to invoke `armasm`. You must specify an input source file and you can specify various options.

The command for invoking the assembler is:

`armasm {`*`options`*`} `*`inputfile`*

where:

`options`
: are commands that instruct the assembler how to assemble the *`inputfile`*. You can invoke `armasm` with any combination of options separated by spaces. You can specify values for some options. To specify a value for an option, use either '`=`' (*`option=value`*) or a space character (*`option value`*).

`inputfile`
: is an assembly source file. It must contain UAL, pre-UAL A32 or T32, or A64 assembly language.

The assembler command line is case-insensitive, except in filenames and where specified. The assembler uses the same command-line ordering rules as the compiler. This means that if the command line contains options that conflict with each other, then the last option found always takes precedence.

8.2 Specify command-line options with an environment variable

The `ARMCOMPILER6_ASMOPT` environment variable can hold command-line options for the assembler.

The syntax is identical to the command-line syntax. The assembler reads the value of `ARMCOMPILER6_ASMOPT` and inserts it at the front of the command string. This means that options specified in `ARMCOMPILER6_ASMOPT` can be overridden by arguments on the command line.

Related concepts

8.1 armasm command-line syntax on page 8-160.

Related information

Toolchain environment variables.

8.3 Using stdin to input source code to the assembler

You can use stdin to pipe output from another program into armasm or to input source code directly on the command line. This is useful if you want to test a short piece of code without having to create a file for it.

To use `stdin` to pipe output from another program into `armasm`, invoke the program and the assembler using the pipe character (|). Use the minus character (-) as the source filename to instruct the assembler to take input from `stdin`. You must specify the output filename using the `-o` option. You can specify the command-line options you want to use. For example to pipe output from `fromelf`:

```
fromelf --disassemble A32input.o | armasm --cpu=8-A.32 -o A32output.o -
```

Note

The source code from `stdin` is stored in an internal cache that can hold up to 8 MB. You can increase this cache size using the `--maxcache` command-line option.

To use `stdin` to input source code directly on the command line:

Procedure

1. Invoke the assembler with the command-line options you want to use. Use the minus character (-) as the source filename to instruct the assembler to take input from `stdin`. You must specify the output filename using the `-o` option. For example:

   ```
   armasm --cpu=8-A.32 -o output.o -
   ```

2. Enter your input. For example:

   ```
           AREA     A32ex, CODE, READONLY
                                   ; Name this block of code A32ex
           ENTRY                   ; Mark first instruction to execute
   start
           MOV      r0, #10        ; Set up parameters
           MOV      r1, #3
           ADD      r0, r0, r1     ; r0 = r0 + r1
   stop
           MOV      r0, #0x18      ; angel_SWIreason_ReportException
           LDR      r1, =0x20026   ; ADP_Stopped_ApplicationExit
           SVC      #0x123456      ; A32 semihosting (formerly SWI)
           END                     ; Mark end of file
   ```

3. Terminate your input by entering:
   - `Ctrl+Z` then `Return` on Microsoft Windows systems.
   - `Ctrl+D` on Unix-based operating systems.

Related concepts

8.1 armasm command-line syntax on page 8-160.

Related references

11.44 --maxcache=n on page 11-272.

8.4 Built-in variables and constants

armasm defines built-in variables that hold information about, for example, the state of armasm, the command-line options used, and the target architecture or processor.

The following table lists the built-in variables defined by armasm:

Table 8-1 Built-in variables

{ARCHITECTURE}	Holds the name of the selected Arm architecture.
{AREANAME}	Holds the name of the current AREA.
{ARMASM_VERSION}	Holds an integer that increases with each version of armasm. The format of the version number is *Mmmuuxx* where: • *M* is the major version number, 6. • *mm* is the minor version number. • *uu* is the update number. • *xx* is reserved for Arm internal use. You can ignore this for the purposes of checking whether the current release is a specific version or within a range of versions. **Note** The built-in variable|ads$version| is deprecated.
|ads$version|	Has the same value as {ARMASM_VERSION}.
{CODESIZE}	Is a synonym for {CONFIG}.
{COMMANDLINE}	Holds the contents of the command line.
{CONFIG}	Has the value: • 64 if the assembler is assembling A64 code. • 32 if the assembler is assembling A32 code. • 16 if the assembler is assembling T32 code.
{CPU}	Holds the name of the selected processor. The value of {CPU} is derived from the value specified in the --cpu option on the command line.
{ENDIAN}	Has the value "big" if the assembler is in big-endian mode, or "little" if it is in little-endian mode.
{FPU}	Holds the name of the selected FPU. The default in AArch32 state is "FP-ARMv8". The default in AArch64 state is "A64".
{INPUTFILE}	Holds the name of the current source file.
{INTER}	Has the Boolean value True if --apcs=/inter is set. The default is {False}.
{LINENUM}	Holds an integer indicating the line number in the current source file.
{LINENUMUP}	When used in a macro, holds an integer indicating the line number of the current macro. The value is the same as {LINENUM} when used in a non-macro context.
{LINENUMUPPER}	When used in a macro, holds an integer indicating the line number of the top macro. The value is the same as {LINENUM} when used in a non-macro context.
{OPT}	Value of the currently-set listing option. You can use the OPT directive to save the current listing option, force a change in it, or restore its original value.
{PC} or .	Address of current instruction.

{PCSTOREOFFSET}	Is the offset between the address of the STR PC,[...] or STM Rb,{..., PC} instruction and the value of PC stored out. This varies depending on the processor or architecture specified.
{ROPI}	Has the Boolean value {True} if --apcs=/ropi is set. The default is {False}.
{RWPI}	Has the Boolean value {True} if --apcs=/rwpi is set. The default is {False}.
{VAR} or @	Current value of the storage area location counter.

You can use built-in variables in expressions or conditions in assembly source code. For example:

```
        IF {ARCHITECTURE} = "8-A"
```

They cannot be set using the SETA, SETL, or SETS directives.

The names of the built-in variables can be in uppercase, lowercase, or mixed, for example:

```
        IF {CpU} = "Generic ARM"
```

Note

All built-in string variables contain case-sensitive values. Relational operations on these built-in variables do not match with strings that contain an incorrect case. Use the command-line options --cpu and --fpu to determine valid values for {CPU}, {ARCHITECTURE}, and {FPU}.

The assembler defines the built-in Boolean constants TRUE and FALSE.

Table 8-2 Built-in Boolean constants

{FALSE}	Logical constant false.
{TRUE}	Logical constant true.

The following table lists the target processor-related built-in variables that are predefined by the assembler. Where the value field is empty, the symbol is a Boolean value and the meaning column describes when its value is {TRUE}.

Table 8-3 Predefined macros

Name	Value	Meaning
{TARGET_ARCH_AARCH32}	boolean	{TRUE} when assembling for AArch32 state. {FALSE} when assembling for AArch64 state.
{TARGET_ARCH_AARCH64}	boolean	{TRUE} when assembling for AArch64 state. {FALSE} when assembling for AArch32 state.
{TARGET_ARCH_ARM}	*num*	The number of the A32 base architecture of the target processor irrespective of whether the assembler is assembling for A32 or T32. The value is defined as zero when assembling for A64, and eight when assembling for A32/T32.
{TARGET_ARCH_THUMB}	*num*	The number of the T32 base architecture of the target processor irrespective of whether the assembler is assembling for A32 or T32. The value is defined as zero when assembling for A64, and five when assembling for A32/T32.

Table 8-3 Predefined macros (continued)

Name	Value	Meaning
{TARGET_ARCH_*XX*}	–	*XX* represents the target architecture and its value depends on the target processor: For the Armv8 architecture: • If you specify the assembler option `--cpu=8-A.32` or `--cpu=8-A.64` then {TARGET_ARCH_8_A} is defined. • If you specify the assembler option `--cpu=8.1-A.32` or `--cpu=8.1-A.64` then {TARGET_ARCH_8_1_A} is defined. For the Armv7 architecture, if you specify `--cpu=Cortex-A8`, for example, then {TARGET_ARCH_7_A} is defined.
{TARGET_FEATURE_EXTENSION_REGISTER_COUNT}	*num*	The number of 64-bit extension registers available in Advanced SIMD or floating-point.
{TARGET_FEATURE_CLZ}	–	If the target processor supports the `CLZ` instruction.
{TARGET_FEATURE_CRYPTOGRAPHY}	–	If the target processor has cryptographic instructions.
{TARGET_FEATURE_DIVIDE}	–	If the target processor supports the hardware divide instructions `SDIV` and `UDIV`.
{TARGET_FEATURE_DOUBLEWORD}	–	If the target processor supports doubleword load and store instructions, for example the A32 and T32 instructions `LDRD` and `STRD` (except the Armv6-M architecture).
{TARGET_FEATURE_DSPMUL}	–	If the DSP-enhanced multiplier (for example the `SMLA`*xy* instruction) is available.
{TARGET_FEATURE_MULTIPLY}	–	If the target processor supports long multiply instructions, for example the A32 and T32 instructions `SMULL`, `SMLAL`, `UMULL`, and `UMLAL` (that is, all architectures except the Armv6-M architecture).
{TARGET_FEATURE_MULTIPROCESSING}	–	If assembling for a target processor with Multiprocessing Extensions.
{TARGET_FEATURE_NEON}	–	If the target processor has Advanced SIMD.
{TARGET_FEATURE_NEON_FP16}	–	If the target processor has Advanced SIMD with half-precision floating-point operations.
{TARGET_FEATURE_NEON_FP32}	–	If the target processor has Advanced SIMD with single-precision floating-point operations.
{TARGET_FEATURE_NEON_INTEGER}	–	If the target processor has Advanced SIMD with integer operations.
{TARGET_FEATURE_UNALIGNED}	–	If the target processor has support for unaligned accesses (all architectures except the Armv6-M architecture).
{TARGET_FPU_SOFTVFP}	–	If assembling with the option `--fpu=SoftVFP`.
{TARGET_FPU_SOFTVFP_VFP}	–	If assembling for a target processor with SoftVFP and floating-point hardware, for example `--fpu=SoftVFP+FP-ARMv8`.
{TARGET_FPU_VFP}	–	If assembling for a target processor with floating-point hardware, without using SoftVFP, for example `--fpu=FP-ARMv8`.
{TARGET_FPU_VFPV2}	–	If assembling for a target processor with VFPv2.
{TARGET_FPU_VFPV3}	–	If assembling for a target processor with VFPv3.
{TARGET_FPU_VFPV4}	–	If assembling for a target processor with VFPv4.

Table 8-3 Predefined macros (continued)

Name	Value	Meaning
{TARGET_PROFILE_A}	–	If assembling for a Cortex®-A profile processor, for example, if you specify the assembler option `--cpu=7-A`.
{TARGET_PROFILE_M}	–	If assembling for a Cortex-M profile processor, for example, if you specify the assembler option `--cpu=7-M`.
{TARGET_PROFILE_R}	–	If assembling for a Cortex-R profile processor, for example, if you specify the assembler option `--cpu=7-R`.

Related concepts

8.5 Identifying versions of armasm in source code on page 8-167.

Related references

11.13 --cpu=name on page 11-239.
11.32 --fpu=name on page 11-260.

8.5 Identifying versions of armasm in source code

The assembler defines the built-in variable `ARMASM_VERSION` to hold the version number of the assembler.

You can use it as follows:

```
IF ( {ARMASM_VERSION} / 100000) >= 6
  ; using armasm in Arm Compiler 6
ELIF ( {ARMASM_VERSION} / 1000000) = 5
  ; using armasm in Arm Compiler 5
ELSE
  ; using armasm in Arm Compiler 4.1 or earlier
ENDIF
```

Note

The built-in variable `|ads$version|` is deprecated.

Related references

8.4 Built-in variables and constants on page 8-163.

8.6 Diagnostic messages

The assembler can provide extra error, warning, and remark diagnostic messages in addition to the default ones.

By default, these additional diagnostic messages are not displayed. However, you can enable them using the command-line options `--diag_error`, `--diag_warning`, and `--diag_remark`.

Related concepts

8.7 Interlocks diagnostics on page 8-169.
8.8 Automatic IT block generation in T32 code on page 8-170.
8.9 T32 branch target alignment on page 8-171.
8.10 T32 code size diagnostics on page 8-172.
8.11 A32 and T32 instruction portability diagnostics on page 8-173.
8.12 T32 instruction width diagnostics on page 8-174.
8.13 Two pass assembler diagnostics on page 8-175.

Related references

11.17 --diag_error=tag[,tag,...] on page 11-245.

8.7 Interlocks diagnostics

`armasm` can report warning messages about possible interlocks in your code caused by the pipeline of the processor chosen by the `--cpu` option.

To do this, use the `--diag_warning 1563` command-line option when invoking `armasm`.

Note

- `armasm` does not have an accurate model of the target processor, so these messages are not reliable when used with a multi-issue processor such as Cortex-A8.
- Interlocks diagnostics apply to A32 and T32 code, but not to A64 code.

Related concepts

8.8 Automatic IT block generation in T32 code on page 8-170.

8.9 T32 branch target alignment on page 8-171.

8.12 T32 instruction width diagnostics on page 8-174.

8.6 Diagnostic messages on page 8-168.

Related references

11.21 --diag_warning=tag[,tag,...] on page 11-249.

8.8 Automatic IT block generation in T32 code

`armasm` can automatically insert an `IT` block for conditional instructions in T32 code, without requiring the use of explicit `IT` instructions.

If you write the following code:

```
        AREA x, CODE
        THUMB
        MOVNE   r0,r1
        NOP
        IT      NE
        MOVNE   r0,r1
        END
```

`armasm` generates the following instructions:

```
        IT      NE
        MOVNE   r0,r1
        NOP
        IT      NE
        MOVNE   r0,r1
```

You can receive warning messages about the automatic generation of `IT` blocks when assembling T32 code. To do this, use the `armasm --diag_warning 1763` command-line option when invoking `armasm`.

Related concepts

8.6 Diagnostic messages on page 8-168.

Related references

11.21 --diag_warning=tag[,tag,...] on page 11-249.

8.9 T32 branch target alignment

`armasm` can issue warnings about non word-aligned branch targets in T32 code.

On some processors, non word-aligned T32 instructions sometimes take one or more additional cycles to execute in loops. This means that it can be an advantage to ensure that branch targets are word-aligned. To ensure `armasm` reports such warnings, use the `--diag_warning 1604` command-line option when invoking it.

Related concepts

8.6 Diagnostic messages on page 8-168.

Related references

11.21 --diag_warning=tag[,tag,...] on page 11-249.

8.10 T32 code size diagnostics

In T32 code, some instructions, for example a branch or `LDR` (PC-relative), can be encoded as either a 32-bit or 16-bit instruction. `armasm` chooses the size of the instruction encoding.

`armasm` can issue a warning when it assembles a T32 instruction to a 32-bit encoding when it could have used a 16-bit encoding.

To enable this warning, use the `--diag_warning 1813` command-line option when invoking `armasm`.

Related concepts

8.17 Instruction width selection in T32 code on page 8-180.

2.2 A32 and T32 instruction sets on page 2-58.

8.6 Diagnostic messages on page 8-168.

Related references

11.21 --diag_warning=tag[,tag,...] on page 11-249.

8.11 A32 and T32 instruction portability diagnostics

`armasm` can issue warnings about instructions that cannot assemble to both A32 and T32 code.

There are a few UAL instructions that can assemble as either A32 code or T32 code, but not both. You can identify these instructions in the source code using the `--diag_warning 1812` command-line option when invoking `armasm`.

It warns for any instruction that cannot be assembled in the other instruction set. This is only a hint, and other factors, like relocation availability or target distance might affect the accuracy of the message.

Related concepts

2.2 A32 and T32 instruction sets on page 2-58.

8.6 Diagnostic messages on page 8-168.

Related references

11.21 --diag_warning=tag[,tag,...] on page 11-249.

8.12 T32 instruction width diagnostics

`armasm` can issue a warning when it assembles a T32 instruction to a 32-bit encoding when it could have used a 16-bit encoding.

If you use the `.W` specifier, the instruction is encoded in 32 bits even if it could be encoded in 16 bits. You can use a diagnostic warning to detect when a branch instruction could have been encoded in 16 bits, but has been encoded in 32 bits. To do this, use the `--diag_warning 1607` command-line option when invoking `armasm`.

Note

This diagnostic does not produce a warning for relocated branch instructions, because the final address is not known. The linker might even insert a veneer, if the branch is out of range for a 32-bit instruction.

Related concepts

8.6 Diagnostic messages on page 8-168.

Related references

11.21 --diag_warning=tag[,tag,...] on page 11-249.

8.13 Two pass assembler diagnostics

`armasm` can issue a warning about code that might not be identical in both assembler passes.

`armasm` is a two pass assembler and the input code that the assembler reads must be identical in both passes. If a symbol is defined after the `:DEF:` test for that symbol, then the code read in pass one might be different from the code read in pass two. `armasm` can warn in this situation.

To do this, use the `--diag_warning 1907` command-line option when invoking `armasm`.

Example

The following example shows that the symbol `foo` is defined after the `:DEF: foo` test.

```
    AREA x,CODE
    [ :DEF: foo
    ]
foo MOV r3, r4
    END
```

Assembling this code with `--diag_warning 1907` generates the message:

```
Warning A1907W: Test for this symbol has been seen and may cause failure in the second pass.
```

Related concepts

8.8 Automatic IT block generation in T32 code on page 8-170.

8.9 T32 branch target alignment on page 8-171.

8.12 T32 instruction width diagnostics on page 8-174.

8.6 Diagnostic messages on page 8-168.

1.3 How the assembler works on page 1-49.

Related references

11.21 --diag_warning=tag[,tag,...] on page 11-249.

1.4 Directives that can be omitted in pass 2 of the assembler on page 1-51.

8.14 Using the C preprocessor

armasm can invoke armclang to preprocess an assembly language source file before assembling it. This allows you to use C preprocessor commands in assembly source code.

If you do this, you must use the --cpreproc command-line option together with the --cpreproc_opts command-line option when invoking the assembler. This causes armasm to call armclang to preprocess the file before assembling it.

Note

As a minimum, you must specify the armclang --target option and either the -mcpu or -march option with --cpreproc_opts.

armasm looks for the armclang binary in the same directory as the armasm binary. If it does not find the binary, it expects it to be on the PATH.

armasm passes the following options by default to armclang if present on the command line:

- Basic pre-processor configuration options, such as -E.
- User specified include directories, -I directives.
- User specified licensing options, such as --site_license.
- Anything specified in --cpreproc_opts.

Some of the options that armasm passes to armclang are converted to the armclang equivalent beforehand. These are shown in the following table:

Table 8-4 armclang equivalent command-line options

armasm	armclang
--thumb	-mthumb
--arm	-marm
-i	-I

armasm correctly interprets the preprocessed #line commands. It can generate error messages and debug_line tables using the information in the #line commands.

Preprocessing an assembly language source file

The following example shows the command you write to preprocess and assemble a file, source.S. The example also passes the compiler options to define a macro called RELEASE, and to undefine a macro called ALPHA.

```
armasm --cpu=cortex-m3 --cpreproc --cpreproc_opts=--target=arm-arm-none-eabi,-mcpu=cortex-
a9,-D,RELEASE,-U,ALPHA source.S
```

Preprocessing an assembly language source file manually

Alternatively, you must manually call armclang to preprocess the file before calling armasm. The following example shows the commands you write to manually preprocess and assemble a file, source.S:

```
armclang --target=arm-arm-none-eabi -mcpu=cortex-m3 -E source.S > preprocessed.S
armasm --cpu=cortex-m3 preprocessed.S
```

In this example, the preprocessor outputs a file called preprocessed.S, and armasm assembles it.

Related references

11.10 --cpreproc on page 11-236.
11.11 --cpreproc_opts=option[,option,...] on page 11-237.

Related information

Specifying a target architecture, processor, and instruction set.
-march armclang option.
-mcpu armclang option.
--target armclang option.

8.15 Address alignment in A32/T32 code

In Armv7-A and Armv7-R, the A bit in the *System Control Register* (SCTLR) controls whether alignment checking is enabled or disabled. In Armv7-M, the `UNALIGN_TRP` bit, bit 3, in the *Configuration and Control Register* (CCR) controls this.

If alignment checking is enabled, all unaligned word and halfword transfers cause an alignment exception. If disabled, unaligned accesses are permitted for the `LDR`, `LDRH`, `STR`, `STRH`, `LDRSH`, `LDRT`, `STRT`, `LDRSHT`, `LDRHT`, `STRHT`, and `TBH` instructions. Other data-accessing instructions always cause an alignment exception for unaligned data.

For `STRD` and `LDRD`, the specified address must be word-aligned.

If all your data accesses are aligned, you can use the `--no_unaligned_access` command-line option to declare that the output object was not permitted to make unaligned access. The linker can then avoid linking in any library functions that support unaligned access if all input objects declare that they were not permitted to use unaligned accesses.

Related references

11.60 --unaligned_access, --no_unaligned_access on page 11-288.

8.16 Address alignment in A64 code

If alignment checking is not enabled, then unaligned accesses are permitted for all load and store instructions other than exclusive load, exclusive store, load acquire, and store release instructions. If alignment checking is enabled, then unaligned accesses are not permitted.

This means all load and store instructions must use addresses that are aligned to the size of the data being accessed. In other words, addresses for 8-byte transfers must be 8-byte aligned, addresses for 4-byte transfers are 4-byte word aligned, and addresses for 2-byte transfers are 2-byte aligned. Unaligned accesses cause an alignment exception.

For any memory access, if the stack pointer is used as the base register, then it must be quadword aligned. Otherwise it generates a stack alignment exception.

If all your data accesses are aligned, you can use the `--no_unaligned_access` command-line option to declare that the output object was not permitted to make unaligned access. The linker can then avoid linking in any library functions that support unaligned access if all input objects declare that they were not permitted to use unaligned accesses.

8.17 Instruction width selection in T32 code

Some T32 instructions can have either a 16-bit encoding or a 32-bit encoding.

If you do not specify the instruction size, by default:

- For forward reference `LDR`, `ADR`, and `B` instructions, `armasm` always generates a 16-bit instruction, even if that results in failure for a target that could be reached using a 32-bit instruction.
- For external reference `LDR` and `B` instructions, `armasm` always generates a 32-bit instruction.
- In all other cases, `armasm` generates the smallest size encoding that can be output.

If you want to override this behavior, you can use the `.W` or `.N` width specifier to ensure a particular instruction size. `armasm` faults if it cannot generate an instruction with the specified width.

The `.W` specifier is ignored when assembling to A32 code, so you can safely use this specifier in code that might assemble to either A32 or T32 code. However, the `.N` specifier is faulted when assembling to A32 code.

Related concepts

8.10 T32 code size diagnostics on page 8-172.

Related references

13.2 Instruction width specifiers on page 13-337.

Chapter 9
Advanced SIMD Programming

Describes Advanced SIMD assembly language programming.

It contains the following sections:

9.1 Architecture support for Advanced SIMD

Advanced SIMD is an optional extension to the Armv8 and Armv7 architectures.

All Advanced SIMD instructions are available on systems that support Advanced SIMD. In A32, some of these instructions are also available on systems that implement the floating-point extension without Advanced SIMD. These are called shared instructions.

In AArch32 state, the Advanced SIMD register bank consists of thirty-two 64-bit registers, and smaller registers are packed into larger ones, as in Armv7.

In AArch64 state, the Advanced SIMD register bank includes thirty-two 128-bit registers and has a new register packing model.

Note

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

Advanced SIMD instructions in A64 are closely based on VFPv4 and A32, but with new instruction mnemonics and some functional enhancements.

Related information

Floating-point support.

Further reading.

9.2 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`.

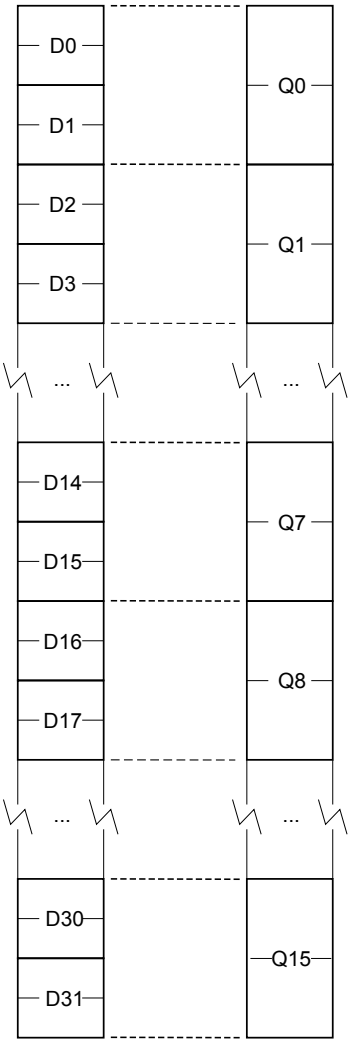


Figure 9-1 Extension register bank for Advanced SIMD in AArch32 state

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

9.3 Extension register bank mapping for Advanced SIMD in AArch64 state on page 9-185.
10.4 Views of the floating-point extension register bank in AArch32 state on page 10-211.
9.4 Views of the Advanced SIMD register bank in AArch32 state on page 9-187.
10.4 Views of the floating-point extension register bank in AArch32 state on page 10-211.

9.3 Extension register bank mapping for Advanced SIMD in AArch64 state

The extension register bank is a collection of registers that can be accessed as 8-bit, 16-bit, 32-bit, 64-bit, or 128-bit.

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.

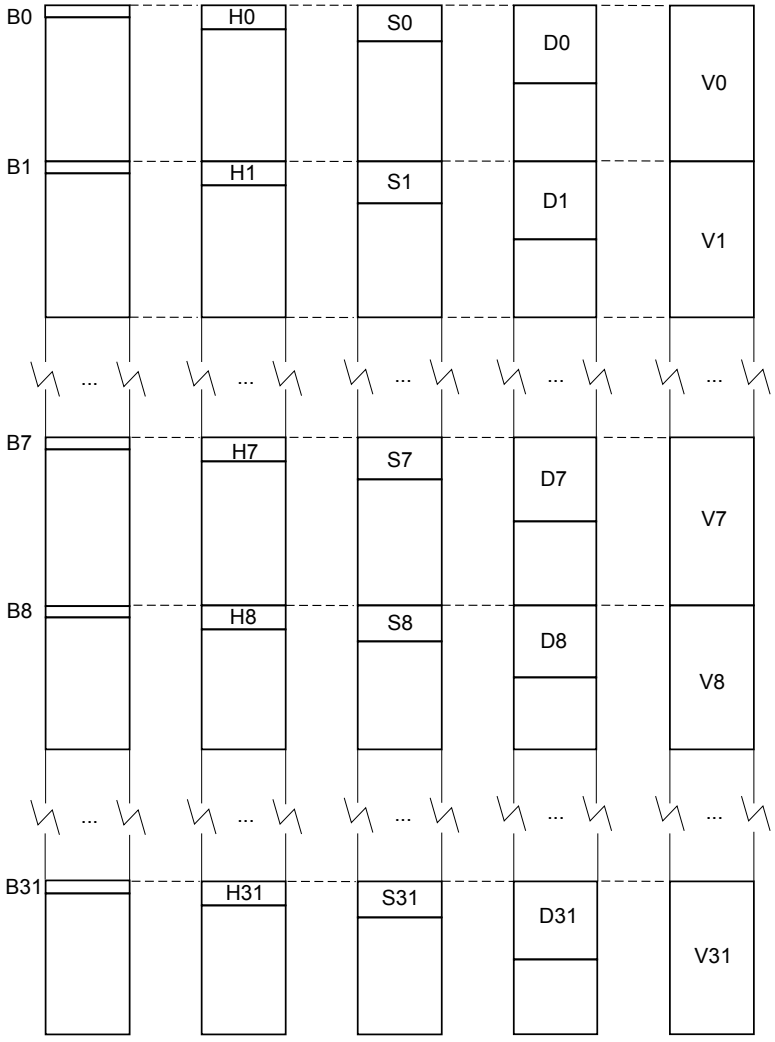


Figure 9-2 Extension register bank for Advanced SIMD in AArch64 state

The mapping between the registers is as follows:

- `D<n>` maps to the least significant half of `V<n>`
- `S<n>` maps to the least significant half of `D<n>`
- `H<n>` maps to the least significant half of `S<n>`
- `B<n>` maps to the least significant half of `H<n>`.

For example, you can access the least significant half of the elements of a vector in `V7` by referring to `D7`.

Registers `Q0-Q31` map directly to registers `V0-V31`.

Related concepts

9.2 Extension register bank mapping for Advanced SIMD in AArch32 state on page 9-183.

10.4 Views of the floating-point extension register bank in AArch32 state on page 10-211.
9.4 Views of the Advanced SIMD register bank in AArch32 state on page 9-187.
10.4 Views of the floating-point extension register bank in AArch32 state on page 10-211.

9.4 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

9.5 Views of the Advanced SIMD register bank in AArch64 state on page 9-188.

9.2 Extension register bank mapping for Advanced SIMD in AArch32 state on page 9-183.

10.4 Views of the floating-point extension register bank in AArch32 state on page 10-211.

9.5 Views of the Advanced SIMD register bank in AArch64 state

Advanced SIMD can have different views of the extension register bank in AArch64 state.

It can view the extension register bank as:

- Thirty-two 128-bit registers `V0-V31`.
- Thirty-two 64-bit registers `D0-D31`.
- Thirty-two 32-bit registers `S0-S31`.
- Thirty-two 16-bit registers `H0-H31`.
- Thirty-two 8-bit registers `B0-B31`.
- A combination of registers from these views.

Related concepts

9.4 Views of the Advanced SIMD register bank in AArch32 state on page 9-187.

9.2 Extension register bank mapping for Advanced SIMD in AArch32 state on page 9-183.

10.4 Views of the floating-point extension register bank in AArch32 state on page 10-211.

9.6 Differences between A32/T32 and A64 Advanced SIMD instruction syntax

The syntax and mnemonics of A64 Advanced SIMD instructions are based on those in A32/T32 but with some differences.

The following table describes the main differences.

Table 9-1 Differences in syntax and mnemonics between A32/T32 and A64 Advanced SIMD instructions

A32/T32	A64
All Advanced SIMD instruction mnemonics begin with `V`, for example `VMAX`.	The first letter of the instruction mnemonic indicates the data type of the instruction. For example, `SMAX`, `UMAX`, and `FMAX` mean signed, unsigned, and floating-point respectively. No suffix means the type is irrelevant and `P` means polynomial.
A mnemonic qualifier specifies the type and width of elements in a vector. For example, in the following instruction, `U32` means 32-bit unsigned integers: `VMAX.U32 Q0, Q1, Q2`	A register qualifier specifies the data width and the number of elements in the register. For example, in the following instruction `.4S` means 4 32-bit elements: `UMAX V0.4S, V1.4S, V2.4S`
The 128-bit vector registers are named Q0-Q15 and the 64-bit vector registers are named D0-D31.	All vector registers are named Vn , where n is a register number between 0 and 31. You only use one of the qualified register names Qn, Dn, Sn, Hn or Bn when referring to a scalar register, to indicate the number of significant bits.
You load a single element into one or more vector registers by appending an index to each register individually, for example: `VLD4.8 {D0[3], D1[3], D2[3], D3[3]}, [R0]`	You load a single element into one or more vector registers by appending the index to the register list, for example: `LD4 {V0.B, V1.B, V2.B, V3.B}[3], [X0]`
You can append a condition code to most Advanced SIMD instruction mnemonics to make them conditional.	A64 has no conditionally executed floating-point or Advanced SIMD instructions.
`L`, `W` and `N` suffixes indicate long, wide and narrow variants of Advanced SIMD data processing instructions. A32/T32 Advanced SIMD does not include vector narrowing or widening second part instructions.	`L`, `W` and `N` suffixes indicate long, wide and narrow variants of Advanced SIMD data processing instructions. You can additionally append a `2` to implement the second part of a narrowing or widening operation, for example: `UADDL2 V0.4S, V1.8H, V2.8H ; take input from 4 high-numbered lanes of V1 and V2`
A32/T32 Advanced SIMD does not include vector reduction instructions.	The `V` Advanced SIMD mnemonic suffix identifies vector reduction instructions, in which the operand is a vector and the result a scalar, for example: `ADDV S0, V1.4S`
The `P` mnemonic qualifier which indicates pairwise instructions is a prefix, for example, `VPADD`.	The `P` mnemonic qualifier is a suffix, for example `ADDP`.

Related concepts

9.10 Advanced SIMD data types in A32/T32 instructions on page 9-194.
9.8 Conditional execution of A32/T32 Advanced SIMD instructions on page 9-192.
9.15 Advanced SIMD scalars on page 9-199.
9.13 Normal, long, wide, and narrow Advanced SIMD instructions on page 9-197.
6.2 Syntax differences between UAL and A64 assembly language on page 6-103.

Related references

15.37 VSEL on page 15-788.
18.8 FCSEL on page 18-1155.

9.7 Load values to Advanced SIMD registers

To load a register with a floating-point immediate value, use `VMOV` in A32 or `FMOV` in A64. Both instructions exist in scalar and vector forms.

The A32 Advanced SIMD instructions `VMOV` and `VMVN` can also load integer immediates. The A64 Advanced SIMD instructions to load integer immediates are `MOVI` and `MVNI`.

You can load any 64-bit integer, single-precision, or double-precision floating-point value from a literal pool using the `VLDR` pseudo-instruction.

Related references

14.54 VLDR pseudo-instruction on page 14-661.
15.23 VMOV (floating-point) on page 15-774.
14.65 VMOV (immediate) on page 14-672.

9.8 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

7.2 Conditional execution in A32 code on page 7-141.

7.3 Conditional execution in T32 code on page 7-142.

Related references

7.13 Comparison of condition code meanings in integer and floating-point code on page 7-152.

7.11 Condition code suffixes on page 7-150.

9.9 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

9.18 Flush-to-zero mode in Advanced SIMD on page 9-202.

Related references

Chapter 9 Advanced SIMD Programming on page 9-181.

Related information

Arm Architecture Reference Manual.

Further reading.

9.10 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 9-2 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

9.10 Advanced SIMD data types in A32/T32 instructions on page 9-194.
9.11 Polynomial arithmetic over {0,1} on page 9-195.

9.11 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

9.10 Advanced SIMD data types in A32/T32 instructions on page 9-194.

9.12 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.

In A32/T32 Advanced SIMD instructions, the size of the elements in an Advanced SIMD vector is specified by a datatype suffix appended to the mnemonic. In A64 Advanced SIMD instructions, the size and number of the elements in an Advanced SIMD vector are specified by a suffix appended to the register.

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

9.15 Advanced SIMD scalars on page 9-199.

9.2 Extension register bank mapping for Advanced SIMD in AArch32 state on page 9-183.

9.16 Extended notation extension for Advanced SIMD in A32/T32 code on page 9-200.

9.10 Advanced SIMD data types in A32/T32 instructions on page 9-194.

9.13 Normal, long, wide, and narrow Advanced SIMD instructions on page 9-197.

9.13 Normal, long, wide, and narrow Advanced SIMD instructions

Many A32/T32 and A64 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 A32/T32 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

9.12 Advanced SIMD vectors on page 9-196.

9.14 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 9-3 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 in AArch32 or FPSR in AArch64) to indicate that saturation has occurred.

Saturating instructions are specified using a `Q` prefix. In A32/T32 Advanced SIMD instructions, this is inserted between the `V` and the instruction mnemonic, or between the S or U and the mnemonic in A64 Advanced SIMD instructions.

Related references

13.7 Saturating instructions on page 13-344.

9.15 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.

In A32/T32 Advanced SIMD instructions, 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`. In A64 Advanced SIMD instructions, you append the index to the element size specifier, so that `Vm.D[x]` is the *x*th doubleword element in vector `Vm`.

In A64 Advanced SIMD scalar instructions, you refer to registers using a name that indicates the number of significant bits. The names are `Bn`, `Hn`, `Sn`, or `Dn`, where *n* is the register number (0-31). The unused high bits are ignored on a read and set to zero on a write.

Other than A32/T32 Advanced SIMD multiply instructions, instructions that access scalars can access any element in the register bank.

A32/T32 Advanced SIMD multiply instructions only allow 16-bit or 32-bit scalars, and can only access the first 32 scalars in the register bank. That is, 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

9.12 Advanced SIMD vectors on page 9-196.

9.2 Extension register bank mapping for Advanced SIMD in AArch32 state on page 9-183.

9.16 Extended notation extension for Advanced SIMD in A32/T32 code

`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.

Note

Extended notation is not supported for A64 code.

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

9.12 Advanced SIMD vectors on page 9-196.

9.10 Advanced SIMD data types in A32/T32 instructions on page 9-194.

9.15 Advanced SIMD scalars on page 9-199.

Related references

21.56 QN, DN, and SN on page 21-1708.

9.17 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 concepts

6.20 The Read-Modify-Write operation on page 6-128.

Related information

Arm Architecture Reference Manual.

Further reading.

9.18 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

9.20 The effects of using flush-to-zero mode in Advanced SIMD on page 9-204.

Related references

9.19 When to use flush-to-zero mode in Advanced SIMD on page 9-203.
9.21 Advanced SIMD operations not affected by flush-to-zero mode on page 9-205.

9.19 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 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 in one of the following ways:

- In A32 code, by setting the FZ bit in the FPSCR to 1. You do this using the `VMRS` and `VMSR` instructions.
- In A64 code, by setting the FZ bit in the FPCR to 1. You do this using the `MRS` and `MSR` 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

9.18 Flush-to-zero mode in Advanced SIMD on page 9-202.

9.20 The effects of using flush-to-zero mode in Advanced SIMD on page 9-204.

9.20 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

9.18 Flush-to-zero mode in Advanced SIMD on page 9-202.

Related references

9.21 Advanced SIMD operations not affected by flush-to-zero mode on page 9-205.

9.21 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

9.18 Flush-to-zero mode in Advanced SIMD on page 9-202.

Related references

14.10 VABS on page 14-614.

15.2 VABS (floating-point) on page 15-753.

14.42 VDUP on page 14-646.

14.51 VLDM on page 14-658.

14.52 VLDR on page 14-659.

14.66 VMOV (register) on page 14-673.

14.67 VMOV (between two general-purpose registers and a 64-bit extension register) on page 14-674.

14.68 VMOV (between a general-purpose register and an Advanced SIMD scalar) on page 14-675.

14.134 VSWP on page 14-743.

Chapter 10
Floating-point Programming

Describes floating-point assembly language programming.

It contains the following sections:

10.1 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 supported in A32 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.

In AArch32 state, the register bank consists of thirty-two 64-bit registers, and smaller registers are packed into larger ones, as in Armv7 and earlier.

In AArch64 state, the register bank includes thirty-two 128-bit registers and has a new register packing model.

Floating point instructions in A64 are closely based on VFPv4 and A32, but with new instruction mnemonics and some functional enhancements.

Related information

Floating-point support.

Further reading.

10.2 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.

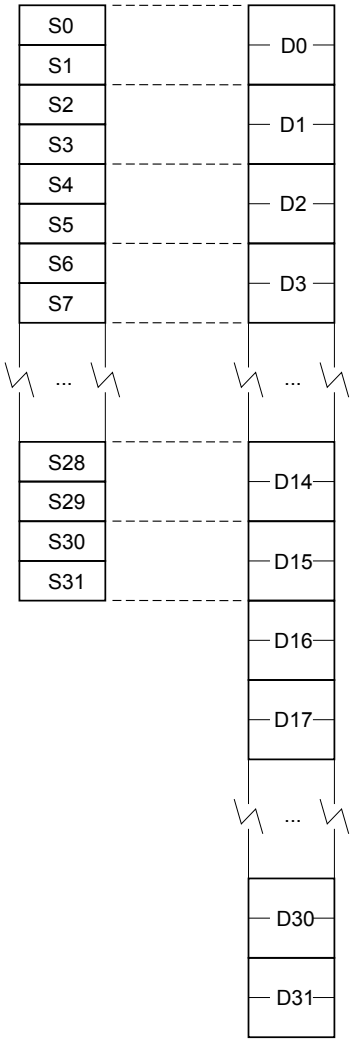


Figure 10-1 Extension register bank for floating-point in AArch32 state

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

10.4 Views of the floating-point extension register bank in AArch32 state on page 10-211.

10.3 Extension register bank mapping in AArch64 state

The extension register bank is a collection of registers that can be accessed as 16-bit, 32-bit, or 64-bit. 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.

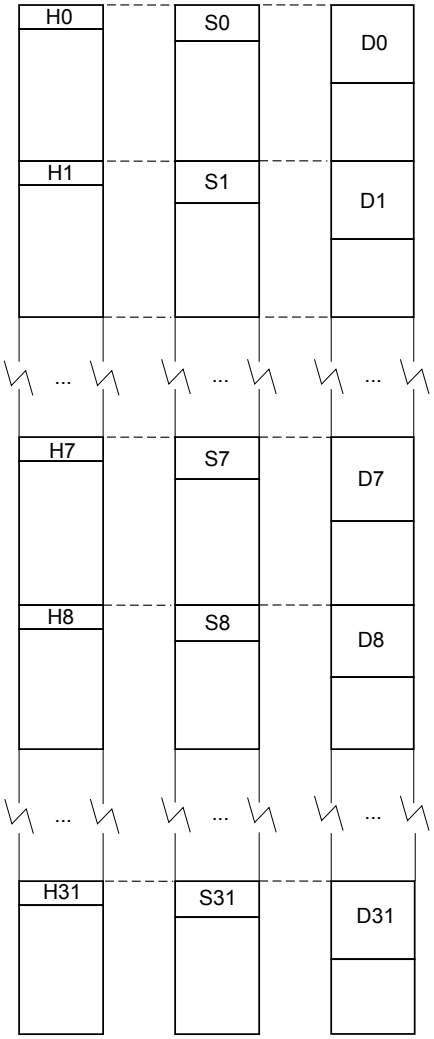


Figure 10-2 Extension register bank for floating-point in AArch64 state

The mapping between the registers is as follows:

- `S<n>` maps to the least significant half of `D<n>`
- `H<n>` maps to the least significant half of `S<n>`

For example, you can access the least significant half of register `D7` by referring to `S7`.

Related concepts

10.5 Views of the floating-point extension register bank in AArch64 state on page 10-212.

10.4 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

10.2 Extension register bank mapping for floating-point in AArch32 state on page 10-208.

10.5 Views of the floating-point extension register bank in AArch64 state

Floating-point can have different views of the extension register bank in AArch64 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`.
- Thirty-two 16-bit registers `H0-H31`.
- A combination of registers from these views.

Related concepts

10.3 Extension register bank mapping in AArch64 state on page 10-210.

10.6 Differences between A32/T32 and A64 floating-point instruction syntax

The syntax and mnemonics of A64 floating-point instructions are based on those in A32/T32 but with some differences.

The following table describes the main differences.

Table 10-1 Differences in syntax and mnemonics between A32/T32 and A64 floating-point instructions

A32/T32	A64
All floating-point instruction mnemonics begin with `V`, for example `VMAX`.	The first letter of the instruction mnemonic indicates the data type of the instruction. For example, `SMAX`, `UMAX`, and `FMAX` mean signed, unsigned, and floating-point respectively. No suffix means the type is irrelevant and `P` means polynomial.
A mnemonic qualifier specifies the type and width of elements in a vector. For example, in the following instruction, `U32` means 32-bit unsigned integers: `VMAX.U32 Q0, Q1, Q2`	A register qualifier specifies the data width and the number of elements in the register. For example, in the following instruction `.4S` means 4 32-bit elements: `UMAX V0.4S, V1.4S, V2.4S`
You can append a condition code to most floating-point instruction mnemonics to make them conditional.	A64 has no conditionally executed floating-point instructions.
The floating-point select instruction, `VSEL`, is unconditionally executed but uses a condition code as an operand. You append the condition code to the mnemonic, for example: `VSELEQ.F32 S1,S2,S3`	There are several floating-point instructions that use a condition code as an operand. You specify the condition code in the final operand position, for example: `FCSEL S1,S2,S3,EQ`
The `P` mnemonic qualifier which indicates pairwise instructions is a prefix, for example, `VPADD`.	The `P` mnemonic qualifier is a suffix, for example `ADDP`.

10.7 Load values to floating-point registers

To load a register with a floating-point immediate value, use `VMOV` in A32 or `FMOV` in A64. Both instructions exist in scalar and vector forms.

You can load any 64-bit integer, single-precision, or double-precision floating-point value from a literal pool using the `VLDR` pseudo-instruction.

Related references

15.17 VLDR pseudo-instruction (floating-point) on page 15-768.
15.23 VMOV (floating-point) on page 15-774.
18.31 FMOV (scalar, immediate) on page 18-1178.

10.8 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

7.2 Conditional execution in A32 code on page 7-141.

7.3 Conditional execution in T32 code on page 7-142.

Related references

7.13 Comparison of condition code meanings in integer and floating-point code on page 7-152.

7.11 Condition code suffixes on page 7-150.

10.9 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

10.13 Flush-to-zero mode in floating-point on page 10-220.

Related references

Chapter 15 Floating-point Instructions (32-bit) on page 15-749.

Related information

Arm Architecture Reference Manual.

Further reading.

10.10 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

9.11 Polynomial arithmetic over {0,1} on page 9-195.

10.11 Extended notation extension for floating-point in A32/T32 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.

Note

Extended notation is not supported for A64 code.

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

10.10 Floating-point data types in A32/T32 instructions on page 10-217.

Related references

21.56 QN, DN, and SN on page 21-1708.

10.12 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 concepts

6.20 The Read-Modify-Write operation on page 6-128.

Related information

Arm Architecture Reference Manual.

Further reading.

10.13 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

10.15 The effects of using flush-to-zero mode in floating-point on page 10-222.

Related references

10.14 When to use flush-to-zero mode in floating-point on page 10-221.

10.16 Floating-point operations not affected by flush-to-zero mode on page 10-223.

10.14 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 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 in one of the following ways:

- In A32 code, by setting the FZ bit in the FPSCR to 1. You do this using the `VMRS` and `VMSR` instructions.
- In A64 code, by setting the FZ bit in the FPCR to 1. You do this using the `MRS` and `MSR` 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

10.13 Flush-to-zero mode in floating-point on page 10-220.

10.15 The effects of using flush-to-zero mode in floating-point on page 10-222.

10.15 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

10.13 Flush-to-zero mode in floating-point on page 10-220.

Related references

10.16 Floating-point operations not affected by flush-to-zero mode on page 10-223.

10.16 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 Arm general-purpose registers (`VMOV`).

Related concepts

10.13 Flush-to-zero mode in floating-point on page 10-220.

Related references

15.2 VABS (floating-point) on page 15-753.

15.14 VLDM (floating-point) on page 15-765.

15.15 VLDR (floating-point) on page 15-766.

15.39 VSTM (floating-point) on page 15-790.

15.40 VSTR (floating-point) on page 15-791.

14.51 VLDM on page 14-658.

14.52 VLDR on page 14-659.

14.126 VSTM on page 14-733.

14.129 VSTR on page 14-738.

15.24 VMOV (between one general-purpose register and single precision floating-point register) on page 15-775.

14.67 VMOV (between two general-purpose registers and a 64-bit extension register) on page 14-674.

15.30 VNEG (floating-point) on page 15-781.

14.80 VNEG on page 14-687.

Chapter 11
armasm Command-line Options

Describes the `armasm` command-line syntax and command-line options.

It contains the following sections:

11.1 --16

Instructs `armasm` to interpret instructions as T32 instructions using the pre-UAL T32 syntax.

This option is equivalent to a `CODE16` directive at the head of the source file. Use the `--thumb` option to specify T32 instructions using the UAL syntax.

Note

Not supported for AArch64 state.

Related references

11.59 --thumb on page 11-287.

21.11 CODE16 directive on page 21-1657.

11.2 --32

A synonym for the `--arm` command-line option.

Note

Not supported for AArch64 state.

Related references

11.4 --arm on page 11-230.

11.3 --apcs=qualifier...qualifier

Controls interworking and position independence when generating code.

Syntax

`--apcs=qualifier...qualifier`

Where *qualifier...qualifier* denotes a list of qualifiers. There must be:

- At least one qualifier present.
- No spaces or commas separating individual qualifiers in the list.

Each instance of *qualifier* must be one of:

none
: Specifies that the input file does not use AAPCS. AAPCS registers are not set up. Other qualifiers are not permitted if you use `none`.

`/interwork, /nointerwork`
: For Armv7-A, `/interwork` specifies that the code in the input file can interwork between A32 and T32 safely.

  For Armv8-A, `/interwork` specifies that the code in the input file can interwork between A32 and T32 safely.

  The default is `/nointerwork`.

  `/nointerwork` is not supported for AArch64 state.

`/inter, /nointer`
: Are synonyms for `/interwork` and `/nointerwork`.

  `/inter` is not supported for AArch64 state.

`/ropi, /noropi`
: `/ropi` specifies that the code in the input file is *Read-Only Position-Independent* (ROPI). The default is `/noropi`.

`/pic, /nopic`
: Are synonyms for `/ropi` and `/noropi`.

`/rwpi, /norwpi`
: `/rwpi` specifies that the code in the input file is *Read-Write Position-Independent* (RWPI). The default is `/norwpi`.

`/pid, /nopid`
: Are synonyms for `/rwpi` and `/norwpi`.

`/fpic, /nofpic`
: `/fpic` specifies that the code in the input file is read-only independent and references to addresses are suitable for use in a Linux shared object. The default is `/nofpic`.

`/hardfp, /softfp`
: Requests hardware or software floating-point linkage. This enables the procedure call standard to be specified separately from the version of the floating-point hardware available through the `--fpu` option. It is still possible to specify the procedure call standard by using the `--fpu` option, but Arm recommends you use `--apcs`. If floating-point support is not permitted (for example, because `--fpu=none` is specified, or because of other means), then `/hardfp` and `/softfp` are ignored. If floating-point support is permitted and the softfp calling convention is used (`--fpu=softvfp` or `--fpu=softvfp+fp-armv8`), then `/hardfp` gives an error.

  `/softfp` is not supported for AArch64 state.

Usage

This option specifies whether you are using the *Procedure Call Standard for the Arm® Architecture* (AAPCS). It can also specify some attributes of code sections.

The AAPCS forms part of the *Base Standard Application Binary Interface for the Arm® Architecture* (BSABI) specification. By writing code that adheres to the AAPCS, you can ensure that separately compiled and assembled modules can work together.

Note

AAPCS qualifiers do not affect the code produced by `armasm`. They are an assertion by the programmer that the code in the input file complies with a particular variant of AAPCS. They cause attributes to be set in the object file produced by `armasm`. The linker uses these attributes to check compatibility of files, and to select appropriate library variants.

Example

```
armasm --cpu=8-A.32 --apcs=/inter/hardfp inputfile.s
```

Related information

Procedure Call Standard for the Arm Architecture.
Application Binary Interface (ABI) for the Arm Architecture.

11.4 --arm

Instructs `armasm` to interpret instructions as A32 instructions. It does not, however, guarantee A32-only code in the object file. This is the default. Using this option is equivalent to specifying the `ARM` or `CODE32` directive at the start of the source file.

Note

Not supported for AArch64 state.

Related references

11.2 --32 on page 11-227.

11.5 --arm_only on page 11-231.

21.7 ARM or CODE32 directive on page 21-1653.

11.5 --arm_only

Instructs `armasm` to only generate A32 code. This is similar to `--arm` but also has the property that `armasm` does not permit the generation of any T32 code.

Note

Not supported for AArch64 state.

Related references

11.4 --arm on page 11-230.

11.6 --bi

A synonym for the `--bigend` command-line option.

Related references

11.7 --bigend on page 11-233.
11.42 --littleend on page 11-270.

11.7 --bigend

Generates code suitable for an Arm processor using big-endian memory access.

The default is `--littleend`.

Related references

11.42 --littleend on page 11-270.
11.6 --bi on page 11-232.

11.8 --brief_diagnostics, --no_brief_diagnostics

Enables and disables the output of brief diagnostic messages.

This option instructs the assembler whether to use a shorter form of the diagnostic output. In this form, the original source line is not displayed and the error message text is not wrapped when it is too long to fit on a single line. The default is `--no_brief_diagnostics`.

Related references

11.17 --diag_error=tag[,tag,...] on page 11-245.
11.21 --diag_warning=tag[,tag,...] on page 11-249.

11.9 --checkreglist

Instructs the `armasm` to check `RLIST`, `LDM`, and `STM` register lists to ensure that all registers are provided in increasing register number order.

When this option is used, `armasm` gives a warning if the registers are not listed in order.

Note

In AArch32 state, this option is deprecated. Use `--diag_warning 1206` instead. In AArch64 state, this option is not supported..

Related references

11.21 --diag_warning=tag[,tag,...] on page 11-249.

11.10 --cpreproc

Instructs `armasm` to call `armclang` to preprocess the input file before assembling it.

Restrictions

You must use `--cpreproc_opts` with this option to correctly configure the `armclang` compiler for pre-processing.

`armasm` only passes the following command-line options to `armclang` by default:

- Basic pre-processor configuration options, such as `-E`.
- User specified include directories, `-I` directives.
- User specified licensing options, such as `--site_license`.
- Anything specified in `--cpreproc_opts`.

Related concepts

8.14 Using the C preprocessor on page 8-176.

Related references

11.11 --cpreproc_opts=option[,option,...] on page 11-237.

Related information

-x armclang option.

Command-line options for preprocessing assembly source code.

11.11 --cpreproc_opts=option[,option,…]

Enables `armasm` to pass options to `armclang` when using the C preprocessor.

Syntax

`--cpreproc_opts=option[,option,…]`

Where `option[,option,…]` is a comma-separated list of C preprocessing options.

At least one option must be specified.

Restrictions

As a minimum, you must specify the `armclang` options `--target` and either `-mcpu` or `-march` in `--cpreproc_opts`.

To assemble code containing C directives that require the C preprocessor, the input assembly source filename must have an upper-case extension `.S`.

You cannot pass the `armclang` option `-x assembler-with-cpp`, because it gets added to `armclang` after the source file name.

Note

Ensure that you specify compatible architectures in the `armclang` options `--target`, `-mcpu` or `-march`, and the `armasm --cpu` option.

Example

The options to the preprocessor in this example are `--cpreproc_opts=--target=arm-arm-none-eabi,-mcpu=cortex-a9,-D,DEF1,-D,DEF2`.

```
armasm --cpu=cortex-a9 --cpreproc --cpreproc_opts=--target=arm-arm-none-eabi,-mcpu=cortex-a9,-D,DEF1,-D,DEF2 -I /path/to/includes1 -I /path/to/includes2 input.S
```

Related concepts

8.14 Using the C preprocessor on page 8-176.

Related references

11.10 --cpreproc on page 11-236.

Related information

Command-line options for preprocessing assembly source code.
Specifying a target architecture, processor, and instruction set.
-march armclang option.
-mcpu armclang option.
--target armclang option.
-x armclang option.

11.12 --cpu=list

Lists the architecture and processor names that are supported by the `--cpu=name` option.

Syntax

```
--cpu=list
```

Related references

11.13 --cpu=name on page 11-239.

11.13 --cpu=name

Enables code generation for the selected Arm processor or architecture.

Syntax

`--cpu=`*`name`*

Where *`name`* is the name of a processor or architecture:

Processor and architecture names are not case-sensitive.

Wildcard characters are not accepted.

The following table shows the supported architectures. For a complete list of the supported architecture and processor names, specify the `--cpu=list` option.

Note

`armasm` does not support architectures later than Armv8.3.

Table 11-1 Supported Arm architectures

Architecture name	Description
`6-M`	Armv6 architecture microcontroller profile.
`6S-M`	Armv6 architecture microcontroller profile with OS extensions.
`7-A`	Armv7 architecture application profile.
`7-A.security`	Armv7-A architecture profile with Security Extensions and includes the `SMC` instruction (formerly `SMI`).
`7-R`	Armv7 architecture real-time profile.
`7-M`	Armv7 architecture microcontroller profile.
`7E-M`	Armv7-M architecture profile with DSP extension.
`8-A.32`	Armv8-A architecture profile, AArch32 state.
`8-A.32.crypto`	Armv8-A architecture profile, AArch32 state with cryptographic instructions.
`8-A.64`	Armv8-A architecture profile, AArch64 state.
`8-A.64.crypto`	Armv8-A architecture profile, AArch64 state with cryptographic instructions.
`8.1-A.32`	Armv8.1, for Armv8-A architecture profile, AArch32 state.
`8.1-A.32.crypto`	Armv8.1, for Armv8-A architecture profile, AArch32 state with cryptographic instructions.
`8.1-A.64`	Armv8.1, for Armv8-A architecture profile, AArch64 state.
`8.1-A.64.crypto`	Armv8.1, for Armv8-A architecture profile, AArch64 state with cryptographic instructions.
`8.2-A.32`	Armv8.2, for Armv8-A architecture profile, AArch32 state.
`8.2-A.32.crypto`	Armv8.2, for Armv8-A architecture profile, AArch32 state with cryptographic instructions.
`8.2-A.64`	Armv8.2, for Armv8-A architecture profile, AArch64 state.
`8.2-A.64.crypto`	Armv8.2, for Armv8-A architecture profile, AArch64 state with cryptographic instructions.
`8.3-A.32`	Armv8.3, for Armv8-A architecture profile, AArch32 state.
`8.3-A.32.crypto`	Armv8.3, for Armv8-A architecture profile, AArch32 state with cryptographic instructions.
`8.3-A.64`	Armv8.3, for Armv8-A architecture profile, AArch64 state.
`8.3-A.64.crypto`	Armv8.3, for Armv8-A architecture profile, AArch64 state with cryptographic instructions.
`8-R`	Armv8-R architecture profile.

Table 11-1 Supported Arm architectures (continued)

Architecture name	Description
`8-M.Base`	Armv8-M baseline architecture profile. Derived from the Armv6-M architecture.
`8-M.Main`	Armv8-M mainline architecture profile. Derived from the Armv7-M architecture.
`8-M.Main.dsp`	Armv8-M mainline architecture profile with DSP extension.

Note

The full list of supported architectures and processors depends on your license.

Default

There is no default option for `--cpu`.

Usage

The following general points apply to processor and architecture options:

Processors

- Selecting the processor selects the appropriate architecture, *Floating-Point Unit* (FPU), and memory organization.
- If you specify a processor for the `--cpu` option, the generated code is optimized for that processor. This enables the assembler to use specific coprocessors or instruction scheduling for optimum performance.

Architectures

- If you specify an architecture name for the `--cpu` option, the generated code can run on any processor supporting that architecture. For example, `--cpu=7-A` produces code that can be used by the Cortex-A9 processor.

FPU

- Some specifications of `--cpu` imply an `--fpu` selection.

  Note

  Any explicit FPU, set with `--fpu` on the command line, overrides an implicit FPU.

- If no `--fpu` option is specified and the `--cpu` option does not imply an `--fpu` selection, then `--fpu=softvfp` is used.

A32/T32

- Specifying a processor or architecture that supports T32 instructions, such as `--cpu=cortex-a9`, does not make the assembler generate T32 code. It only enables features of the processor to be used, such as long multiply. Use the `--thumb` option to generate T32 code, unless the processor only supports T32 instructions.

  Note

  Specifying the target processor or architecture might make the generated object code incompatible with other Arm processors. For example, A32 code generated for architecture Armv8 might not run on a Cortex-A9 processor, if the generated object code includes instructions specific to Armv8. Therefore, you must choose the lowest common denominator processor suited to your purpose.

- If the architecture only supports T32, you do not have to specify `--thumb` on the command line. For example, if building for Cortex-M4 or Armv7-M with `--cpu=7-M`, you do not have to specify `--thumb` on the command line, because Armv7-M only supports T32. Similarly, Armv6-M and other T32-only architectures.

Restrictions

You cannot specify both a processor and an architecture on the same command-line.

Example

```
armasm --cpu=Cortex-A17 inputfile.s
```

Related references

11.3 --apcs=qualifier...qualifier on page 11-228.
11.12 --cpu=list on page 11-238.
11.32 --fpu=name on page 11-260.
11.59 --thumb on page 11-287.
11.61 --unsafe on page 11-289.

Related information

Arm Architecture Reference Manual.

11.14 --debug

Instructs the assembler to generate DWARF debug tables.

`--debug` is a synonym for `-g`. The default is DWARF 3.

Note

Local symbols are not preserved with `--debug`. You must specify `--keep` if you want to preserve the local symbols to aid debugging.

Related references

11.23 --dwarf2 on page 11-251.
11.24 --dwarf3 on page 11-252.
11.36 --keep on page 11-264.
11.33 -g on page 11-261.

11.15 --depend=dependfile

Writes makefile dependency lines to a file.

Source file dependency lists are suitable for use with make utilities.

Related references

11.45 --md on page 11-273.
11.16 --depend_format=string on page 11-244.

11.16 --depend_format=string

Specifies the format of output dependency files, for compatibility with some UNIX make programs.

Syntax

`--depend_format=`*`string`*

Where *`string`* is one of:

`unix`
: generates dependency file entries using UNIX-style path separators.

`unix_escaped`
: is the same as `unix`, but escapes spaces with `\`.

`unix_quoted`
: is the same as `unix`, but surrounds path names with double quotes.

Related references

11.15 --depend=dependfile on page 11-243.

11.17 --diag_error=tag[,tag,…]

Sets diagnostic messages that have a specific tag to Error severity.

Syntax

`--diag_error=tag[,tag,…]`
Where *tag* can be:

- A diagnostic message number to set to error severity. This is the four-digit number, *nnnn*, with the tool letter prefix, but without the letter suffix indicating the severity.
- `warning`, to treat all warnings as errors.

Usage

Diagnostic messages output by the assembler can be identified by a tag in the form of {*prefix*}*number*, where the *prefix* is A.

You can specify more than one tag with this option by separating each tag using a comma. You can specify the optional assembler prefix `A` before the tag number. If any prefix other than A is included, the message number is ignored.

The following table shows the meaning of the term severity used in the option descriptions:

Table 11-2 Severity of diagnostic messages

Severity	Description
Error	Errors indicate violations in the syntactic or semantic rules of assembly language. Assembly continues, but object code is not generated.
Warning	Warnings indicate unusual conditions in your code that might indicate a problem. Assembly continues, and object code is generated unless any problems with an Error severity are detected.
Remark	Remarks indicate common, but not recommended, use of assembly language. These diagnostics are not issued by default. Assembly continues, and object code is generated unless any problems with an Error severity are detected.

Related references

11.8 --brief_diagnostics, --no_brief_diagnostics on page 11-234.
11.18 --diag_remark=tag[,tag,...] on page 11-246.
11.20 --diag_suppress=tag[,tag,...] on page 11-248.
11.21 --diag_warning=tag[,tag,...] on page 11-249.

11.18 --diag_remark=tag[,tag,…]

Sets diagnostic messages that have a specific tag to Remark severity.

Syntax

`--diag_remark=`*`tag[,tag,…]`*

Where *`tag`* is a comma-separated list of diagnostic message numbers. This is the four-digit number, *`nnnn`*, with the tool letter prefix, but without the letter suffix indicating the severity.

Usage

Diagnostic messages output by the assembler can be identified by a tag in the form of {*`prefix`*}*`number`*, where the *`prefix`* is A.

You can specify more than one tag with this option by separating each tag using a comma. You can specify the optional assembler prefix `A` before the tag number. If any prefix other than A is included, the message number is ignored.

Related references

11.8 --brief_diagnostics, --no_brief_diagnostics on page 11-234.
11.17 --diag_error=tag[,tag,...] on page 11-245.
11.20 --diag_suppress=tag[,tag,...] on page 11-248.
11.21 --diag_warning=tag[,tag,...] on page 11-249.

11.19 --diag_style={arm|ide|gnu}

Specifies the display style for diagnostic messages.

Syntax

`--diag_style=`*`string`*

Where *`string`* is one of:

`arm`
: Display messages using the legacy Arm compiler style.

`ide`
: Include the line number and character count for any line that is in error. These values are displayed in parentheses.

`gnu`
: Display messages in the format used by `gcc`.

Usage

`--diag_style=gnu` matches the format reported by the GNU Compiler, `gcc`.

`--diag_style=ide` matches the format reported by Microsoft Visual Studio.

Choosing the option `--diag_style=ide` implicitly selects the option `--brief_diagnostics`. Explicitly selecting `--no_brief_diagnostics` on the command line overrides the selection of `--brief_diagnostics` implied by `--diag_style=ide`.

Selecting either the option `--diag_style=arm` or the option `--diag_style=gnu` does not imply any selection of `--brief_diagnostics`.

Default

The default is `--diag_style=arm`.

Related references

11.8 --brief_diagnostics, --no_brief_diagnostics on page 11-234.

11.20 --diag_suppress=tag[,tag,…]

Suppresses diagnostic messages that have a specific tag.

Syntax

`--diag_suppress=tag[,tag,…]`
Where *tag* can be:

- A diagnostic message number to be suppressed. This is the four-digit number, *nnnn*, with the tool letter prefix, but without the letter suffix indicating the severity.
- `error`, to suppress all errors that can be downgraded.
- `warning`, to suppress all warnings.

Diagnostic messages output by `armasm` can be identified by a tag in the form of {*prefix*}*number*, where the *prefix* is A.

You can specify more than one tag with this option by separating each tag using a comma.

Example

For example, to suppress the warning messages that have numbers `1293` and `187`, use the following command:

```
armasm --cpu=8-A.64 --diag_suppress=1293,187
```

You can specify the optional assembler prefix `A` before the tag number. For example:

```
armasm --cpu=8-A.64 --diag_suppress=A1293,A187
```

If any prefix other than A is included, the message number is ignored. Diagnostic message tags can be cut and pasted directly into a command line.

Related references

11.8 --brief_diagnostics, --no_brief_diagnostics on page 11-234.
11.17 --diag_error=tag[,tag,...] on page 11-245.
11.18 --diag_remark=tag[,tag,...] on page 11-246.
11.20 --diag_suppress=tag[,tag,...] on page 11-248.
11.21 --diag_warning=tag[,tag,...] on page 11-249.

11.21 --diag_warning=tag[,tag,…]

Sets diagnostic messages that have a specific tag to Warning severity.

Syntax

`--diag_warning=`*`tag[,tag,…]`*

Where *`tag`* can be:

- A diagnostic message number to set to warning severity. This is the four-digit number, *`nnnn`*, with the tool letter prefix, but without the letter suffix indicating the severity.
- `error`, to set all errors that can be downgraded to warnings.

Diagnostic messages output by the assembler can be identified by a tag in the form of {*`prefix`*}*`number`*, where the *`prefix`* is A.

You can specify more than one tag with this option by separating each tag using a comma.

You can specify the optional assembler prefix `A` before the tag number. If any prefix other than A is included, the message number is ignored.

Related references

11.8 --brief_diagnostics, --no_brief_diagnostics on page 11-234.
11.17 --diag_error=tag[,tag,...] on page 11-245.
11.18 --diag_remark=tag[,tag,...] on page 11-246.
11.20 --diag_suppress=tag[,tag,...] on page 11-248.

11.22 --dllexport_all

Controls symbol visibility when building DLLs.

This option gives all exported global symbols `STV_PROTECTED` visibility in ELF rather than `STV_HIDDEN`, unless overridden by source directives.

Related references

21.27 EXPORT or GLOBAL on page 21-1673.

11.23 --dwarf2

Uses DWARF 2 debug table format.

Note

Not supported for AArch64 state.

This option can be used with `--debug`, to instruct `armasm` to generate DWARF 2 debug tables.

Related references

11.14 --debug on page 11-242.
11.24 --dwarf3 on page 11-252.

11.24 --dwarf3

Uses DWARF 3 debug table format.

This option can be used with `--debug`, to instruct the assembler to generate DWARF 3 debug tables. This is the default if `--debug` is specified.

Related references

11.14 --debug on page 11-242.
11.23 --dwarf2 on page 11-251.

11.25 --errors=errorfile

Redirects the output of diagnostic messages from stderr to the specified errors file.

11.26 --exceptions, --no_exceptions

Enables or disables exception handling.

Note

Not supported for AArch64 state.

These options instruct `armasm` to switch on or off exception table generation for all functions defined by `FUNCTION` (or `PROC`) and `ENDFUNC` (or `ENDP`) directives.

`--no_exceptions` causes no tables to be generated. It is the default.

Related references

11.27 --exceptions_unwind, --no_exceptions_unwind on page 11-255.
21.39 FRAME UNWIND ON on page 21-1686.
21.40 FRAME UNWIND OFF on page 21-1687.
21.41 FUNCTION or PROC on page 21-1688.
21.24 ENDFUNC or ENDP on page 21-1670.

11.27 --exceptions_unwind, --no_exceptions_unwind

Enables or disables function unwinding for exception-aware code. This option is only effective if `--exceptions` is enabled.

Note

Not supported for AArch64 state.

The default is `--exceptions_unwind`.

For finer control, use the `FRAME UNWIND ON` and `FRAME UNWIND OFF` directives.

Related references

11.26 --exceptions, --no_exceptions on page 11-254.
21.39 FRAME UNWIND ON on page 21-1686.
21.40 FRAME UNWIND OFF on page 21-1687.
21.41 FUNCTION or PROC on page 21-1688.
21.24 ENDFUNC or ENDP on page 21-1670.

11.28 --execstack, --no_execstack

Generates a `.note.GNU-stack` section marking the stack as either executable or non-executable.

You can also use the `AREA` directive to generate either an executable or non-executable `.note.GNU-stack` section. The following code generates an executable `.note.GNU-stack` section. Omitting the `CODE` attribute generates a non-executable `.note.GNU-stack` section.

```
    AREA    |.note.GNU-stack|,ALIGN=0,READONLY,NOALLOC,CODE
```

In the absence of `--execstack` and `--no_execstack`, the `.note.GNU-stack` section is not generated unless it is specified by the `AREA` directive.

If both the command-line option and source directive are used and are different, then the stack is marked as executable.

Table 11-3 Specifying a command-line option and an AREA directive for GNU-stack sections

	`--execstack` command-line option	`--no_execstack` command-line option
execstack `AREA` directive	execstack	execstack
no_execstack `AREA` directive	execstack	no_execstack

Related references

21.6 AREA on page 21-1650.

11.29 --execute_only

Adds the `EXECONLY AREA` attribute to all code sections.

Usage

The `EXECONLY AREA` attribute causes the linker to treat the section as execute-only.

It is the user's responsibility to ensure that the code in the section is safe to run in execute-only memory. For example:

- The code must not contain literal pools.
- The code must not attempt to load data from the same, or another, execute-only section.

Restrictions

This option is only supported for:

- Processors that support the Armv8-M.mainline or Armv8-M.baseline architecture.
- Processors that support the Armv7-M architecture, such as Cortex-M3, Cortex-M4, and Cortex-M7.
- Processors that support the Armv6-M architecture.

  Note

  Arm has only performed limited testing of execute-only code on Armv6-M targets.

11.30 --fpmode=model

Specifies floating-point standard conformance and sets library attributes and floating-point optimizations.

Syntax

`--fpmode=model`

Where *model* is one of:

none
Source code is not permitted to use any floating-point type or floating-point instruction. This option overrides any explicit `--fpu=name` option.

ieee_full
All facilities, operations, and representations guaranteed by the IEEE standard are available in single and double-precision. Modes of operation can be selected dynamically at runtime.

ieee_fixed
IEEE standard with round-to-nearest and no inexact exceptions.

ieee_no_fenv
IEEE standard with round-to-nearest and no exceptions. This mode is compatible with the Java floating-point arithmetic model.

std
IEEE finite values with denormals flushed to zero, round-to-nearest and no exceptions. It is C and C++ compatible. This is the default option.

Finite values are as predicted by the IEEE standard. It is not guaranteed that NaNs and infinities are produced in all circumstances defined by the IEEE model, or that when they are produced, they have the same sign. Also, it is not guaranteed that the sign of zero is that predicted by the IEEE model.

fast
Some value altering optimizations, where accuracy is sacrificed to fast execution. This is not IEEE compatible, and is not standard C.

Note

This does not cause any changes to the code that you write.

Example

```
armasm --cpu=8-A.32 --fpmode ieee_full inputfile.s
```

Related references

11.32 --fpu=name on page 11-260.

Related information

IEEE Standards Association.

11.31 --fpu=list

Lists the FPU architecture names that are supported by the `--fpu=name` option.

Example

```
armasm --fpu=list
```

Related references

11.30 --fpmode=model on page 11-258.

11.32 --fpu=name on page 11-260.

11.32 --fpu=name

Specifies the target FPU architecture.

Syntax

`--fpu=`*`name`*

Where *`name`* is the name of the target FPU architecture. Specify `--fpu=list` to list the supported FPU architecture names that you can use with `--fpu=name`.

The default floating-point architecture depends on the target architecture.

Note

Software floating-point linkage is not supported for AArch64 state.

Usage

If you specify this option, it overrides any implicit FPU option that appears on the command line, for example, where you use the `--cpu` option. Floating-point instructions also produce either errors or warnings if assembled for the wrong target FPU.

`armasm` sets a build attribute corresponding to name in the object file. The linker determines compatibility between object files, and selection of libraries, accordingly.

Related references

11.30 --fpmode=model on page 11-258.

11.33 -g

Enables the generation of debug tables.

This option is a synonym for `--debug`.

Related references

11.14 --debug on page 11-242.

11.34 --help

Displays a summary of the main command-line options.

Default

This is the default if you specify `armasm` without any options or source files.

Related references

11.63 --version_number on page 11-291.
11.65 --vsn on page 11-293.

11.35 -idir[,dir, ...]

Adds directories to the source file include path.

Any directories added using this option have to be fully qualified.

Related references

21.43 GET or INCLUDE on page 21-1690.

11.36 --keep

Instructs the assembler to keep named local labels in the symbol table of the object file, for use by the debugger.

Related references

21.48 KEEP on page 21-1697.

11.37 --length=n

Sets the listing page length.

Length zero means an unpaged listing. The default is 66 lines.

Related references

11.40 --list=file on page 11-268.

11.38 --li

A synonym for the `--littleend` command-line option.

Related references

11.42 --littleend on page 11-270.
11.7 --bigend on page 11-233.

11.39 --library_type=lib

Enables the selected library to be used at link time.

Syntax

`--library_type=lib`

Where `lib` is one of:

`standardlib`
Specifies that the full Arm runtime libraries are selected at link time. This is the default.

`microlib`
Specifies that the C micro-library (microlib) is selected at link time.

Note

- This option can be used with the compiler, assembler, or linker when use of the libraries require more specialized optimizations.
- This option can be overridden at link time by providing it to the linker.
- microlib is not supported for AArch64 state.

Related information

Building an application with microlib.

11.40 --list=file

Instructs the assembler to output a detailed listing of the assembly language produced by the assembler to a file.

If - is given as *file*, the listing is sent to `stdout`.

Use the following command-line options to control the behavior of `--list`:

- `--no_terse`.
- `--width`.
- `--length`.
- `--xref`.

Related references

11.50 --no_terse on page 11-278.
11.66 --width=n on page 11-294.
11.37 --length=n on page 11-265.
11.67 --xref on page 11-295.
21.55 OPT on page 21-1706.

11.41 --list=

Instructs the assembler to send the detailed assembly language listing to *inputfile*.lst.

Note

You can use `--list` without the equals sign and filename to send the output to *inputfile*.lst. However, this syntax is deprecated and the assembler issues a warning. This syntax is to be removed in a later release. Use `--list=` instead.

Related references

11.40 --list=file on page 11-268.

11.42 --littleend

Generates code suitable for an Arm processor using little-endian memory access.

Related references

11.7 --bigend on page 11-233.
11.38 --li on page 11-266.

11.43 -m

Instructs the assembler to write source file dependency lists to `stdout`.

Related references

11.45 --md on page 11-273.

11.44 --maxcache=n

Sets the maximum source cache size in bytes.

The default is 8MB. `armasm` gives a warning if the size is less than 8MB.

11.45 --md

Creates makefile dependency lists.

This option instructs the assembler to write source file dependency lists to *inputfile*.d.

Related references

11.43 -m on page 11-271.

11.46 --no_code_gen

Instructs the assembler to exit after pass 1, generating no object file. This option is useful if you only want to check the syntax of the source code or directives.

11.47 --no_esc

Instructs the assembler to ignore C-style escaped special characters, such as \n and \t.

11.48 --no_hide_all

Gives all exported and imported global symbols `STV_DEFAULT` visibility in ELF rather than `STV_HIDDEN`, unless overridden using source directives.

You can use the following directives to specify an attribute that overrides the implicit symbol visibility:

- `EXPORT`.
- `EXTERN`.
- `GLOBAL`.
- `IMPORT`.

Related references

21.27 EXPORT or GLOBAL on page 21-1673.
21.45 IMPORT and EXTERN on page 21-1693.

11.49 --no_regs

Instructs `armasm` not to predefine register names.

Note

This option is deprecated. In AArch32 state, use `--regnames=none` instead.

Related references

11.56 --regnames on page 11-284.

11.50 --no_terse

Instructs the assembler to show in the list file the lines of assembly code that it has skipped because of conditional assembly.

If you do not specify this option, the assembler does not output the skipped assembly code to the list file.

This option turns off the terse flag. By default the terse flag is on.

Related references

11.40 --list=file on page 11-268.

11.51 --no_warn

Turns off warning messages.

Related references

11.21 --diag_warning=tag[,tag,...] on page 11-249.

11.52 -o filename

Specifies the name of the output file.

If this option is not used, the assembler creates an object filename in the form *inputfilename*.o. This option is case-sensitive.

11.53 --pd

A synonym for the `--predefine` command-line option.

Related references

11.54 --predefine "directive" on page 11-282.

11.54 --predefine "directive"

Instructs `armasm` to pre-execute one of the `SETA`, `SETL`, or `SETS` directives.

You must enclose *`directive`* in quotes, for example:

```
armasm --cpu=8-A.64 --predefine "VariableName SETA 20" inputfile.s
```

`armasm` also executes a corresponding `GBLL`, `GBLS`, or `GBLA` directive to define the variable before setting its value.

The variable name is case-sensitive. The variables defined using the command line are global to `armasm` source files specified on the command line.

Considerations when using --predefine

Be aware of the following:

- The command-line interface of your system might require you to enter special character combinations, such as `\"`, to include strings in *`directive`*. Alternatively, you can use `--via` *`file`* to include a `--predefine` argument. The command-line interface does not alter arguments from `--via` files.
- `--predefine` is not equivalent to the compiler option `-D`*`name`*. `--predefine` defines a global variable whereas `-D`*`name`* defines a macro that the C preprocessor expands.

  Although you can use predefined global variables in combination with assembly control directives, for example `IF` and `ELSE` to control conditional assembly, they are not intended to provide the same functionality as the C preprocessor in `armasm`. If you require this functionality, Arm recommends you use the compiler to pre-process your assembly code.

Related references

11.53 --pd on page 11-281.
21.42 GBLA, GBLL, and GBLS on page 21-1689.
21.44 IF, ELSE, ENDIF, and ELIF on page 21-1691.
21.63 SETA, SETL, and SETS on page 21-1716.

11.55 --reduce_paths, --no_reduce_paths

Enables or disables the elimination of redundant path name information in file paths.

Windows systems impose a 260 character limit on file paths. Where relative pathnames exist whose absolute names expand to longer than 260 characters, you can use the `--reduce_paths` option to reduce absolute pathname length by matching up directories with corresponding instances of `..` and eliminating the directory/`..` sequences in pairs.

`--no_reduce_paths` is the default.

Note

Arm recommends that you avoid using long and deeply nested file paths, in preference to minimizing path lengths using the `--reduce_paths` option.

Note

This option is valid for 32-bit Windows systems only.

11.56 --regnames

Controls the predefinition of register names.

Note

Not supported for AArch64 state.

Syntax

`--regnames=option`

Where `option` is one of the following:

none
Instructs `armasm` not to predefine register names.

callstd
Defines additional register names based on the AAPCS variant that you are using, as specified by the `--apcs` option.

all
Defines all AAPCS registers regardless of the value of `--apcs`.

Related references

11.49 --no_regs on page 11-277.
3.7 Predeclared core register names in AArch32 state on page 3-71.
3.8 Predeclared extension register names in AArch32 state on page 3-72.
11.56 --regnames on page 11-284.
11.3 --apcs=qualifier...qualifier on page 11-228.

11.57 --report-if-not-wysiwyg

Instructs `armasm` to report when it outputs an encoding that was not directly requested in the source code.

This can happen when `armasm`:

- Uses a pseudo-instruction that is not available in other assemblers, for example `MOV32`.
- Outputs an encoding that does not directly match the instruction mnemonic, for example if the assembler outputs the `MVN` encoding when assembling the `MOV` instruction.
- Inserts additional instructions where necessary for instruction syntax semantics, for example `armasm` can insert a missing `IT` instruction before a conditional T32 instruction.

Note

Not supported for AArch64 state.

11.58 --show_cmdline

Outputs the command line used by the assembler.

Usage

Shows the command line after processing by the assembler, and can be useful to check:

- The command line a build system is using.
- How the assembler is interpreting the supplied command line, for example, the ordering of command-line options.

The commands are shown normalized, and the contents of any via files are expanded.

The output is sent to the standard error stream (`stderr`).

Related references

11.64 --via=filename on page 11-292.

11.59 --thumb

Instructs `armasm` to interpret instructions as T32 instructions, using UAL syntax. This is equivalent to a `THUMB` directive at the start of the source file.

Note

Not supported for AArch64 state.

Related references

11.4 --arm on page 11-230.

21.65 THUMB directive on page 21-1719.

11.60 --unaligned_access, --no_unaligned_access

Enables or disables unaligned accesses to data on Arm-based processors.

These options instruct the assembler to set an attribute in the object file to enable or disable the use of unaligned accesses.

11.61 --unsafe

Enables instructions for other architectures to be assembled without error.

Note

Not supported for AArch64 state.

It downgrades error messages to corresponding warning messages. It also suppresses warnings about operator precedence.

Related concepts

12.20 Binary operators on page 12-317.

Related references

11.17 --diag_error=tag[,tag,...] on page 11-245.

11.21 --diag_warning=tag[,tag,...] on page 11-249.

11.62 --untyped_local_labels

Causes `armasm` not to set the T32 bit for the address of a numeric local label referenced in an `LDR` pseudo-instruction.

Note

Not supported for AArch64 state.

When this option is not used, if you reference a numeric local label in an `LDR` pseudo-instruction, and the label is in T32 code, then `armasm` sets the T32 bit (bit 0) of the address. You can then use the address as the target for a `BX` or `BLX` instruction.

If you require the actual address of the numeric local label, without the T32 bit set, then use this option.

Note

When using this option, if you use the address in a branch (register) instruction, `armasm` treats it as an A32 code address, causing the branch to arrive in A32 state, meaning it would interpret this code as A32 instructions.

Example

```
THUMB
    ...
1
    ...
    LDR r0,=%B1 ; r0 contains the address of numeric local label "1",
                ; T32 bit is not set if --untyped_local_labels was used
    ...
```

Related concepts

12.10 Numeric local labels on page 12-307.

Related references

13.54 LDR pseudo-instruction on page 13-417.

13.15 B on page 13-359.

11.63 --version_number

Displays the version of `armasm` you are using.

Usage

The assembler displays the version number in the format `Mmmuuxx`, where:

- *`M`* is the major version number, `6`.
- *`mm`* is the minor version number.
- *`uu`* is the update number.
- *`xx`* is reserved for Arm internal use. You can ignore this for the purposes of checking whether the current release is a specific version or within a range of versions.

11.64 --via=filename

Reads an additional list of input filenames and assembler options from *filename*.

Syntax

`--via=`*filename*

Where *filename* is the name of a via file containing options to be included on the command line.

Usage

You can enter multiple `--via` options on the assembler command line. The `--via` options can also be included within a via file.

Related concepts

22.1 Overview of via files on page 22-1724.

Related references

22.2 Via file syntax rules on page 22-1725.

11.65 --vsn

Displays the version information and the license details.

Note

`--vsn` is intended to report the version information for manual inspection. The `Component` line indicates the release of Arm Compiler you are using. If you need to access the version in other tools or scripts, for example in build scripts, use the output from `--version_number`.

Example

```
> armasm --vsn
Product: ARM Compiler N.n
Component: ARM Compiler N.n
Tool: armasm [tool_id]
license_type
Software supplied by: ARM Limited
```

11.66 --width=n

Sets the listing page width.

The default is 79 characters.

Related references

11.40 --list=file on page 11-268.

11.67 --xref

Instructs the assembler to list cross-referencing information on symbols, including where they were defined and where they were used, both inside and outside macros.

The default is off.

Related references

11.40 --list=file on page 11-268.

Chapter 12
Symbols, Literals, Expressions, and Operators

Describes how you can use symbols to represent variables, addresses, and constants in code, and how you can combine these with operators to create numeric or string expressions.

It contains the following sections:

12.1 Symbol naming rules

You must follow some rules when naming symbols in assembly language source code.

The following rules apply:

- Symbol names must be unique within their scope.
- You can use uppercase letters, lowercase letters, numeric characters, or the underscore character in symbol names. Symbol names are case-sensitive, and all characters in the symbol name are significant.
- Do not use numeric characters for the first character of symbol names, except in numeric local labels.
- Symbols must not use the same name as built-in variable names or predefined symbol names.
- If you use the same name as an instruction mnemonic or directive, use double bars to delimit the symbol name. For example:

  ```
  ||ASSERT||
  ```

  The bars are not part of the symbol.
- You must not use the symbols `|$a|`, `|$t|`, or `|$d|` as program labels. These are mapping symbols that mark the beginning of A32, T32, and A64 code, and data within the object file. You must not use `|$x|`in A64 code.
- Symbols beginning with the characters $v are mapping symbols that relate to floating-point code. Arm recommends you avoid using symbols beginning with $v in your source code.

If you have to use a wider range of characters in symbols, for example, when working with compilers, use single bars to delimit the symbol name. For example:

```
|.text|
```

The bars are not part of the symbol. You cannot use bars, semicolons, or newlines within the bars.

Related concepts

12.10 Numeric local labels on page 12-307.

Related references

3.7 Predeclared core register names in AArch32 state on page 3-71.

3.8 Predeclared extension register names in AArch32 state on page 3-72.

8.4 Built-in variables and constants on page 8-163.

12.2 Variables

You can declare numeric, logical, or string variables using assembler directives.

The value of a variable can be changed as assembly proceeds. Variables are local to the assembler. This means that in the generated code or data, every instance of the variable has a fixed value.

The type of a variable cannot be changed. Variables are one of the following types:

- Numeric.
- Logical.
- String.

The range of possible values of a numeric variable is the same as the range of possible values of a numeric constant or numeric expression.

The possible values of a logical variable are `{TRUE}` or `{FALSE}`.

The range of possible values of a string variable is the same as the range of values of a string expression.

Use the `GBLA`, `GBLL`, `GBLS`, `LCLA`, `LCLL`, and `LCLS` directives to declare symbols representing variables, and assign values to them using the `SETA`, `SETL`, and `SETS` directives.

Example

```
a    SETA 100
L1   MOV R1, #(a*5) ; In the object file, this is MOV R1, #500
a    SETA 200       ; Value of 'a' is 200 only after this point.
                    ; The previous instruction is always MOV R1, #500
     ...
     BNE L1         ; When the processor branches to L1, it executes
                    ; MOV R1, #500
```

Related concepts

12.14 Numeric expressions on page 12-311.
12.12 String expressions on page 12-309.
12.3 Numeric constants on page 12-300.
12.17 Logical expressions on page 12-314.

Related references

21.42 GBLA, GBLL, and GBLS on page 21-1689.
21.49 LCLA, LCLL, and LCLS on page 21-1698.
21.63 SETA, SETL, and SETS on page 21-1716.

12.3 Numeric constants

You can define 32-bit numeric constants using the `EQU` assembler directive.

Numeric constants are 32-bit integers in A32 and T32 code. You can set them using unsigned numbers in the range 0 to 2^{32}-1, or signed numbers in the range -2^{31} to 2^{31} -1. However, the assembler makes no distinction between -*n* and 2^{32}-*n*.

In A64 code, numeric constants are 64-bit integers. You can set them using unsigned numbers in the range 0 to 2^{64}-1, or signed numbers in the range -2^{63} to 2^{63}-1. However, the assembler makes no distinction between -*n* and 2^{64}-*n*.

Relational operators such as >= use the unsigned interpretation. This means that 0 > -1 is `{FALSE}`.

Use the `EQU` directive to define constants. You cannot change the value of a numeric constant after you define it. You can construct expressions by combining numeric constants and binary operators.

Related concepts

12.14 Numeric expressions on page 12-311.

Related references

12.15 Syntax of numeric literals on page 12-312.

21.26 EQU on page 21-1672.

12.4 Assembly time substitution of variables

You can assign a string variable to all or part of a line of assembly language code. A string variable can contain numeric and logical variables.

Use the variable with a `$` prefix in the places where the value is to be substituted for the variable. The dollar character instructs `armasm` to substitute the string into the source code line before checking the syntax of the line. `armasm` faults if the substituted line is larger than the source line limit.

Numeric and logical variables can also be substituted. The current value of the variable is converted to a hexadecimal string (or `T` or `F` for logical variables) before substitution.

Use a dot to mark the end of the variable name if the following character would be permissible in a symbol name. You must set the contents of the variable before you can use it.

If you require a `$` that you do not want to be substituted, use `$$`. This is converted to a single `$`.

You can include a variable with a `$` prefix in a string. Substitution occurs in the same way as anywhere else.

Substitution does not occur within vertical bars, except that vertical bars within double quotes do not affect substitution.

Example

```
    ; straightforward substitution
            GBLS    add4ff
            ;
add4ff      SETS    "ADD  r4,r4,#0xFF"    ; set up add4ff
            $add4ff.00                    ; invoke add4ff
            ; this produces
            ADD  r4,r4,#0xFF00
    ; elaborate substitution
            GBLS    s1
            GBLS    s2
            GBLS    fixup
            GBLA    count
            ;
count       SETA    14
s1          SETS    "a$$b$count" ; s1 now has value a$b0000000E
s2          SETS    "abc"
fixup       SETS    "|xy$s2.z|"  ; fixup now has value |xyabcz|
|C$$code|   MOV     r4,#16       ; but the label here is C$$code
```

Related references

5.1 Syntax of source lines in assembly language on page 5-94.
12.1 Symbol naming rules on page 12-298.

12.5 Register-relative and PC-relative expressions

The assembler supports PC-relative and register-relative expressions.

A register-relative expression evaluates to a named register combined with a numeric expression.

You write a PC-relative expression in source code as a label or the PC, optionally combined with a numeric expression. Some instructions can also accept PC-relative expressions in the form `[PC, #number]`.

If you specify a label, the assembler calculates the offset from the PC value of the current instruction to the address of the label. The assembler encodes the offset in the instruction. If the offset is too large, the assembler produces an error. The offset is either added to or subtracted from the PC value to form the required address.

Arm recommends you write PC-relative expressions using labels rather than the PC because the value of the PC depends on the instruction set.

Note

- In A32 code, the value of the PC is the address of the current instruction plus 8 bytes.
- In T32 code:
  - For `B`, `BL`, `CBNZ`, and `CBZ` instructions, the value of the PC is the address of the current instruction plus 4 bytes.
  - For all other instructions that use labels, the value of the PC is the address of the current instruction plus 4 bytes, with bit[1] of the result cleared to 0 to make it word-aligned.
- In A64 code, the value of the PC is the address of the current instruction.

Example

```
        LDR     r4,=data+4*n    ; n is an assembly-time variable
        ; code
        MOV     pc,lr
data    DCD     value_0
        ; n-1 DCD directives
        DCD     value_n         ; data+4*n points here
        ; more DCD directives
```

Related concepts

12.6 Labels on page 12-303.

Related references

21.52 MAP on page 21-1703.

12.6 Labels

A label is a symbol that represents the memory address of an instruction or data.

The address can be PC-relative, register-relative, or absolute. Labels are local to the source file unless you make them global using the `EXPORT` directive.

The address given by a label is calculated during assembly. `armasm` calculates the address of a label relative to the origin of the section where the label is defined. A reference to a label within the same section can use the PC plus or minus an offset. This is called *PC-relative addressing*.

Addresses of labels in other sections are calculated at link time, when the linker has allocated specific locations in memory for each section.

Related concepts

12.7 Labels for PC-relative addresses on page 12-304.

12.8 Labels for register-relative addresses on page 12-305.

12.9 Labels for absolute addresses on page 12-306.

Related references

5.1 Syntax of source lines in assembly language on page 5-94.

21.27 EXPORT or GLOBAL on page 21-1673.

12.7 Labels for PC-relative addresses

A label can represent the PC value plus or minus the offset from the PC to the label. Use these labels as targets for branch instructions, or to access small items of data embedded in code sections.

You can define PC-relative labels using a label on an instruction or on one of the data definition directives.

You can also use the section name of an `AREA` directive as a label for PC-relative addresses. In this case the label points to the first byte of the specified `AREA`. Arm does not recommend using `AREA` names as branch targets because when branching from A32 to T32 state or T32 to A32 state in this way, the processor does not change the state properly.

Related references

21.6 AREA on page 21-1650.
21.15 DCB on page 21-1661.
21.16 DCD and DCDU on page 21-1662.
21.18 DCFD and DCFDU on page 21-1664.
21.19 DCFS and DCFSU on page 21-1665.
21.20 DCI on page 21-1666.
21.21 DCQ and DCQU on page 21-1667.
21.22 DCW and DCWU on page 21-1668.

12.8 Labels for register-relative addresses

A label can represent a named register plus a numeric value. You define these labels in a storage map. They are most commonly used to access data in data sections.

You can use the `EQU` directive to define additional register-relative labels, based on labels defined in storage maps.

Note

Register-relative addresses are not supported in A64 code.

Example of storage map definitions

```
        MAP     0,r9
        MAP     0xff,r9
```

Related references

21.17 DCDO on page 21-1663.
21.26 EQU on page 21-1672.
21.52 MAP on page 21-1703.
21.64 SPACE or FILL on page 21-1718.

12.9 Labels for absolute addresses

A label can represent the absolute address of code or data.

These labels are numeric constants. In A32 and T32 code they are integers in the range 0 to 2^{32}-1. In A64 code, they are integers in the range 0 to 2^{64}-1. They address the memory directly. You can use labels to represent absolute addresses using the `EQU` directive. To ensure that the labels are used correctly when referenced in code, you can specify the absolute address as:

- A32 code with the `ARM` directive.
- T32 code with the `THUMB` directive.
- Data.

Example of defining labels for absolute address

```
abc EQU 2               ; assigns the value 2 to the symbol abc
xyz EQU label+8         ; assigns the address (label+8) to the symbol xyz
fiq EQU 0x1C, ARM       ; assigns the absolute address 0x1C to the symbol fiq
                        ; and marks it as A32 code
```

Related concepts

12.6 Labels on page 12-303.

12.7 Labels for PC-relative addresses on page 12-304.

12.8 Labels for register-relative addresses on page 12-305.

Related references

21.26 EQU on page 21-1672.

12.10 Numeric local labels

Numeric local labels are a type of label that you refer to by number rather than by name. They are used in a similar way to PC-relative labels, but their scope is more limited.

A numeric local label is a number in the range 0-99, optionally followed by a name. Unlike other labels, a numeric local label can be defined many times and the same number can be used for more than one numeric local label in an area.

Numeric local labels do not appear in the object file. This means that, for example, a debugger cannot set a breakpoint directly on a numeric local label, like it can for named local labels kept using the `KEEP` directive.

A numeric local label can be used in place of *`symbol`* in source lines in an assembly language module:

- On its own, that is, where there is no instruction or directive.
- On a line that contains an instruction.
- On a line that contains a code- or data-generating directive.

A numeric local label is generally used where you might use a PC-relative label.

Numeric local labels are typically used for loops and conditional code within a routine, or for small subroutines that are only used locally. They are particularly useful when you are generating labels in macros.

The scope of numeric local labels is limited by the `AREA` directive. Use the `ROUT` directive to limit the scope of numeric local labels more tightly. A reference to a numeric local label refers to a matching label within the same scope. If there is no matching label within the scope in either direction, `armasm` generates an error message and the assembly fails.

You can use the same number for more than one numeric local label even within the same scope. By default, `armasm` links a numeric local label reference to:

- The most recent numeric local label with the same number, if there is one within the scope.
- The next following numeric local label with the same number, if there is not a preceding one within the scope.

Use the optional parameters to modify this search pattern if required.

Related concepts

12.6 Labels on page 12-303.

Related references

5.1 Syntax of source lines in assembly language on page 5-94.
12.11 Syntax of numeric local labels on page 12-308.
21.51 MACRO and MEND on page 21-1700.
21.48 KEEP on page 21-1697.
21.62 ROUT on page 21-1715.

12.11 Syntax of numeric local labels

When referring to numeric local labels you can specify how `armasm` searches for the label.

Syntax

```
n[routname] ; a numeric local label
%[F|B][A|T]n[routname] ; a reference to a numeric local label
```

where:

n
: is the number of the numeric local label in the range 0-99.

routname
: is the name of the current scope.

%
: introduces the reference.

F
: instructs `armasm` to search forwards only.

B
: instructs `armasm` to search backwards only.

A
: instructs `armasm` to search all macro levels.

T
: instructs `armasm` to look at this macro level only.

Usage

If neither `F` nor `B` is specified, `armasm` searches backwards first, then forwards.

If neither `A` nor `T` is specified, `armasm` searches all macros from the current level to the top level, but does not search lower level macros.

If *routname* is specified in either a label or a reference to a label, `armasm` checks it against the name of the nearest preceding `ROUT` directive. If it does not match, `armasm` generates an error message and the assembly fails.

Related concepts

12.10 Numeric local labels on page 12-307.

Related references

21.62 ROUT on page 21-1715.

12.12 String expressions

String expressions consist of combinations of string literals, string variables, string manipulation operators, and parentheses.

Characters that cannot be placed in string literals can be placed in string expressions using the `:CHR:` unary operator. Any ASCII character from 0 to 255 is permitted.

The value of a string expression cannot exceed 5120 characters in length. It can be of zero length.

Example

```
improb  SETS    "literal":CC:(strvar2:LEFT:4)
            ; sets the variable improb to the value "literal"
            ; with the left-most four characters of the
            ; contents of string variable strvar2 appended
```

Related concepts

12.13 String literals on page 12-310.

12.19 Unary operators on page 12-316.

12.2 Variables on page 12-299.

Related references

12.22 String manipulation operators on page 12-319.

21.63 SETA, SETL, and SETS on page 21-1716.

12.13 String literals

String literals consist of a series of characters or spaces contained between double quote characters.

The length of a string literal is restricted by the length of the input line.

To include a double quote character or a dollar character within the string literal, include the character twice as a pair. For example, you must use `$$` if you require a single `$` in the string.

C string escape sequences are also enabled and can be used within the string, unless `--no_esc` is specified.

Examples

```
abc     SETS    "this string contains only one "" double quote"
def     SETS    "this string contains only one $$ dollar symbol"
```

Related references

5.1 Syntax of source lines in assembly language on page 5-94.
11.47 --no_esc on page 11-275.

12.14 Numeric expressions

Numeric expressions consist of combinations of numeric constants, numeric variables, ordinary numeric literals, binary operators, and parentheses.

Numeric expressions can contain register-relative or program-relative expressions if the overall expression evaluates to a value that does not include a register or the PC.

Numeric expressions evaluate to 32-bit integers in A32 and T32 code. You can interpret them as unsigned numbers in the range 0 to 2^{32}-1, or signed numbers in the range -2^{31} to 2^{31}-1. However, `armasm` makes no distinction between -*n* and 2^{32}-*n*. Relational operators such as >= use the unsigned interpretation. This means that 0 > -1 is `{FALSE}`.

In A64 code, numeric expressions evaluate to 64-bit integers. You can interpret them as unsigned numbers in the range 0 to 2^{64}-1, or signed numbers in the range -2^{63} to 2^{63}-1. However, `armasm` makes no distinction between -*n* and 2^{64}-*n*.

Note

`armasm` does not support 64-bit arithmetic variables. See *21.63 SETA, SETL, and SETS* on page 21-1716 (Restrictions) for a workaround.

Arm recommends that you only use `armasm` for legacy Arm syntax assembly code, and that you use the `armclang` assembler and GNU syntax for all new assembly files.

Example

```
a   SETA    256*256          ; 256*256 is a numeric expression
    MOV     r1,#(a*22)       ; (a*22) is a numeric expression
```

Related concepts

12.20 Binary operators on page 12-317.

12.2 Variables on page 12-299.

12.3 Numeric constants on page 12-300.

Related references

12.15 Syntax of numeric literals on page 12-312.

21.63 SETA, SETL, and SETS on page 21-1716.

12.15 Syntax of numeric literals

Numeric literals consist of a sequence of characters, or a single character in quotes, evaluating to an integer.

They can take any of the following forms:

- *decimal-digits*.
- *0xhexadecimal-digits*.
- *&hexadecimal-digits*.
- *n_base-n-digits*.
- *'character'*.

where:

decimal-digits
: Is a sequence of characters using only the digits 0 to 9.

hexadecimal-digits
: Is a sequence of characters using only the digits 0 to 9 and the letters A to F or a to f.

*n_*
: Is a single digit between 2 and 9 inclusive, followed by an underscore character.

base-n-digits
: Is a sequence of characters using only the digits 0 to (*n*-1)

character
: Is any single character except a single quote. Use the standard C escape character (\') if you require a single quote. The character must be enclosed within opening and closing single quotes. In this case, the value of the numeric literal is the numeric code of the character.

You must not use any other characters. The sequence of characters must evaluate to an integer.

In A32/T32 code, the range is 0 to 2^{32}-1, except in `DCQ`, `DCQU`, `DCD`, and `DCDU` directives.

In A64 code, the range is 0 to 2^{64}-1, except in `DCD` and `DCDU` directives.

Note

- In the `DCQ` and `DCQU`, the integer range is 0 to 2^{64}-1
- In the `DCO` and `DCOU` directives, the integer range is 0 to 2^{128}-1

Examples

```
a       SETA    34906
addr    DCD     0xA10E
        LDR     r4,=&1000000F
        DCD     2_11001010
c3      SETA    8_74007
        DCQ     0x0123456789abcdef
        LDR     r1,='A'      ; pseudo-instruction loading 65 into r1
        ADD     r3,r2,#'\''  ; add 39 to contents of r2, result to r3
```

Related concepts

12.3 Numeric constants on page 12-300.

12.16 Syntax of floating-point literals

Floating-point literals consist of a sequence of characters evaluating to a floating-point number.

They can take any of the following forms:

- {-}*digits*E{-}*digits*
- {-}{*digits*}.digits
- {-}{*digits*}.digitsE{-}*digits*
- 0x*hexdigits*
- &*hexdigits*
- 0f_*hexdigits*
- 0d_*hexdigits*

where:

digits
: Are sequences of characters using only the digits 0 to 9. You can write E in uppercase or lowercase. These forms correspond to normal floating-point notation.

hexdigits
: Are sequences of characters using only the digits 0 to 9 and the letters A to F or a to f. These forms correspond to the internal representation of the numbers in the computer. Use these forms to enter infinities and NaNs, or if you want to be sure of the exact bit patterns you are using.

The 0x and & forms allow the floating-point bit pattern to be specified by any number of hex digits.

The 0f_ form requires the floating-point bit pattern to be specified by exactly 8 hex digits.

The 0d_ form requires the floating-point bit pattern to be specified by exactly 16 hex digits.

The range for half-precision floating-point values is:

- Maximum 65504 (IEEE format) or 131008 (alternative format).
- Minimum 0.00012201070785522461.

The range for single-precision floating-point values is:

- Maximum 3.40282347e+38.
- Minimum 1.17549435e-38.

The range for double-precision floating-point values is:

- Maximum 1.79769313486231571e+308.
- Minimum 2.22507385850720138e-308.

Floating-point numbers are only available if your system has floating-point, Advanced SIMD with floating-point.

Examples

```
    DCFD    1E308,-4E-100
    DCFS    1.0
    DCFS    0.02
    DCFD    3.725e15
    DCFS    0x7FC00000              ; Quiet NaN
    DCFD    &FFF0000000000000       ; Minus infinity
```

Related concepts

12.3 Numeric constants on page 12-300.

Related references

12.15 Syntax of numeric literals on page 12-312.

12.17 Logical expressions

Logical expressions consist of combinations of logical literals (`{TRUE}` or `{FALSE}`), logical variables, Boolean operators, relations, and parentheses.

Relations consist of combinations of variables, literals, constants, or expressions with appropriate relational operators.

Related references

12.26 Boolean operators on page 12-323.
12.25 Relational operators on page 12-322.

12.18 Logical literals

Logical or Boolean literals can have one of two values, `{TRUE}` or `{FALSE}`.

Related concepts

12.13 String literals on page 12-310.

Related references

12.15 Syntax of numeric literals on page 12-312.

12.19 Unary operators

Unary operators return a string, numeric, or logical value. They have higher precedence than other operators and are evaluated first.

A unary operator precedes its operand. Adjacent operators are evaluated from right to left.

The following table lists the unary operators that return strings:

Table 12-1 Unary operators that return strings

Operator	Usage	Description
`:CHR:`	`:CHR:A`	Returns the character with ASCII code A.
`:LOWERCASE:`	`:LOWERCASE:string`	Returns the given string, with all uppercase characters converted to lowercase.
`:REVERSE_CC:`	`:REVERSE_CC:cond_code`	Returns the inverse of the condition code in `cond_code`, or an error if `cond_code` does not contain a valid condition code.
`:STR:`	`:STR:A`	In A32 and T32 code, returns an 8-digit hexadecimal string corresponding to a numeric expression, or the string `"T"` or `"F""` if used on a logical expression. In A64 code, returns a 16-digit hexadecimal string.
`:UPPERCASE:`	`:UPPERCASE:string`	Returns the given string, with all lowercase characters converted to uppercase.

The following table lists the unary operators that return numeric values:

Table 12-2 Unary operators that return numeric or logical values

Operator	Usage	Description
`?`	`?A`	Number of bytes of code generated by line defining symbol A.
+ and -	`+A` `-A`	Unary plus. Unary minus. + and – can act on numeric and PC-relative expressions.
`:BASE:`	`:BASE:A`	If A is a PC-relative or register-relative expression, `:BASE:` returns the number of its register component. `:BASE:` is most useful in macros.
`:CC_ENCODING:`	`:CC_ENCODING:cond_code`	Returns the numeric value of the condition code in `cond_code`, or an error if `cond_code` does not contain a valid condition code.
`:DEF:`	`:DEF:A`	{`TRUE`} if A is defined, otherwise {`FALSE`}.
`:INDEX:`	`:INDEX:A`	If A is a register-relative expression, `:INDEX:` returns the offset from that base register. `:INDEX:` is most useful in macros.
`:LEN:`	`:LEN:A`	Length of string A.
`:LNOT:`	`:LNOT:A`	Logical complement of A.
`:NOT:`	`:NOT:A`	Bitwise complement of A (~ is an alias, for example `~A`).
`:RCONST:`	`:RCONST:Rn`	Number of register. In A32/T32 code, 0-15 corresponds to R0-R15. In A64 code, 0-30 corresponds to W0-W30 or X0-X30.

Related concepts

12.20 Binary operators on page 12-317.

12.20 Binary operators

You write binary operators between the pair of sub-expressions they operate on. They have lower precedence than unary operators.

Note

The order of precedence is not the same as in C.

Related concepts

12.28 Difference between operator precedence in assembly language and C on page 12-325.

Related references

12.21 Multiplicative operators on page 12-318.
12.22 String manipulation operators on page 12-319.
12.23 Shift operators on page 12-320.
12.24 Addition, subtraction, and logical operators on page 12-321.
12.25 Relational operators on page 12-322.
12.26 Boolean operators on page 12-323.

12.21 Multiplicative operators

Multiplicative operators have the highest precedence of all binary operators. They act only on numeric expressions.

The following table shows the multiplicative operators:

Table 12-3 Multiplicative operators

Operator	Alias	Usage	Explanation
*		A*B	Multiply
/		A/B	Divide
:MOD:	%	A:MOD:B	A modulo B

You can use the `:MOD:` operator on PC-relative expressions to ensure code is aligned correctly. These alignment checks have the form *`PC-relative`*`:MOD:`*`Constant`*. For example:

```
    AREA x,CODE
    ASSERT ({PC}:MOD:4) == 0
    DCB 1
y   DCB 2
    ASSERT (y:MOD:4) == 1
    ASSERT ({PC}:MOD:4) == 2
    END
```

Related concepts

12.20 Binary operators on page 12-317.

12.5 Register-relative and PC-relative expressions on page 12-302.

12.14 Numeric expressions on page 12-311.

Related references

12.15 Syntax of numeric literals on page 12-312.

12.22 String manipulation operators

You can use string manipulation operators to concatenate two strings, or to extract a substring.

The following table shows the string manipulation operators. In `CC`, both `A` and `B` must be strings. In the slicing operators `LEFT` and `RIGHT`:

- `A` must be a string.
- `B` must be a numeric expression.

Table 12-4 String manipulation operators

Operator	Usage	Explanation
`:CC:`	`A:CC:B`	B concatenated onto the end of A
`:LEFT:`	`A:LEFT:B`	The left-most B characters of A
`:RIGHT:`	`A:RIGHT:B`	The right-most B characters of A

Related concepts

12.12 String expressions on page 12-309.
12.14 Numeric expressions on page 12-311.

12.23 Shift operators

Shift operators act on numeric expressions, by shifting or rotating the first operand by the amount specified by the second.

The following table shows the shift operators:

Table 12-5 Shift operators

Operator	Alias	Usage	Explanation
`:ROL:`		`A:ROL:B`	Rotate A left by B bits
`:ROR:`		`A:ROR:B`	Rotate A right by B bits
`:SHL:`	`<<`	`A:SHL:B`	Shift A left by B bits
`:SHR:`	`>>`	`A:SHR:B`	Shift A right by B bits

Note

`SHR` is a logical shift and does not propagate the sign bit.

Related concepts

12.20 Binary operators on page 12-317.

12.24 Addition, subtraction, and logical operators

Addition, subtraction, and logical operators act on numeric expressions.

Logical operations are performed bitwise, that is, independently on each bit of the operands to produce the result.

The following table shows the addition, subtraction, and logical operators:

Table 12-6 Addition, subtraction, and logical operators

Operator	Alias	Usage	Explanation
+		A+B	Add A to B
-		A-B	Subtract B from A
:AND:	&	A:AND:B	Bitwise AND of A and B
:EOR:	^	A:EOR:B	Bitwise Exclusive OR of A and B
:OR:		A:OR:B	Bitwise OR of A and B

The use of | as an alias for :OR: is deprecated.

Related concepts

12.20 Binary operators on page 12-317.

12.25 Relational operators

Relational operators act on two operands of the same type to produce a logical value.

The operands can be one of:

- Numeric.
- PC-relative.
- Register-relative.
- Strings.

Strings are sorted using ASCII ordering. String `A` is less than string `B` if it is a leading substring of string `B`, or if the left-most character in which the two strings differ is less in string `A` than in string `B`.

Arithmetic values are unsigned, so the value of `0>-1` is `{FALSE}`.

The following table shows the relational operators:

Table 12-7 Relational operators

Operator	Alias	Usage	Explanation
`=`	`==`	`A=B`	A equal to B
`>`		`A>B`	A greater than B
`>=`		`A>=B`	A greater than or equal to B
`<`		`A<B`	A less than B
`<=`		`A<=B`	A less than or equal to B
`/=`	`<> !=`	`A/=B`	A not equal to B

Related concepts

12.20 Binary operators on page 12-317.

12.26 Boolean operators

Boolean operators perform standard logical operations on their operands. They have the lowest precedence of all operators.

In all three cases, both A and B must be expressions that evaluate to either `{TRUE}` or `{FALSE}`.

The following table shows the Boolean operators:

Table 12-8 Boolean operators

Operator	Alias	Usage	Explanation
`:LAND:`	&&	`A:LAND:B`	Logical AND of A and B
`:LEOR:`		`A:LEOR:B`	Logical Exclusive OR of A and B
`:LOR:`	||	`A:LOR:B`	Logical OR of A and B

Related concepts

12.20 Binary operators on page 12-317.

12.27 Operator precedence

`armasm` includes an extensive set of operators for use in expressions. It evaluates them using a strict order of precedence.

Many of the operators resemble their counterparts in high-level languages such as C.

`armasm` evaluates operators in the following order:

1. Expressions in parentheses are evaluated first.
2. Operators are applied in precedence order.
3. Adjacent unary operators are evaluated from right to left.
4. Binary operators of equal precedence are evaluated from left to right.

Related concepts

12.19 Unary operators on page 12-316.

12.20 Binary operators on page 12-317.

12.28 Difference between operator precedence in assembly language and C on page 12-325.

Related references

12.21 Multiplicative operators on page 12-318.

12.22 String manipulation operators on page 12-319.

12.23 Shift operators on page 12-320.

12.24 Addition, subtraction, and logical operators on page 12-321.

12.25 Relational operators on page 12-322.

12.26 Boolean operators on page 12-323.

12.28 Difference between operator precedence in assembly language and C

`armasm` does not follow exactly the same order of precedence when evaluating operators as a C compiler.

For example, `(1 + 2 :SHR: 3)` evaluates as `(1 + (2 :SHR: 3))` = 1 in assembly language. The equivalent expression in C evaluates as `((1 + 2) >> 3)` = 0.

Arm recommends you use brackets to make the precedence explicit.

If your code contains an expression that would parse differently in C, and you are not using the `--unsafe` option, `armasm` gives a warning:

```
A1466W: Operator precedence means that expression would evaluate differently in C
```

In the following tables:
- The highest precedence operators are at the top of the list.
- The highest precedence operators are evaluated first.
- Operators of equal precedence are evaluated from left to right.

The following table shows the order of precedence of operators in assembly language, and a comparison with the order in C.

Table 12-9 Operator precedence in Arm assembly language

assembly language precedence	equivalent C operators
unary operators	unary operators
`* / :MOD:`	`* / %`
string manipulation	n/a
`:SHL: :SHR: :ROR: :ROL:`	`<< >>`
`+ - :AND: :OR: :EOR:`	`+ - & | ^`
`= > >= < <= /= <>`	`== > >= < <= !=`
`:LAND: :LOR: :LEOR:`	`&& ||`

The following table shows the order of precedence of operators in C.

Table 12-10 Operator precedence in C

C precedence
unary operators
`* / %`
`+ -` (as binary operators)
`<< >>`
`< <= > >=`
`== !=`
`&`
`^`
`|`
`&&`
`||`

Related concepts

12.20 Binary operators on page 12-317.

Related references

12.27 Operator precedence on page 12-324.

Chapter 13
A32 and T32 Instructions

Describes the A32 and T32 instructions supported in AArch32 state.

It contains the following sections:

13.1 A32 and T32 instruction summary

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

Table 13-1 Summary of instructions

Mnemonic	Brief description
ADC, ADD	Add with Carry, Add
ADR	Load program or register-relative address (short range)
ADRL pseudo-instruction	Load program or register-relative address (medium 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
CPY pseudo-instruction	Copy
CRC32	CRC32
CRC32C	CRC32C
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

Table 13-1 Summary of instructions (continued)

Mnemonic	Brief description
HLT	Halting breakpoint
HVC	Hypervisor Call
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
LDR pseudo-instruction	Load Register pseudo-instruction
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
MOV32 pseudo-instruction	Move 32-bit immediate to register
MRC	Move from Coprocessor to Register
MRRC	Move from Coprocessor to Registers
MRS	Move from PSR to Register
MRS pseudo-instruction	Move from system Coprocessor to Register
MSR	Move from Register to PSR
MSR pseudo-instruction	Move from Register to system Coprocessor

Table 13-1 Summary of instructions (continued)

Mnemonic	Brief description
MUL	Multiply
MVN	Move Not
NEG pseudo-instruction	Negate
NOP	No Operation
ORN	Logical OR NOT
ORR	Logical OR
PKHBT, PKHTB	Pack Halfwords
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

Table 13-1 Summary of instructions (continued)

Mnemonic	Brief description
	(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)
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

Table 13-1 Summary of instructions (continued)

Mnemonic	Brief description
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
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 14 Advanced SIMD Instructions (32-bit)* on page 14-599 and *Chapter 15 Floating-point Instructions (32-bit)* on page 15-749
WFE, WFI, YIELD	Wait For Event, Wait For Interrupt, Yield

13.2 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
```

13.3 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

13.6 Shift operations on page 13-341.

Related references

13.4 Syntax of Operand2 as a constant on page 13-339.

13.5 Syntax of Operand2 as a register with optional shift on page 13-340.

13.4 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.

You can use the `--diag_warning 1645` assembler command line option to check when an instruction substitution occurs.

Related concepts

13.6 Shift operations on page 13-341.

Related references

13.3 Flexible second operand (Operand2) on page 13-338.
13.5 Syntax of Operand2 as a register with optional shift on page 13-340.

13.5 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

13.6 Shift operations on page 13-341.

Related references

13.3 Flexible second operand (Operand2) on page 13-338.
13.4 Syntax of Operand2 as a constant on page 13-339.

13.6 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*.

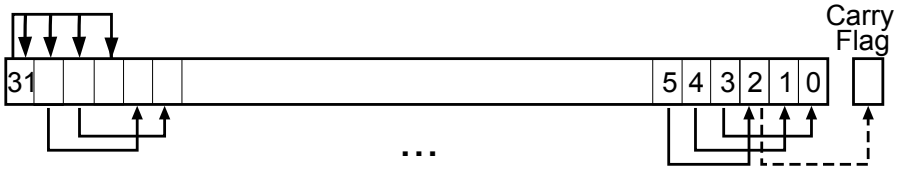


Figure 13-1 ASR #3

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.

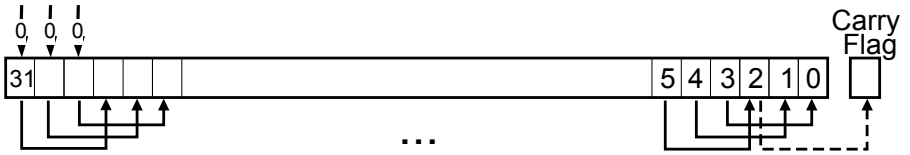


Figure 13-2 LSR #3

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.

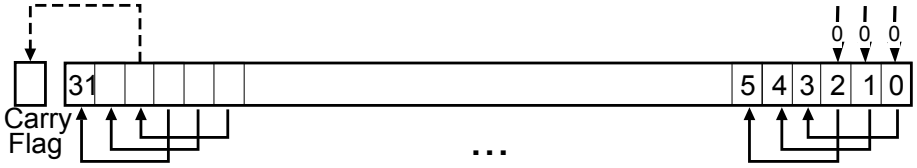


Figure 13-3 LSL #3

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.

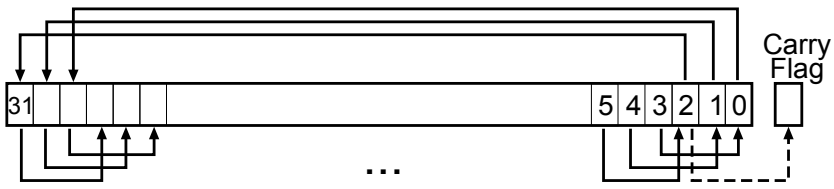


Figure 13-4 ROR #3

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`*.

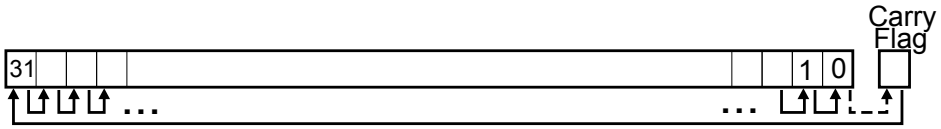


Figure 13-5 RRX

Related references

13.3 Flexible second operand (Operand2) on page 13-338.
13.4 Syntax of Operand2 as a constant on page 13-339.
13.5 Syntax of Operand2 as a register with optional shift on page 13-340.

13.7 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 concepts

9.14 Saturating Advanced SIMD instructions on page 9-198.

Related references

13.82 QADD on page 13-457.
13.89 QSUB on page 13-464.
13.86 QDADD on page 13-461.
13.87 QDSUB on page 13-462.
13.120 SMLAxy on page 13-501.
13.125 SMLAWy on page 13-507.
13.132 SMULxy on page 13-514.
13.134 SMULWy on page 13-516.
13.137 SSAT on page 13-520.
13.189 USAT on page 13-584.
13.71 MSR (general-purpose register to PSR) on page 13-441.

13.8 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

13.3 Flexible second operand (Operand2) on page 13-338.
7.11 Condition code suffixes on page 7-150.

13.9 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

13.3 Flexible second operand (Operand2) on page 13-338.
7.11 Condition code suffixes on page 7-150.
13.151 SUBS pc, lr on page 13-542.

13.10 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.

Use the `ADRL` pseudo-instruction to assemble a wider range of effective addresses.

`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 13-2 PC-relative offsets

Instruction	Offset range
A32 `ADR`	See *13.4 Syntax of Operand2 as a constant* on page 13-339.
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`.

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

In A32 code, R*d* can be PC or SP but use of SP is deprecated.

Related concepts

6.10 Load addresses to a register using ADR on page 6-114.
12.5 Register-relative and PC-relative expressions on page 12-302.

Related references

13.4 Syntax of Operand2 as a constant on page 13-339.
13.12 ADRL pseudo-instruction on page 13-353.
21.6 AREA on page 21-1650.
7.11 Condition code suffixes on page 7-150.

13.11 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.

Use the `ADRL` pseudo-instruction to assemble a wider range of effective addresses.

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 13-3 Register-relative offsets

Instruction	Offset range
A32 `ADR`	See *13.4 Syntax of Operand2 as a constant* on page 13-339
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.

[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.

Related concepts

12.5 Register-relative and PC-relative expressions on page 12-302.

Related references

13.4 Syntax of Operand2 as a constant on page 13-339.
13.12 ADRL pseudo-instruction on page 13-353.
21.52 MAP on page 21-1703.
21.29 FIELD on page 21-1676.
7.11 Condition code suffixes on page 7-150.

13.12 ADRL pseudo-instruction

Load a PC-relative or register-relative address into a register.

Syntax

`ADRL{cond} Rd,label`

where:

cond
: is an optional condition code.

Rd
: is the register to load.

label
: is a PC-relative or register-relative expression.

Usage

`ADRL` always assembles to two 32-bit instructions. Even if the address can be reached in a single instruction, a second, redundant instruction is produced.

If the assembler cannot construct the address in two instructions, it generates an error message and the assembly fails. You can use the `LDR` pseudo-instruction for loading a wider range of addresses.

`ADRL` is similar to the `ADR` instruction, except `ADRL` can load a wider range of addresses because it generates two data processing instructions.

`ADRL` produces position-independent code, because the address is PC-relative or register-relative.

If *label* is PC-relative, it must evaluate to an address in the same assembler area as the `ADRL` pseudo-instruction.

If you use `ADRL` 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.

Architectures and range

The available range depends on the instruction set in use:

A32
: The range of the instruction is any value that can be generated by two `ADD` or two `SUB` instructions. That is, any value that can be produced by the addition of two values, each of which is 8 bits rotated right by any even number of bits within a 32-bit word.

T32, 32-bit encoding
: ±1MB bytes to a byte, halfword, or word-aligned address.

T32, 16-bit encoding
: `ADRL` is not available.

The given range is relative to a point four bytes (in T32 code) or two words (in A32 code) after the address of the current instruction.

Note

`ADRL` is not available in Armv6-M and Armv8-M.baseline.

Related concepts

12.5 Register-relative and PC-relative expressions on page 12-302.
6.4 Load immediate values on page 6-105.

Related references

13.4 Syntax of Operand2 as a constant on page 13-339.

13.54 LDR pseudo-instruction on page 13-417.
21.6 AREA on page 21-1650.
13.9 ADD on page 13-347.
7.11 Condition code suffixes on page 7-150.

Related information

Arm Architecture Reference Manual.

13.13 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

13.3 Flexible second operand (Operand2) on page 13-338.

13.151 SUBS pc, lr on page 13-542.

7.11 Condition code suffixes on page 7-150.

13.14 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

13.63 MOV on page 13-431.
7.11 Condition code suffixes on page 7-150.

13.15 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 13-4 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 concepts

12.5 Register-relative and PC-relative expressions on page 12-302.

Related references

7.11 Condition code suffixes on page 7-150.

Related information

Information about image structure and generation.

13.16 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

7.11 Condition code suffixes on page 7-150.

13.17 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

7.11 Condition code suffixes on page 7-150.

13.18 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

13.3 Flexible second operand (Operand2) on page 13-338.

13.151 SUBS pc, lr on page 13-542.

7.11 Condition code suffixes on page 7-150.

13.19 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.

13.20 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 13-5 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 concepts

12.5 Register-relative and PC-relative expressions on page 12-302.

Related references

7.11 Condition code suffixes on page 7-150.

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

Related information

Information about image structure and generation.

13.21 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

- There are no equivalent instructions to `BLX` to change between AArch32 and AArch64 state. The only way to change execution state is by a change of exception level.
- Armv8-M, 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 13-6 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

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

Table 13-6 BLX instruction availability and range (continued)

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

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 concepts

12.5 Register-relative and PC-relative expressions on page 12-302.

Related references

7.11 Condition code suffixes on page 7-150.

13.2 Instruction width specifiers on page 13-337.

Related information

Information about image structure and generation.

13.22 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

- There are no equivalent instructions to `BX` to change between AArch32 and AArch64 state. The only way to change execution state is by a change of exception level.
- Armv8-M, 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 13-7 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 concepts

12.5 Register-relative and PC-relative expressions on page 12-302.

Related references

7.11 Condition code suffixes on page 7-150.
13.2 Instruction width specifiers on page 13-337.

Related information

Information about image structure and generation.

13.23 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 13-8 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 concepts

12.5 Register-relative and PC-relative expressions on page 12-302.

Related references

7.11 Condition code suffixes on page 7-150.

Related information

Information about image structure and generation.

13.24 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

13.15 B on page 13-359.
13.28 CMP and CMN on page 13-377.
13.2 Instruction width specifiers on page 13-337.

13.25 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

7.11 Condition code suffixes on page 7-150.

13.26 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

7.11 Condition code suffixes on page 7-150.

Related information

Arm Architecture Reference Manual.

13.27 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

7.11 Condition code suffixes on page 7-150.

13.28 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

13.3 Flexible second operand (Operand2) on page 13-338.
7.11 Condition code suffixes on page 7-150.

13.29 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

3.2 Processor modes, and privileged and unprivileged software execution on page 3-65.

13.30 CPY pseudo-instruction

Copy a value from one register to another.

Syntax

`CPY{cond} Rd, Rm`

where:

cond
: is an optional condition code.

Rd
: is the destination register.

Rm
: is the register holding the value to be copied.

Operation

The `CPY` pseudo-instruction copies a value from one register to another, without changing the condition flags.

`CPY Rd, Rm` assembles to `MOV Rd, Rm`.

Architectures

This pseudo-instruction is available in A32 code and in T32 code.

Register restrictions

Using SP or PC for both *Rd* and *Rm* is deprecated.

Condition flags

This instruction does not change the condition flags.

Related references

13.63 MOV on page 13-431.

13.31 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 *13.2 Instruction width specifiers* on page 13-337. 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

13.31 CRC32 on page 13-382.
13.1 A32 and T32 instruction summary on page 13-332.

13.32 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 *13.2 Instruction width specifiers* on page 13-337. 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 Armv8.1 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

13.31 CRC32 on page 13-382.
13.1 A32 and T32 instruction summary on page 13-332.

13.33 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

13.75 NOP on page 13-447.
7.11 Condition code suffixes on page 7-150.

13.34 DCPS1 (T32 instruction)

Debug switch to exception level 1 (EL1).

Note

This instruction is supported only in Armv8.

Syntax

```
DCPS1
```

Usage

This instruction is valid in Debug state only, and is always UNDEFINED in Non-debug state.

DCPS1 targets EL1 and:

- If EL1 is using AArch32, the processing element (PE) enters SVC mode. If EL3 is using AArch32, Secure SVC is an EL3 mode. This means DCPS1 causes the PE to enter EL3.
- If EL1 is using AArch64, the PE enters EL1h, and executes future instructions as A64 instructions.

In Non-debug state, use the SVC instruction to generate a trap to EL1.

Availability

This 32-bit instruction is available in T32 only.

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

Related references

13.152 SVC on page 13-544.
13.35 DCPS2 (T32 instruction) on page 13-386.
13.36 DCPS3 (T32 instruction) on page 13-387.

Related information

Arm Architecture Reference Manual.

13.35 DCPS2 (T32 instruction)

Debug switch to exception level 2.

Note

This instruction is supported only in Armv8.

Syntax

```
DCPS2
```

Usage

This instruction is valid in Debug state only, and is always UNDEFINED in Non-debug state.

DCPS2 targets EL2 and:

- If EL2 is using AArch32, the PE enters Hyp mode.
- If EL2 is using AArch64, the PE enters EL2h, and executes future instructions as A64 instructions.

In Non-debug state, use the HVC instruction to generate a trap to EL2.

Availability

This 32-bit instruction is available in T32 only.

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

Related references

13.43 HVC on page 13-397.
13.34 DCPS1 (T32 instruction) on page 13-385.
13.36 DCPS3 (T32 instruction) on page 13-387.

Related information

Arm Architecture Reference Manual.

13.36 DCPS3 (T32 instruction)

Debug switch to exception level 3.

Note

This instruction is supported only in Armv8.

Syntax

```
DCPS3
```

Usage

This instruction is valid in Debug state only, and is always UNDEFINED in Non-debug state.

DCPS3 targets EL3 and:

- If EL3 is using AArch32, the PE enters Monitor mode.
- If EL3 is using AArch64, the PE enters EL3h, and executes future instructions as A64 instructions.

In Non-debug state, use the SMC instruction to generate a trap to EL3.

Availability

This 32-bit instruction is available in T32 only.

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

Related references

13.119 SMC on page 13-500.
13.35 DCPS2 (T32 instruction) on page 13-386.
13.34 DCPS1 (T32 instruction) on page 13-385.

Related information

Arm Architecture Reference Manual.

13.37 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

7.11 Condition code suffixes on page 7-150.

13.38 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

7.11 Condition code suffixes on page 7-150.

13.39 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

13.3 Flexible second operand (Operand2) on page 13-338.

13.151 SUBS pc, lr on page 13-542.

7.11 Condition code suffixes on page 7-150.

13.40 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

3.2 Processor modes, and privileged and unprivileged software execution on page 3-65.

Related references

13.63 MOV on page 13-431.
13.151 SUBS pc, lr on page 13-542.
7.11 Condition code suffixes on page 7-150.
13.43 HVC on page 13-397.

13.41 ESB

Error Synchronization Barrier.

Syntax

```
ESB{c}{q} ; A1 general registers (A32)

ESB{c}.W ; T1 general registers (T32)
```

Where:

q

Is an optional instruction width specifier. See *13.2 Instruction width specifiers* on page 13-337.

c

Is an optional instruction condition code. See *Chapter 7 Condition Codes* on page 7-139.

Architectures supported

Supported in the Armv8-A and Armv8-R architectures.

Usage

Error Synchronization Barrier.

Related references

13.1 A32 and T32 instruction summary on page 13-332.

13.42 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

13.75 NOP on page 13-447.

13.43 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

3.2 Processor modes, and privileged and unprivileged software execution on page 3-65.

Related references

13.40 ERET on page 13-394.

13.44 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

7.11 Condition code suffixes on page 7-150.

13.45 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 13-9 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
```

13.46 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

13.47 LDAEX on page 13-403.
13.143 STL on page 13-527.
13.144 STLEX on page 13-528.
7.11 Condition code suffixes on page 7-150.

13.47 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 concepts

8.15 Address alignment in A32/T32 code on page 8-178.

Related references

13.143 STL on page 13-527.
13.46 LDA on page 13-402.
13.144 STLEX on page 13-528.
7.11 Condition code suffixes on page 7-150.

13.48 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 concepts

12.5 Register-relative and PC-relative expressions on page 12-302.

8.15 Address alignment in A32/T32 code on page 8-178.

Related references

7.11 Condition code suffixes on page 7-150.

13.49 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.

You can use the `--diag_warning 1645` assembler command line option to check when an instruction substitution occurs.

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 concepts

6.16 Stack implementation using LDM and STM on page 6-122.
8.15 Address alignment in A32/T32 code on page 8-178.

Related references

13.80 POP on page 13-455.
7.11 Condition code suffixes on page 7-150.

13.50 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 13-10 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

[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.

Table 13-10 Offsets and architectures, LDR, word, halfword, and byte (continued)

Instruction	Immediate offset	Pre-indexed	Post-indexed
T32 32-bit encoding, doubleword	-1020 to 1020 [i]	-1020 to 1020 [i]	-1020 to 1020 [i]
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 concepts

8.15 Address alignment in A32/T32 code on page 8-178.

Related references

7.11 Condition code suffixes on page 7-150.

[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.

13.51 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 13-11 PC-relative offsets

Instruction	Offset range
A32 LDR, LDRB, LDRSB, LDRH, LDRSH [m]	±4095
A32 LDRD	±255
32-bit T32 LDR, LDRB, LDRSB, LDRH, LDRSH [m]	±4095

[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.

Table 13-11 PC-relative offsets (continued)

Instruction	Offset range
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 concepts

12.5 Register-relative and PC-relative expressions on page 12-302.

8.15 Address alignment in A32/T32 code on page 8-178.

Related references

7.11 Condition code suffixes on page 7-150.

[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.

[p] Rt must be in the range R0-R7. There are no byte, halfword, or doubleword 16-bit instructions.

13.52 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 13-12 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

[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.

Table 13-12 Options and architectures, LDR (register offsets) (continued)

Instruction	±*Rm* [q]	shift		
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 concepts

8.15 Address alignment in A32/T32 code on page 8-178.

Related references

7.11 Condition code suffixes on page 7-150.

[q] Where ±Rm is shown, you can use –Rm, +Rm, or Rm. Where +Rm is shown, you cannot use –Rm.
[s] Rt, Rn, and Rm must all be in the range R0-R7.

13.53 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 13-13 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.
[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.

Table 13-13 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 concepts

12.5 Register-relative and PC-relative expressions on page 12-302.
8.15 Address alignment in A32/T32 code on page 8-178.

Related references

21.29 FIELD on page 21-1676.
21.52 MAP on page 21-1703.
7.11 Condition code suffixes on page 7-150.

13.54 LDR pseudo-instruction

Load a register with either a 32-bit immediate value or an address.

Note

This describes the `LDR` pseudo-instruction only, and not the `LDR` instruction.

Syntax

`LDR{cond}{.W} Rt, =expr`

`LDR{cond}{.W} Rt, =label_expr`

where:

cond
: is an optional condition code.

.W
: is an optional instruction width specifier.

Rt
: is the register to be loaded.

expr
: evaluates to a numeric value.

label_expr
: is a PC-relative or external expression of an address in the form of a label plus or minus a numeric value.

Usage

When using the `LDR` pseudo-instruction:

- If the value of *expr* can be loaded with a valid `MOV` or `MVN` instruction, the assembler uses that instruction.
- If a valid `MOV` or `MVN` instruction cannot be used, or if the `label_expr` syntax is used, the assembler places the constant in a literal pool and generates a PC-relative `LDR` instruction that reads the constant from the literal pool.

  Note

  — An address loaded in this way is fixed at link time, so the code is not position-independent.
  — The address holding the constant remains valid regardless of where the linker places the ELF section containing the `LDR` instruction.

The assembler places the value of *label_expr* in a literal pool and generates a PC-relative `LDR` instruction that loads the value from the literal pool.

If *label_expr* is an external expression, or is not contained in the current section, the assembler places a linker relocation directive in the object file. The linker generates the address at link time.

If *label_expr* is either a named or numeric local label, the assembler places a linker relocation directive in the object file and generates a symbol for that local label. The address is generated at link time. If the local label references T32 code, the T32 bit (bit 0) of the address is set.

The offset from the PC to the value in the literal pool must be less than ±4KB (in an A32 or 32-bit T32 encoding) or in the range 0 to +1KB (16-bit T32 encoding). You are responsible for ensuring that there is a literal pool within range.

If the label referenced is in T32 code, the LDR pseudo-instruction sets the T32 bit (bit 0) of *label_expr*.

Note

In *RealView® Compilation Tools* (RVCT) v2.2, the T32 bit of the address was not set. If you have code that relies on this behavior, use the command line option `--untyped_local_labels` to force the assembler not to set the T32 bit when referencing labels in T32 code.

LDR in T32 code

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 immediate value could be loaded in a 16-bit MOV, or there is a literal pool within reach of a 16-bit PC-relative load.

If the value to be loaded is not known in the first pass of the assembler, LDR without `.W` generates a 16-bit instruction in T32 code, even if that results in a 16-bit PC-relative load for a value that could be generated in a 32-bit MOV or MVN instruction. However, if the value is known in the first pass, and it can be generated using a 32-bit MOV or MVN instruction, the MOV or MVN instruction is used.

In UAL syntax, the LDR pseudo-instruction never generates a 16-bit flag-setting MOV instruction. Use the `--diag_warning 1727` assembler command line option to check when a 16-bit instruction could have been used.

You can use the MOV32 pseudo-instruction for generating immediate values or addresses without loading from a literal pool.

Examples

```
        LDR     r3,=0xff0    ; loads 0xff0 into R3
                             ; =>  MOV.W r3,#0xff0
        LDR     r1,=0xfff    ; loads 0xfff into R1
                             ; =>  LDR r1,[pc,offset_to_litpool]
                             ;     ...
                             ;     litpool DCD 0xfff
        LDR     r2,=place    ; loads the address of
                             ; place into R2
                             ; =>  LDR r2,[pc,offset_to_litpool]
                             ;     ...
                             ;     litpool DCD place
```

Related concepts

12.3 Numeric constants on page 12-300.
12.5 Register-relative and PC-relative expressions on page 12-302.
12.10 Numeric local labels on page 12-307.

Related references

11.62 --untyped_local_labels on page 11-290.
13.64 MOV32 pseudo-instruction on page 13-433.
7.11 Condition code suffixes on page 7-150.
21.50 LTORG on page 21-1699.

13.55 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 13-14 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 13-14 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 concepts

8.15 Address alignment in A32/T32 code on page 8-178.

Related references

7.11 Condition code suffixes on page 7-150.

13.56 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 Rn, 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 concepts

8.15 Address alignment in A32/T32 code on page 8-178.

Related references

7.11 Condition code suffixes on page 7-150.

13.57 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 `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 `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

13.63 MOV on page 13-431.
7.11 Condition code suffixes on page 7-150.

13.58 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

13.63 MOV on page 13-431.
7.11 Condition code suffixes on page 7-150.

13.59 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

7.11 Condition code suffixes on page 7-150.

13.60 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

7.11 Condition code suffixes on page 7-150.

13.61 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

7.11 Condition code suffixes on page 7-150.

13.62 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

7.11 Condition code suffixes on page 7-150.

13.63 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, Rm` when `Rm` is not PC or SP.
- `MOV Rd, PC` when `Rd` is not PC or SP.

You can use SP for `Rd` or `Rm`. But this is deprecated except for the following cases:

- `MOV SP, Rm` when `Rm` is not PC or SP.
- `MOV Rd, SP` when `Rd` 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 concepts

6.5 Load immediate values using MOV and MVN on page 6-106.

Related references

13.3 Flexible second operand (Operand2) on page 13-338.
13.151 SUBS pc, lr on page 13-542.
7.11 Condition code suffixes on page 7-150.

13.64 MOV32 pseudo-instruction

Load a register with either a 32-bit immediate value or any address.

Syntax

`MOV32{cond} Rd, expr`

where:

cond
: is an optional condition code.

Rd
: is the register to be loaded. *Rd* must not be SP or PC.

expr
: can be any one of the following:

  symbol
  : A label in this or another program area.

  #constant
  : Any 32-bit immediate value.

  symbol + constant
  : A label plus a 32-bit immediate value.

Usage

`MOV32` always generates two 32-bit instructions, a `MOV`, `MOVT` pair. This enables you to load any 32-bit immediate, or to access the whole 32-bit address space.

The main purposes of the `MOV32` pseudo-instruction are:

- To generate literal constants when an immediate value cannot be generated in a single instruction.
- To load a PC-relative or external address into a register. The address remains valid regardless of where the linker places the ELF section containing the `MOV32`.

  Note

  An address loaded in this way is fixed at link time, so the code is not position-independent.

`MOV32` sets the T32 bit (bit 0) of the address if the label referenced is in T32 code.

Architectures

This pseudo-instruction is available in A32 and T32.

Examples

```
    MOV32 r3, #0xABCDEF12  ; loads 0xABCDEF12 into R3
    MOV32 r1, Trigger+12   ; loads the address that is 12 bytes
                           ; higher than the address Trigger into R1
```

Related references

7.11 Condition code suffixes on page 7-150.

13.65 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. The assembler implements the `MOV32` pseudo-instruction for convenient generation of this 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

13.64 MOV32 pseudo-instruction on page 13-433.
7.11 Condition code suffixes on page 7-150.

13.66 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

7.11 Condition code suffixes on page 7-150.

13.67 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 `pn`, 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

7.11 Condition code suffixes on page 7-150.

13.68 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

3.12 Current Program Status Register in AArch32 state on page 3-76.

Related references

13.69 MRS (system coprocessor register to general-purpose register) on page 13-439.
13.70 MSR (general-purpose register to system coprocessor register) on page 13-440.
13.71 MSR (general-purpose register to PSR) on page 13-441.
7.11 Condition code suffixes on page 7-150.

13.69 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

13.68 MRS (PSR to general-purpose register) on page 13-437.

13.70 MSR (general-purpose register to system coprocessor register) on page 13-440.

13.71 MSR (general-purpose register to PSR) on page 13-441.

7.11 Condition code suffixes on page 7-150.

Related information

Arm Architecture Reference Manual.

13.70 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

13.68 MRS (PSR to general-purpose register) on page 13-437.
13.69 MRS (system coprocessor register to general-purpose register) on page 13-439.
13.71 MSR (general-purpose register to PSR) on page 13-441.
7.11 Condition code suffixes on page 7-150.
13.160 SYS on page 13-553.

Related information

Arm Architecture Reference Manual.

13.71 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. Rm must not be PC.

Syntax on architectures other than Arm®v6-M, Arm®v7-M, Arm®v8-M.baseline, and Arm®v8-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. Rm 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 Arm®v6-M, Arm®v7-M, Arm®v8-M.baseline, and Arm®v8-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

13.68 MRS (PSR to general-purpose register) on page 13-437.
13.69 MRS (system coprocessor register to general-purpose register) on page 13-439.
13.70 MSR (general-purpose register to system coprocessor register) on page 13-440.
7.11 Condition code suffixes on page 7-150.

13.72 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

7.11 Condition code suffixes on page 7-150.

13.73 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 concepts

6.5 Load immediate values using MOV and MVN on page 6-106.

Related references

13.3 Flexible second operand (Operand2) on page 13-338.
13.151 SUBS pc, lr on page 13-542.
7.11 Condition code suffixes on page 7-150.

13.74 NEG pseudo-instruction

Negate the value in a register.

Syntax

`NEG{cond} Rd, Rm`

where:

cond
: is an optional condition code.

Rd
: is the destination register.

Rm
: is the register containing the value that is subtracted from zero.

Operation

The `NEG` pseudo-instruction negates the value in one register and stores the result in a second register.

`NEG{cond} Rd, Rm` assembles to `RSBS{cond} Rd, Rm, #0`.

Architectures

The 32-bit encoding of this pseudo-instruction is available in A32 and T32.

There is no 16-bit encoding of this pseudo-instruction available T32.

Register restrictions

In A32 instructions, using SP or PC for *Rd* or *Rm* is deprecated. In T32 instructions, you cannot use SP or PC for *Rd* or *Rm*.

Condition flags

This pseudo-instruction updates the condition flags, based on the result.

Related references

13.9 ADD on page 13-347.

13.75 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

7.11 Condition code suffixes on page 7-150.

13.76 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

13.3 Flexible second operand (Operand2) on page 13-338.
13.151 SUBS pc, lr on page 13-542.
7.11 Condition code suffixes on page 7-150.

13.77 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

13.3 Flexible second operand (Operand2) on page 13-338.

13.151 SUBS pc, lr on page 13-542.

7.11 Condition code suffixes on page 7-150.

13.78 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

7.11 Condition code suffixes on page 7-150.

13.79 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 `NOP`s.

Related concepts

12.5 Register-relative and PC-relative expressions on page 12-302.

Related references

7.11 Condition code suffixes on page 7-150.

13.80 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

13.49 LDM on page 13-407.
13.81 PUSH on page 13-456.
7.11 Condition code suffixes on page 7-150.

13.81 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

13.49 LDM on page 13-407.
13.80 POP on page 13-455.
7.11 Condition code suffixes on page 7-150.

13.82 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 general-purpose 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

13.68 MRS (PSR to general-purpose register) on page 13-437.
3.10 The Q flag in AArch32 state on page 3-74.
7.11 Condition code suffixes on page 7-150.

13.83 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

3.10 The Q flag in AArch32 state on page 3-74.
7.11 Condition code suffixes on page 7-150.

13.84 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

3.10 The Q flag in AArch32 state on page 3-74.
7.11 Condition code suffixes on page 7-150.

13.85 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

3.10 The Q flag in AArch32 state on page 3-74.
7.11 Condition code suffixes on page 7-150.

13.86 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

3.10 The Q flag in AArch32 state on page 3-74.
13.68 MRS (PSR to general-purpose register) on page 13-437.
7.11 Condition code suffixes on page 7-150.

13.87 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

3.10 The Q flag in AArch32 state on page 3-74.
13.68 MRS (PSR to general-purpose register) on page 13-437.
7.11 Condition code suffixes on page 7-150.

13.88 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

3.10 The Q flag in AArch32 state on page 3-74.
7.11 Condition code suffixes on page 7-150.

13.89 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

3.10 The Q flag in AArch32 state on page 3-74.
13.68 MRS (PSR to general-purpose register) on page 13-437.
7.11 Condition code suffixes on page 7-150.

13.90 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

3.10 The Q flag in AArch32 state on page 3-74.
7.11 Condition code suffixes on page 7-150.

13.91 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

3.10 The Q flag in AArch32 state on page 3-74.
7.11 Condition code suffixes on page 7-150.

13.92 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

7.11 Condition code suffixes on page 7-150.

13.93 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

7.11 Condition code suffixes on page 7-150.

13.94 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

7.11 Condition code suffixes on page 7-150.

13.95 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

7.11 Condition code suffixes on page 7-150.

13.96 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

3.2 Processor modes, and privileged and unprivileged software execution on page 3-65.

Related references

13.136 SRS on page 13-518.
7.11 Condition code suffixes on page 7-150.

13.97 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 `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 `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

13.63 MOV on page 13-431.
7.11 Condition code suffixes on page 7-150.

13.98 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

13.63 MOV on page 13-431.
7.11 Condition code suffixes on page 7-150.

13.99 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

13.3 Flexible second operand (Operand2) on page 13-338.
7.11 Condition code suffixes on page 7-150.

13.100 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

13.3 Flexible second operand (Operand2) on page 13-338.
7.11 Condition code suffixes on page 7-150.

13.101 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

13.107 SEL on page 13-487.
7.11 Condition code suffixes on page 7-150.

13.102 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

13.107 SEL on page 13-487.
7.11 Condition code suffixes on page 7-150.

13.103 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

13.107 SEL on page 13-487.
7.11 Condition code suffixes on page 7-150.

13.104 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

13.3 Flexible second operand (Operand2) on page 13-338.
7.11 Condition code suffixes on page 7-150.

13.105 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

7.11 Condition code suffixes on page 7-150.

13.106 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

7.11 Condition code suffixes on page 7-150.

13.107 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

3.11 Application Program Status Register on page 3-75.

Related references

7.11 Condition code suffixes on page 7-150.

13.108 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
```

13.109 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 *13.2 Instruction width specifiers* on page 13-337.

imm
Is the unsigned immediate 0 or 1.

Architectures supported

Supported in the Armv8.1 architecture 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

13.1 A32 and T32 instruction summary on page 13-332.

13.110 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

13.111 SEVL on page 13-492.
13.75 NOP on page 13-447.
7.11 Condition code suffixes on page 7-150.

13.111 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

13.110 SEV on page 13-491.
13.75 NOP on page 13-447.
7.11 Condition code suffixes on page 7-150.

13.112 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.

13.113 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

7.11 Condition code suffixes on page 7-150.

13.114 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

7.11 Condition code suffixes on page 7-150.

13.115 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

7.11 Condition code suffixes on page 7-150.

13.116 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

7.11 Condition code suffixes on page 7-150.

13.117 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

7.11 Condition code suffixes on page 7-150.

13.118 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

7.11 Condition code suffixes on page 7-150.

13.119 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

7.11 Condition code suffixes on page 7-150.

Related information

Arm Architecture Reference Manual.

13.120 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

13.68 MRS (PSR to general-purpose register) on page 13-437.

13.71 MSR (general-purpose register to PSR) on page 13-441.
7.11 Condition code suffixes on page 7-150.

13.121 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

7.11 Condition code suffixes on page 7-150.

13.122 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

7.11 Condition code suffixes on page 7-150.

13.123 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

7.11 Condition code suffixes on page 7-150.

13.124 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

7.11 Condition code suffixes on page 7-150.

13.125 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

13.68 MRS (PSR to general-purpose register) on page 13-437.
7.11 Condition code suffixes on page 7-150.

13.126 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

7.11 Condition code suffixes on page 7-150.

13.127 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

7.11 Condition code suffixes on page 7-150.

13.128 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

7.11 Condition code suffixes on page 7-150.

13.129 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

7.11 Condition code suffixes on page 7-150.

13.130 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

7.11 Condition code suffixes on page 7-150.

13.131 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

7.11 Condition code suffixes on page 7-150.

13.132 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

13.68 MRS (PSR to general-purpose register) on page 13-437.
13.71 MSR (general-purpose register to PSR) on page 13-441.
7.11 Condition code suffixes on page 7-150.

13.133 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

7.11 Condition code suffixes on page 7-150.

13.134 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

13.68 MRS (PSR to general-purpose register) on page 13-437.
7.11 Condition code suffixes on page 7-150.

13.135 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

7.11 Condition code suffixes on page 7-150.

13.136 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

6.16 Stack implementation using LDM and STM on page 6-122.
3.2 Processor modes, and privileged and unprivileged software execution on page 3-65.

Related references

13.49 LDM on page 13-407.
7.11 Condition code suffixes on page 7-150.

13.137 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

13.138 SSAT16 on page 13-521.
13.68 MRS (PSR to general-purpose register) on page 13-437.
7.11 Condition code suffixes on page 7-150.

13.138 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

13.68 MRS (PSR to general-purpose register) on page 13-437.
7.11 Condition code suffixes on page 7-150.

13.139 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

13.107 SEL on page 13-487.
7.11 Condition code suffixes on page 7-150.

13.140 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

13.107 SEL on page 13-487.
7.11 Condition code suffixes on page 7-150.

13.141 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

13.107 SEL on page 13-487.
7.11 Condition code suffixes on page 7-150.

13.142 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 concepts

12.5 Register-relative and PC-relative expressions on page 12-302.

8.15 Address alignment in A32/T32 code on page 8-178.

Related references

7.11 Condition code suffixes on page 7-150.

13.143 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

13.47 LDAEX on page 13-403.
13.46 LDA on page 13-402.
13.144 STLEX on page 13-528.
7.11 Condition code suffixes on page 7-150.

13.144 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 Rn, 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 concepts

8.15 Address alignment in A32/T32 code on page 8-178.

Related references

13.47 LDAEX on page 13-403.
13.143 STL on page 13-527.
13.46 LDA on page 13-402.
7.11 Condition code suffixes on page 7-150.

13.145 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.

You can use the `--diag_warning 1645` assembler command-line option to check when an instruction substitution occurs.

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 *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

```
STMDB   r1!,{r3-r6,r11,r12}
```

Incorrect example

```
STM     r5!,{r5,r4,r9} ; value stored for R5 unknown
```

Related concepts

6.16 Stack implementation using LDM and STM on page 6-122.
8.15 Address alignment in A32/T32 code on page 8-178.

Related references

13.80 POP on page 13-455.
7.11 Condition code suffixes on page 7-150.

13.146 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 13-15 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
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

[z] Must be divisible by 4.
[aa] Rt and Rn must be in the range R0-R7.

Table 13-15 Offsets and architectures, STR, word, halfword, and byte (continued)

Instruction	Immediate offset	Pre-indexed	Post-indexed
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

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 instructions you can use PC for Rt in STR word instructions and PC for Rn in STR instructions with immediate 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 these A32 instructions.

In T32 code, using PC in STR 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 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 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 concepts

8.15 Address alignment in A32/T32 code on page 8-178.

Related references

7.11 Condition code suffixes on page 7-150.

[ab] Rt must be in the range R0-R7.
[ac] Must be divisible by 2.

13.147 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 13-16 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 13-16 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 concepts

8.15 Address alignment in A32/T32 code on page 8-178.

Related references

7.11 Condition code suffixes on page 7-150.

13.148 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 13-17 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`

[af] You can use –Rm, +Rm, or Rm.

Table 13-17 Offsets and architectures, STR (User mode) (continued)

Instruction	Immediate offset	Post-indexed	±*Rm* [af]	shift
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 concepts

8.15 Address alignment in A32/T32 code on page 8-178.

Related references

7.11 Condition code suffixes on page 7-150.

13.149 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 concepts

8.15 Address alignment in A32/T32 code on page 8-178.

Related references

7.11 Condition code suffixes on page 7-150.

13.150 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

13.3 Flexible second operand (Operand2) on page 13-338.
13.151 SUBS pc, lr on page 13-542.
7.11 Condition code suffixes on page 7-150.

13.151 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

13.13 AND on page 13-355.
13.63 MOV on page 13-431.
13.3 Flexible second operand (Operand2) on page 13-338.
13.9 ADD on page 13-347.
7.11 Condition code suffixes on page 7-150.

13.152 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

7.11 Condition code suffixes on page 7-150.

13.153 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

13.56 LDREX on page 13-421.
7.11 Condition code suffixes on page 7-150.

13.154 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

7.11 Condition code suffixes on page 7-150.

13.155 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

7.11 Condition code suffixes on page 7-150.

13.156 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

7.11 Condition code suffixes on page 7-150.

13.157 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

7.11 Condition code suffixes on page 7-150.

13.158 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

7.11 Condition code suffixes on page 7-150.

13.159 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

7.11 Condition code suffixes on page 7-150.

13.160 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

7.11 Condition code suffixes on page 7-150.

Related information

Arm Architecture Reference Manual.

13.161 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.

Related concepts

8.15 Address alignment in A32/T32 code on page 8-178.

13.162 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

13.3 Flexible second operand (Operand2) on page 13-338.
7.11 Condition code suffixes on page 7-150.

13.163 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

13.3 Flexible second operand (Operand2) on page 13-338.
7.11 Condition code suffixes on page 7-150.

13.164 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 instruction condition code. See *Chapter 7 Condition Codes* on page 7-139. It specifies the condition under which the instruction is executed. If `cond` is omitted, it defaults to *always* (`AL`).

q

Is an optional instruction width specifier. See *13.2 Instruction width specifiers* on page 13-337.

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

7.11 Condition code suffixes on page 7-150.
13.2 Instruction width specifiers on page 13-337.

13.165 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

13.107 SEL on page 13-487.
7.11 Condition code suffixes on page 7-150.

13.166 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

13.107 SEL on page 13-487.
7.11 Condition code suffixes on page 7-150.

13.167 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

13.107 SEL on page 13-487.
7.11 Condition code suffixes on page 7-150.

13.168 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

7.11 Condition code suffixes on page 7-150.

13.169 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, is a 16-bit unsigned immediate, in the range 0 to 65535.

T1 general registers
For T32, is an 8-bit unsigned immediate, in the range 0 to 255.

T2 general registers
For T32, is a 16-bit unsigned immediate, in the range 0 to 65535.

———— **Note** ————

The PE ignores the value of this constant.

————————————

c

Is an optional instruction condition code. See *Chapter 7 Condition Codes* on page 7-139. Arm deprecates using any *c* value other than AL.

q

Is an optional instruction width specifier. See *13.2 Instruction width specifiers* on page 13-337.

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

13.1 A32 and T32 instruction summary on page 13-332.

13.170 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

7.11 Condition code suffixes on page 7-150.

13.171 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

7.11 Condition code suffixes on page 7-150.

13.172 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

7.11 Condition code suffixes on page 7-150.

13.173 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

7.11 Condition code suffixes on page 7-150.

13.174 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

7.11 Condition code suffixes on page 7-150.

13.175 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

7.11 Condition code suffixes on page 7-150.

13.176 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

7.11 Condition code suffixes on page 7-150.

13.177 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

7.11 Condition code suffixes on page 7-150.

13.178 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

7.11 Condition code suffixes on page 7-150.

13.179 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

7.11 Condition code suffixes on page 7-150.

13.180 UND pseudo-instruction

Generate an architecturally undefined instruction.

Syntax

`UND{cond}{.W} {#expr}`

where:

cond
: is an optional condition code.

.W
: is an optional instruction width specifier.

expr
: evaluates to a numeric value. The following table shows the range and encoding of *expr* in the instruction, where Y shows the locations of the bits that encode for *expr* and V is the 4 bits that encode for the condition code.

  If *expr* is omitted, the value 0 is used.

Table 13-18 Range and encoding of expr

Instruction	Encoding	Number of bits for *expr*	Range
A32	0xV7FYYYFY	16	0-65535
T32 32-bit encoding	0xF7FYAYFY	12	0-4095
T32 16-bit encoding	0xDEYY	8	0-255

Usage

An attempt to execute an undefined instruction causes the Undefined instruction exception. Architecturally undefined instructions are expected to remain undefined.

UND in T32 code

You can use the `.W` width specifier to force `UND` to generate a 32-bit instruction in T32 code. `UND.W` always generates a 32-bit instruction, even if *expr* is in the range 0-255.

Disassembly

The encodings that this pseudo-instruction produces disassemble to `DCI`.

Related references

7.11 Condition code suffixes on page 7-150.

13.181 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 $0 \le x \le 2^8 - 1$. 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

7.11 Condition code suffixes on page 7-150.

13.182 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 $0 \leq x \leq 2^{16}$ -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

7.11 Condition code suffixes on page 7-150.

13.183 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

7.11 Condition code suffixes on page 7-150.

13.184 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

7.11 Condition code suffixes on page 7-150.

13.185 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

7.11 Condition code suffixes on page 7-150.

13.186 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 $0 \leq x \leq 2^{16}$ -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

7.11 Condition code suffixes on page 7-150.

13.187 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

7.11 Condition code suffixes on page 7-150.

13.188 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

7.11 Condition code suffixes on page 7-150.

13.189 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

13.138 SSAT16 on page 13-521.
13.68 MRS (PSR to general-purpose register) on page 13-437.
7.11 Condition code suffixes on page 7-150.

13.190 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

13.68 MRS (PSR to general-purpose register) on page 13-437.
7.11 Condition code suffixes on page 7-150.

13.191 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

13.107 SEL on page 13-487.
7.11 Condition code suffixes on page 7-150.

13.192 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 instruction is available in A32 and T32.

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

Related references

13.107 SEL on page 13-487.
7.11 Condition code suffixes on page 7-150.

13.193 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 instruction is available in A32 and T32.

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

Related references

13.107 SEL on page 13-487.
7.11 Condition code suffixes on page 7-150.

13.194 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

7.11 Condition code suffixes on page 7-150.

13.195 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

7.11 Condition code suffixes on page 7-150.

13.196 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

7.11 Condition code suffixes on page 7-150.

13.197 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

7.11 Condition code suffixes on page 7-150.

13.198 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

7.11 Condition code suffixes on page 7-150.

13.199 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

7.11 Condition code suffixes on page 7-150.

13.200 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

13.75 NOP on page 13-447.
7.11 Condition code suffixes on page 7-150.
13.110 SEV on page 13-491.
13.111 SEVL on page 13-492.
13.201 WFI on page 13-597.

13.201 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

13.75 NOP on page 13-447.
7.11 Condition code suffixes on page 7-150.
13.200 WFE on page 13-596.

13.202 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

13.75 NOP on page 13-447.
7.11 Condition code suffixes on page 7-150.

Chapter 14
Advanced SIMD Instructions (32-bit)

Describes Advanced SIMD assembly language instructions.

It contains the following sections:

14.1 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 14-1 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
VHADD, VHSUB	Halving Add, Halving Subtract
VLD	Vector Load
VMAX, VMIN	Maximum, Minimum

Table 14-1 Summary of Advanced SIMD instructions (continued)

Mnemonic	Brief description
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
VRECPS	Reciprocal Step
VREV	Reverse elements
VRHADD	Halving Add, Round

Table 14-1 Summary of Advanced SIMD instructions (continued)

Mnemonic	Brief description
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
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
VUZP, VZIP	Vector interleave and de-interleave

14.2 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 14-2 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

14.51 VLDM on page 14-658.
14.52 VLDR on page 14-659.
14.53 VLDR (post-increment and pre-decrement) on page 14-660.
14.54 VLDR pseudo-instruction on page 14-661.
14.67 VMOV (between two general-purpose registers and a 64-bit extension register) on page 14-674.
14.68 VMOV (between a general-purpose register and an Advanced SIMD scalar) on page 14-675.
14.72 VMRS on page 14-679.
14.73 VMSR on page 14-680.
14.89 VPOP on page 14-696.
14.90 VPUSH on page 14-697.
14.126 VSTM on page 14-733.
14.129 VSTR on page 14-738.
14.130 VSTR (post-increment and pre-decrement) on page 14-739.

14.3 Cryptographic instructions

A set of cryptographic instructions is available in some implementations of 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.

14.4 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.

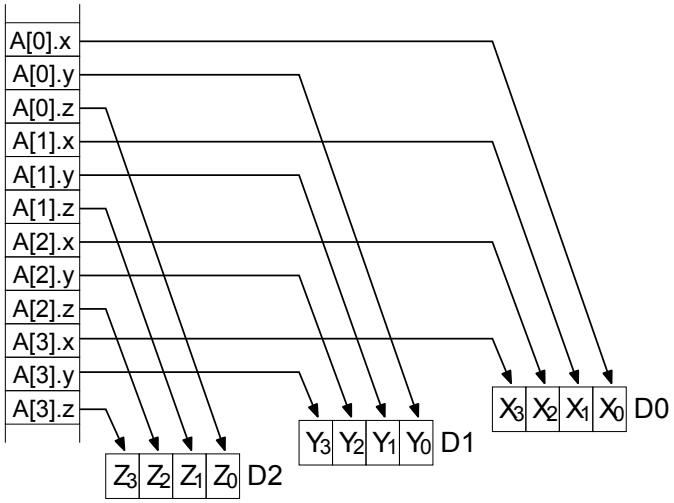


Figure 14-1 De-interleaving an array of 3-element structures

Related concepts

14.5 Alignment restrictions in load and store element and structure instructions on page 14-609.

Related references

14.48 VLDn (single n-element structure to one lane) on page 14-652.
14.49 VLDn (single n-element structure to all lanes) on page 14-654.
14.50 VLDn (multiple n-element structures) on page 14-656.
14.127 VSTn (multiple n-element structures) on page 14-734.
14.128 VSTn (single n-element structure to one lane) on page 14-736.

Related information

Arm Architecture Reference Manual.

14.5 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

14.4 Interleaving provided by load and store element and structure instructions on page 14-608.

Related references

14.48 VLDn (single n-element structure to one lane) on page 14-652.

14.49 VLDn (single n-element structure to all lanes) on page 14-654.

14.50 VLDn (multiple n-element structures) on page 14-656.

14.127 VSTn (multiple n-element structures) on page 14-734.

14.128 VSTn (single n-element structure to one lane) on page 14-736.

Related information

Arm Architecture Reference Manual.

14.6 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 instruction condition code. See *Chapter 7 Condition Codes* on page 7-139.

q
Is an optional instruction width specifier. See *13.2 Instruction width specifiers* on page 13-337.

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

14.1 Summary of Advanced SIMD instructions on page 14-603.

14.7 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 instruction condition code. See *Chapter 7 Condition Codes* on page 7-139.

q
Is an optional instruction width specifier. See *13.2 Instruction width specifiers* on page 13-337.

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

14.1 Summary of Advanced SIMD instructions on page 14-603.

14.8 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 concepts

9.10 Advanced SIMD data types in A32/T32 instructions on page 9-194.

Related references

7.11 Condition code suffixes on page 7-150.

14.9 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 concepts

9.10 Advanced SIMD data types in A32/T32 instructions on page 9-194.

Related references

7.11 Condition code suffixes on page 7-150.

14.10 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 concepts

9.10 Advanced SIMD data types in A32/T32 instructions on page 9-194.

Related references

14.91 VQABS on page 14-698.

7.11 Condition code suffixes on page 7-150.

14.11 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

7.11 Condition code suffixes on page 7-150.

14.12 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 concepts

9.10 Advanced SIMD data types in A32/T32 instructions on page 9-194.

Related references

14.14 VADDL and VADDW on page 14-618.
14.92 VQADD on page 14-699.
7.11 Condition code suffixes on page 7-150.

14.13 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 concepts

9.10 Advanced SIMD data types in A32/T32 instructions on page 9-194.

Related references

14.105 VRADDHN on page 14-712.

7.11 Condition code suffixes on page 7-150.

14.14 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 concepts

9.10 Advanced SIMD data types in A32/T32 instructions on page 9-194.

Related references

14.12 VADD on page 14-616.
7.11 Condition code suffixes on page 7-150.

14.15 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 concepts

9.10 Advanced SIMD data types in A32/T32 instructions on page 9-194.

Related references

14.17 VBIC (immediate) on page 14-621.
7.11 Condition code suffixes on page 7-150.

14.16 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

7.11 Condition code suffixes on page 7-150.

14.17 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 14-3 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 concepts

9.10 Advanced SIMD data types in A32/T32 instructions on page 9-194.

Related references

14.15 VAND (immediate) on page 14-619.

7.11 Condition code suffixes on page 7-150.

14.18 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

7.11 Condition code suffixes on page 7-150.

14.19 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

7.11 Condition code suffixes on page 7-150.

14.20 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

7.11 Condition code suffixes on page 7-150.

14.21 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

7.11 Condition code suffixes on page 7-150.

14.22 VCADD

Vector Complex Add.

Syntax

```
VCADD{q}.dt {Dd,} Dn, Dm, #rotate ; 64-bit SIMD vector FP/SIMD registers
VCADD{q}.dt {Qd,} Qn, Qm, #rotate ; 128-bit SIMD vector FP/SIMD registers
```

Where:

q
Is an optional instruction width specifier. See *13.2 Instruction width specifiers* on page 13-337.

dt
Is the data type for the elements of the vectors, and can be either `F16` or `F32`.

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.

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

14.1 Summary of Advanced SIMD instructions on page 14-603.

14.23 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 concepts

9.10 Advanced SIMD data types in A32/T32 instructions on page 9-194.

Related references

7.11 Condition code suffixes on page 7-150.

14.24 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 concepts

9.10 Advanced SIMD data types in A32/T32 instructions on page 9-194.

Related references

7.11 Condition code suffixes on page 7-150.

14.25 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 concepts

9.10 Advanced SIMD data types in A32/T32 instructions on page 9-194.

Related references

7.11 Condition code suffixes on page 7-150.

14.26 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 concepts

9.10 Advanced SIMD data types in A32/T32 instructions on page 9-194.

Related references

7.11 Condition code suffixes on page 7-150.

14.27 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 concepts

9.10 Advanced SIMD data types in A32/T32 instructions on page 9-194.

Related references

7.11 Condition code suffixes on page 7-150.

14.28 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 concepts

9.10 Advanced SIMD data types in A32/T32 instructions on page 9-194.

Related references

7.11 Condition code suffixes on page 7-150.

14.29 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 concepts

9.10 Advanced SIMD data types in A32/T32 instructions on page 9-194.

Related references

7.11 Condition code suffixes on page 7-150.

14.30 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 concepts

9.10 Advanced SIMD data types in A32/T32 instructions on page 9-194.

Related references

7.11 Condition code suffixes on page 7-150.

14.31 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 concepts

9.10 Advanced SIMD data types in A32/T32 instructions on page 9-194.

Related references

7.11 Condition code suffixes on page 7-150.

14.32 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 concepts

9.10 Advanced SIMD data types in A32/T32 instructions on page 9-194.

Related references

7.11 Condition code suffixes on page 7-150.

14.33 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 concepts

9.10 Advanced SIMD data types in A32/T32 instructions on page 9-194.

Related references

7.11 Condition code suffixes on page 7-150.

14.34 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 concepts

9.10 Advanced SIMD data types in A32/T32 instructions on page 9-194.

Related references

7.11 Condition code suffixes on page 7-150.

14.35 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:

q
Is an optional instruction width specifier. See *13.2 Instruction width specifiers* on page 13-337.

dt
Is the data type for the elements of the vectors, and can be either `F16` or `F32`.

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.

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

14.1 Summary of Advanced SIMD instructions on page 14-603.

14.36 VCMLA (by element)

Vector Complex Multiply Accumulate (by element).

Syntax

```
VCMLA{q}.F16 Dd, Dn, Dm[index], #rotate ; Double,halfprec FP/SIMD registers

VCMLA{q}.F32 Dd, Dn, Dm[0], #rotate ; Double,singleprec FP/SIMD registers

VCMLA{q}.F32 Qd, Qn, Dm[0], #rotate ; Quad,singleprec FP/SIMD registers

VCMLA{q}.F16 Qd, Qn, Dm[index], #rotate ; Halfprec,quad FP/SIMD registers
```

Where:

q
Is an optional instruction width specifier. See *13.2 Instruction width specifiers* on page 13-337.

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.

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

14.1 Summary of Advanced SIMD instructions on page 14-603.

14.37 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 concepts

9.10 Advanced SIMD data types in A32/T32 instructions on page 9-194.

Related references

7.11 Condition code suffixes on page 7-150.

14.38 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

7.11 Condition code suffixes on page 7-150.

14.39 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

7.11 Condition code suffixes on page 7-150.

14.40 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

```
VCVTmode.type Qd, Qm
```

```
VCVTmode.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.

14.41 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.

14.42 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 an general-purpose register.

Related references

7.11 Condition code suffixes on page 7-150.

14.43 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

7.11 Condition code suffixes on page 7-150.

14.44 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:

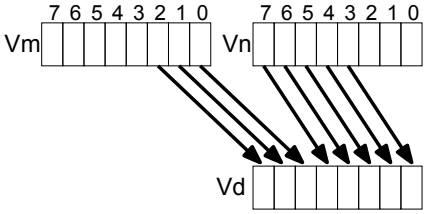


Figure 14-2 Operation of doubleword VEXT for imm = 3

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

7.11 Condition code suffixes on page 7-150.

14.45 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

14.74 VMUL on page 14-681.
7.11 Condition code suffixes on page 7-150.

14.46 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 concepts

9.10 Advanced SIMD data types in A32/T32 instructions on page 9-194.

Related references

7.11 Condition code suffixes on page 7-150.

14.47 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 concepts

9.10 Advanced SIMD data types in A32/T32 instructions on page 9-194.

Related references

7.11 Condition code suffixes on page 7-150.

14.48 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 14-4 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
	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

[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 14-4 Permitted combinations of parameters for VLDn (single n-element structure to one lane) (continued)

n	*datatype*	list [ag]	*align* [ah]	alignment
	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

14.4 Interleaving provided by load and store element and structure instructions on page 14-608.

Related references

7.11 Condition code suffixes on page 7-150.

14.49 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 14-5 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
	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

[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 14-5 Permitted combinations of parameters for VLDn (single n-element structure to all lanes) (continued)

n	*datatype*	list [ai]	*align* [aj]	alignment
		{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

14.4 Interleaving provided by load and store element and structure instructions on page 14-608.

Related references

7.11 Condition code suffixes on page 7-150.

14.50 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 14-6 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
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

[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 14-6 Permitted combinations of parameters for VLDn (multiple n-element structures) (continued)

n	*datatype*	list [ak]	*align* [al]	*alignment*
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

14.4 Interleaving provided by load and store element and structure instructions on page 14-608.

Related references

7.11 Condition code suffixes on page 7-150.

14.51 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 concepts

6.16 Stack implementation using LDM and STM on page 6-122.

Related references

7.11 Condition code suffixes on page 7-150.
15.14 VLDM (floating-point) on page 15-765.

14.52 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 concepts

12.5 Register-relative and PC-relative expressions on page 12-302.

Related references

14.54 VLDR pseudo-instruction on page 14-661.
7.11 Condition code suffixes on page 7-150.
15.15 VLDR (floating-point) on page 15-766.

14.53 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

14.51 VLDM on page 14-658.
14.52 VLDR on page 14-659.
7.11 Condition code suffixes on page 7-150.
15.16 VLDR (post-increment and pre-decrement, floating-point) on page 15-767.

14.54 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 concepts

9.10 Advanced SIMD data types in A32/T32 instructions on page 9-194.

Related references

14.52 VLDR on page 14-659.
7.11 Condition code suffixes on page 7-150.
14.54 VLDR pseudo-instruction on page 14-661.

14.55 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 concepts

9.10 Advanced SIMD data types in A32/T32 instructions on page 9-194.

Related references

14.86 VPADD on page 14-693.

7.11 Condition code suffixes on page 7-150.

14.56 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.

14.57 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 concepts

9.11 Polynomial arithmetic over {0,1} on page 9-195.

9.10 Advanced SIMD data types in A32/T32 instructions on page 9-194.

Related references

7.11 Condition code suffixes on page 7-150.

14.58 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 concepts

9.10 Advanced SIMD data types in A32/T32 instructions on page 9-194.

Related references

7.11 Condition code suffixes on page 7-150.

14.59 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 concepts

9.10 Advanced SIMD data types in A32/T32 instructions on page 9-194.

Related references

7.11 Condition code suffixes on page 7-150.

14.60 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

9.11 Polynomial arithmetic over {0,1} on page 9-195.
9.10 Advanced SIMD data types in A32/T32 instructions on page 9-194.

Related references

7.11 Condition code suffixes on page 7-150.

14.61 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 concepts

9.10 Advanced SIMD data types in A32/T32 instructions on page 9-194.

Related references

7.11 Condition code suffixes on page 7-150.

14.62 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 concepts

9.11 Polynomial arithmetic over {0,1} on page 9-195.
9.10 Advanced SIMD data types in A32/T32 instructions on page 9-194.

Related references

7.11 Condition code suffixes on page 7-150.

14.63 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 concepts

9.11 Polynomial arithmetic over {0,1} on page 9-195.

9.10 Advanced SIMD data types in A32/T32 instructions on page 9-194.

Related references

7.11 Condition code suffixes on page 7-150.

14.64 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 concepts

9.10 Advanced SIMD data types in A32/T32 instructions on page 9-194.

Related references

7.11 Condition code suffixes on page 7-150.

14.65 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 14-7 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 concepts

9.10 Advanced SIMD data types in A32/T32 instructions on page 9-194.

Related references

7.11 Condition code suffixes on page 7-150.

[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 $\pm n * 2^{-r}$, where n and r are integers, $16 <= n <= 31$, $0 <= r <= 7$.

14.66 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

7.11 Condition code suffixes on page 7-150.

14.67 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

7.11 Condition code suffixes on page 7-150.

14.68 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 concepts

9.15 Advanced SIMD scalars on page 9-199.

9.10 Advanced SIMD data types in A32/T32 instructions on page 9-194.

Related references

7.11 Condition code suffixes on page 7-150.

14.69 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 concepts

9.10 Advanced SIMD data types in A32/T32 instructions on page 9-194.

Related references

7.11 Condition code suffixes on page 7-150.

14.70 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 concepts

9.10 Advanced SIMD data types in A32/T32 instructions on page 9-194.

Related references

7.11 Condition code suffixes on page 7-150.

14.71 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 concepts

9.10 Advanced SIMD data types in A32/T32 instructions on page 9-194.

Related references

14.65 VMOV (immediate) on page 14-672.
14.17 VBIC (immediate) on page 14-621.
7.11 Condition code suffixes on page 7-150.

14.72 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

9.17 Advanced SIMD system registers in AArch32 state on page 9-201.
7.11 Condition code suffixes on page 7-150.
15.27 VMRS (floating-point) on page 15-778.

14.73 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

9.17 Advanced SIMD system registers in AArch32 state on page 9-201.
7.11 Condition code suffixes on page 7-150.
15.28 VMSR (floating-point) on page 15-779.

14.74 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 concepts

9.11 Polynomial arithmetic over {0,1} on page 9-195.

9.10 Advanced SIMD data types in A32/T32 instructions on page 9-194.

Related references

7.11 Condition code suffixes on page 7-150.

14.75 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 concepts

9.10 Advanced SIMD data types in A32/T32 instructions on page 9-194.

Related references

7.11 Condition code suffixes on page 7-150.

14.76 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 concepts

9.11 Polynomial arithmetic over {0,1} on page 9-195.
9.10 Advanced SIMD data types in A32/T32 instructions on page 9-194.

Related references

7.11 Condition code suffixes on page 7-150.

14.77 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 concepts

9.10 Advanced SIMD data types in A32/T32 instructions on page 9-194.

Related references

7.11 Condition code suffixes on page 7-150.

14.78 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

7.11 Condition code suffixes on page 7-150.

14.79 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 14-8 Available immediate values in VMVN (immediate)

datatype	imm
I8	-
I16	0xFFXY, 0xXYFF
I32	0xFFFFFFXY, 0xFFFFXYFF, 0xFFXYFFFF, 0xXYFFFFFF
	0xFFFFXY00, 0xFFXY0000
I64	-
F32	-

Related concepts

9.10 Advanced SIMD data types in A32/T32 instructions on page 9-194.

Related references

7.11 Condition code suffixes on page 7-150.

14.80 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 concepts

9.10 Advanced SIMD data types in A32/T32 instructions on page 9-194.

Related references

15.30 VNEG (floating-point) on page 15-781.

7.11 Condition code suffixes on page 7-150.

14.81 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

7.11 Condition code suffixes on page 7-150.

14.82 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 concepts

9.10 Advanced SIMD data types in A32/T32 instructions on page 9-194.

Related references

14.84 VORR (immediate) on page 14-691.
7.11 Condition code suffixes on page 7-150.

14.83 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

14.66 VMOV (register) on page 14-673.
7.11 Condition code suffixes on page 7-150.

14.84 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 14-9 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 concepts

9.10 Advanced SIMD data types in A32/T32 instructions on page 9-194.

Related references

7.11 Condition code suffixes on page 7-150.

14.85 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.

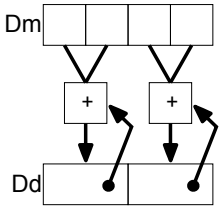


Figure 14-3 Example of operation of VPADAL (in this case for data type S16)

Related concepts

9.10 Advanced SIMD data types in A32/T32 instructions on page 9-194.

Related references

7.11 Condition code suffixes on page 7-150.

14.86 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.

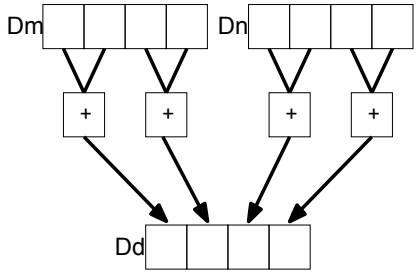


Figure 14-4 Example of operation of VPADD (in this case, for data type I16)

Related concepts

9.10 Advanced SIMD data types in A32/T32 instructions on page 9-194.

Related references

7.11 Condition code suffixes on page 7-150.

14.87 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.

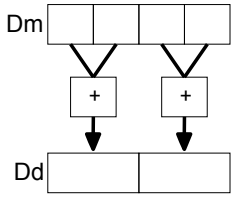


Figure 14-5 Example of operation of doubleword VPADDL (in this case, for data type S16)

Related concepts

9.10 Advanced SIMD data types in A32/T32 instructions on page 9-194.

Related references

7.11 Condition code suffixes on page 7-150.

14.88 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 concepts

9.10 Advanced SIMD data types in A32/T32 instructions on page 9-194.

Related references

7.11 Condition code suffixes on page 7-150.

14.89 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 concepts

6.16 Stack implementation using LDM and STM on page 6-122.

Related references

7.11 Condition code suffixes on page 7-150.

14.90 VPUSH on page 14-697.

15.34 VPOP (floating-point) on page 15-785.

14.90 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 concepts

6.16 Stack implementation using LDM and STM on page 6-122.

Related references

7.11 Condition code suffixes on page 7-150.
14.89 VPOP on page 14-696.
15.35 VPUSH (floating-point) on page 15-786.

14.91 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 concepts

9.10 Advanced SIMD data types in A32/T32 instructions on page 9-194.

Related references

7.11 Condition code suffixes on page 7-150.

14.92 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 concepts

9.10 Advanced SIMD data types in A32/T32 instructions on page 9-194.

Related references

7.11 Condition code suffixes on page 7-150.

14.93 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 concepts

9.10 Advanced SIMD data types in A32/T32 instructions on page 9-194.

Related references

7.11 Condition code suffixes on page 7-150.

14.94 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 concepts

9.10 Advanced SIMD data types in A32/T32 instructions on page 9-194.

Related references

7.11 Condition code suffixes on page 7-150.

14.95 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 concepts

9.10 Advanced SIMD data types in A32/T32 instructions on page 9-194.

Related references

7.11 Condition code suffixes on page 7-150.

14.96 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 concepts

9.10 Advanced SIMD data types in A32/T32 instructions on page 9-194.

Related references

7.11 Condition code suffixes on page 7-150.

14.97 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 concepts

9.10 Advanced SIMD data types in A32/T32 instructions on page 9-194.

Related references

7.11 Condition code suffixes on page 7-150.

14.98 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 concepts

9.10 Advanced SIMD data types in A32/T32 instructions on page 9-194.

Related references

7.11 Condition code suffixes on page 7-150.

14.99 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 concepts

9.10 Advanced SIMD data types in A32/T32 instructions on page 9-194.

Related references

7.11 Condition code suffixes on page 7-150.

14.100 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 14-10 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 concepts

9.10 Advanced SIMD data types in A32/T32 instructions on page 9-194.

Related references

7.11 Condition code suffixes on page 7-150.

14.101 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 concepts

9.10 Advanced SIMD data types in A32/T32 instructions on page 9-194.

Related references

7.11 Condition code suffixes on page 7-150.

14.102 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 14-11 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 concepts

9.10 Advanced SIMD data types in A32/T32 instructions on page 9-194.

Related references

7.11 Condition code suffixes on page 7-150.

14.103 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 14-12 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 concepts

9.10 Advanced SIMD data types in A32/T32 instructions on page 9-194.

Related references

7.11 Condition code suffixes on page 7-150.

14.104 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 concepts

9.10 Advanced SIMD data types in A32/T32 instructions on page 9-194.

Related references

7.11 Condition code suffixes on page 7-150.

14.105 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 concepts

9.10 Advanced SIMD data types in A32/T32 instructions on page 9-194.

Related references

7.11 Condition code suffixes on page 7-150.

14.106 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 14-13 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 concepts

9.10 Advanced SIMD data types in A32/T32 instructions on page 9-194.

Related references

7.11 Condition code suffixes on page 7-150.

[ao] The Division by Zero exception bit in the FPSCR (FPSCR[1]) is set

14.107 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 14-14 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

7.11 Condition code suffixes on page 7-150.

14.108 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

7.11 Condition code suffixes on page 7-150.

14.109 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 concepts

9.10 Advanced SIMD data types in A32/T32 instructions on page 9-194.

Related references

7.11 Condition code suffixes on page 7-150.

14.110 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 concepts

9.10 Advanced SIMD data types in A32/T32 instructions on page 9-194.

Related references

7.11 Condition code suffixes on page 7-150.

14.111 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 14-15 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 concepts

9.10 Advanced SIMD data types in A32/T32 instructions on page 9-194.

Related references

14.83 VORR (register) on page 14-690.

7.11 Condition code suffixes on page 7-150.

14.112 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 14-16 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 concepts

9.10 Advanced SIMD data types in A32/T32 instructions on page 9-194.

Related references

14.70 VMOVN on page 14-677.

7.11 Condition code suffixes on page 7-150.

14.113 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.

14.114 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 14-17 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 concepts

9.10 Advanced SIMD data types in A32/T32 instructions on page 9-194.

Related references

7.11 Condition code suffixes on page 7-150.

[ap] The Division by Zero exception bit in the FPSCR (FPSCR[1]) is set

14.115 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 14-18 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

7.11 Condition code suffixes on page 7-150.

14.116 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 14-19 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 concepts

9.10 Advanced SIMD data types in A32/T32 instructions on page 9-194.

Related references

7.11 Condition code suffixes on page 7-150.

14.117 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 concepts

9.10 Advanced SIMD data types in A32/T32 instructions on page 9-194.

Related references

7.11 Condition code suffixes on page 7-150.

14.118 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 14-20 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.

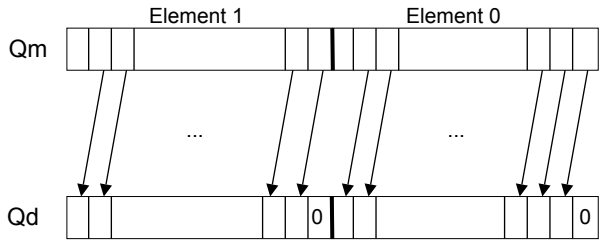


Figure 14-6 Operation of quadword VSHL.I64 Qd, Qm, #1

Related concepts

9.10 Advanced SIMD data types in A32/T32 instructions on page 9-194.

Related references

7.11 Condition code suffixes on page 7-150.

14.119 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 concepts

9.10 Advanced SIMD data types in A32/T32 instructions on page 9-194.

Related references

7.11 Condition code suffixes on page 7-150.

14.120 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 14-21 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 concepts

9.10 Advanced SIMD data types in A32/T32 instructions on page 9-194.

Related references

7.11 Condition code suffixes on page 7-150.

14.121 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 14-22 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 concepts

9.10 Advanced SIMD data types in A32/T32 instructions on page 9-194.

Related references

14.83 VORR (register) on page 14-690.
7.11 Condition code suffixes on page 7-150.

14.122 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 14-23 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 concepts

9.10 Advanced SIMD data types in A32/T32 instructions on page 9-194.

Related references

14.70 VMOVN on page 14-677.

7.11 Condition code suffixes on page 7-150.

14.123 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.

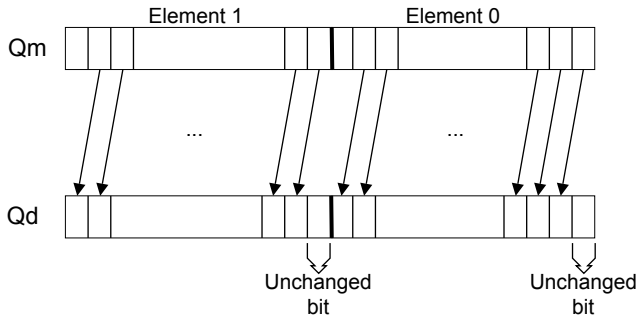


Figure 14-7 Operation of quadword VSLI.64 Qd, Qm, #1

Related references

7.11 Condition code suffixes on page 7-150.

14.124 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 14-24 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 concepts

9.10 Advanced SIMD data types in A32/T32 instructions on page 9-194.

Related references

7.11 Condition code suffixes on page 7-150.

14.125 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.

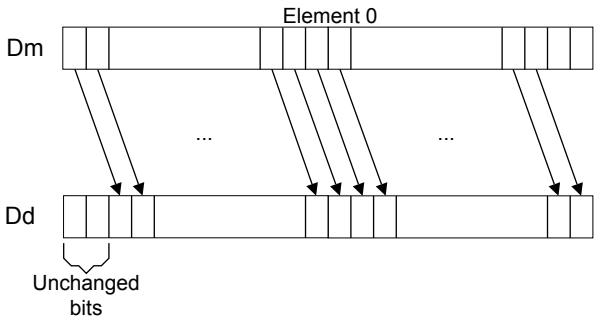


Figure 14-8 Operation of doubleword VSRI.64 Dd, Dm, #2

Related references

7.11 Condition code suffixes on page 7-150.

14.126 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 concepts

6.16 Stack implementation using LDM and STM on page 6-122.

Related references

7.11 Condition code suffixes on page 7-150.
15.39 VSTM (floating-point) on page 15-790.

14.127 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 14-25 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
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

[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 14-25 Permitted combinations of parameters for VSTn (multiple n-element structures) (continued)

n	*datatype*	list [aq]	*align* [ar]	*alignment*
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

14.4 Interleaving provided by load and store element and structure instructions on page 14-608.

14.5 Alignment restrictions in load and store element and structure instructions on page 14-609.

Related references

7.11 Condition code suffixes on page 7-150.

14.128 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

`VSTn` stores one *n*-element structure into memory from one or more Advanced SIMD registers.

Table 14-26 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
		{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

[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 14-26 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], 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

14.4 Interleaving provided by load and store element and structure instructions on page 14-608.
14.5 Alignment restrictions in load and store element and structure instructions on page 14-609.

Related references

7.11 Condition code suffixes on page 7-150.

14.129 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 concepts

12.5 Register-relative and PC-relative expressions on page 12-302.

Related references

7.11 Condition code suffixes on page 7-150.
15.40 VSTR (floating-point) on page 15-791.

14.130 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

14.129 VSTR on page 14-738.
14.126 VSTM on page 14-733.
7.11 Condition code suffixes on page 7-150.
15.41 VSTR (post-increment and pre-decrement, floating-point) on page 15-792.

14.131 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 concepts

9.10 Advanced SIMD data types in A32/T32 instructions on page 9-194.

Related references

7.11 Condition code suffixes on page 7-150.

14.132 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 concepts

9.10 Advanced SIMD data types in A32/T32 instructions on page 9-194.

Related references

7.11 Condition code suffixes on page 7-150.

14.133 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 concepts

9.10 Advanced SIMD data types in A32/T32 instructions on page 9-194.

Related references

7.11 Condition code suffixes on page 7-150.

14.134 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

7.11 Condition code suffixes on page 7-150.

14.135 VTBL and VTBX

Vector Table Lookup, Vector Table Extension.

Syntax

`Vop{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

7.11 Condition code suffixes on page 7-150.

14.136 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`:

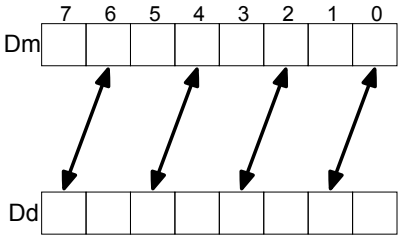


Figure 14-9 Operation of doubleword VTRN.8

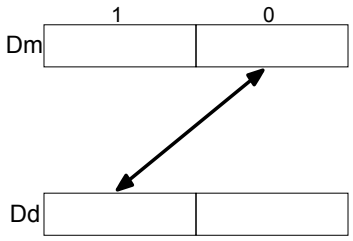


Figure 14-10 Operation of doubleword VTRN.32

Related references

7.11 Condition code suffixes on page 7-150.

14.137 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

7.11 Condition code suffixes on page 7-150.

14.138 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 14-27 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 14-28 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

14.4 Interleaving provided by load and store element and structure instructions on page 14-608.

Related references

14.136 VTRN on page 14-745.
7.11 Condition code suffixes on page 7-150.

14.139 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 14-29 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 14-30 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

14.4 Interleaving provided by load and store element and structure instructions on page 14-608.

Related references

14.136 VTRN on page 14-745.
7.11 Condition code suffixes on page 7-150.

Chapter 15
Floating-point Instructions (32-bit)

Describes floating-point assembly language instructions.

It contains the following sections:

15.1 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 15-1 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 15-1 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

15.2 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

7.11 Condition code suffixes on page 7-150.

15.3 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

7.11 Condition code suffixes on page 7-150.

15.4 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

7.11 Condition code suffixes on page 7-150.

15.5 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

7.11 Condition code suffixes on page 7-150.

15.6 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

7.11 Condition code suffixes on page 7-150.

15.7 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.

15.8 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

7.11 Condition code suffixes on page 7-150.

15.9 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

7.11 Condition code suffixes on page 7-150.

15.10 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.

15.11 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

7.11 Condition code suffixes on page 7-150.

15.12 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

15.29 VMUL (floating-point) on page 15-780.
7.11 Condition code suffixes on page 7-150.

15.13 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 *13.2 Instruction width specifiers* on page 13-337.

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

15.1 Summary of floating-point instructions on page 15-751.

15.14 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 concepts

6.16 Stack implementation using LDM and STM on page 6-122.

Related references

7.11 Condition code suffixes on page 7-150.

15.15 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 concepts

12.5 Register-relative and PC-relative expressions on page 12-302.

Related references

15.17 VLDR pseudo-instruction (floating-point) on page 15-768.
7.11 Condition code suffixes on page 7-150.

15.16 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

15.14 VLDM (floating-point) on page 15-765.
15.15 VLDR (floating-point) on page 15-766.
7.11 Condition code suffixes on page 7-150.

15.17 VLDR pseudo-instruction (floating-point)

The VLDR pseudo-instruction loads a constant value into a floating-point single-precision or double-precision register.

Note

This description is for the VLDR pseudo-instruction only.

Syntax

`VLDR{cond}.F64 Dd,=constant`

`VLDR{cond}.F32 Sd,=constant`

where:

cond
is an optional condition code.

Dd or *Sd*
is the extension register to be loaded.

constant
is an immediate value of the appropriate type for the extension register width.

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 concepts

10.10 Floating-point data types in A32/T32 instructions on page 10-217.

Related references

15.15 VLDR (floating-point) on page 15-766.
7.11 Condition code suffixes on page 7-150.

15.18 VLLDM

Floating-point Lazy Load Multiple.

Syntax

`VLLDM{c}{q} Rn`

Where:

c
Is an optional condition code. See *Chapter 7 Condition Codes* on page 7-139.

q
Is an optional instruction width specifier. See *13.2 Instruction width specifiers* on page 13-337.

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

15.1 Summary of floating-point instructions on page 15-751.

15.19 VLSTM

Floating-point Lazy Store Multiple.

Syntax

`VLSTM{c}{q} Rn`

Where:

c
Is an optional condition code. See *Chapter 7 Condition Codes* on page 7-139.

q
Is an optional instruction width specifier. See *13.2 Instruction width specifiers* on page 13-337.

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

15.1 Summary of floating-point instructions on page 15-751.

15.20 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.

15.21 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

7.11 Condition code suffixes on page 7-150.

15.22 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

7.11 Condition code suffixes on page 7-150.

15.23 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

7.11 Condition code suffixes on page 7-150.

15.24 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

7.11 Condition code suffixes on page 7-150.

15.25 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

7.11 Condition code suffixes on page 7-150.

15.26 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 concepts

10.10 Floating-point data types in A32/T32 instructions on page 10-217.

Related references

7.11 Condition code suffixes on page 7-150.

15.27 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

7.11 Condition code suffixes on page 7-150.

15.28 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

7.11 Condition code suffixes on page 7-150.

15.29 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

7.11 Condition code suffixes on page 7-150.

15.30 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

7.11 Condition code suffixes on page 7-150.

15.31 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

7.11 Condition code suffixes on page 7-150.

15.32 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

7.11 Condition code suffixes on page 7-150.

15.33 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

7.11 Condition code suffixes on page 7-150.

15.34 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 concepts

6.16 Stack implementation using LDM and STM on page 6-122.

Related references

7.11 Condition code suffixes on page 7-150.
15.35 VPUSH (floating-point) on page 15-786.

15.35 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 concepts

6.16 Stack implementation using LDM and STM on page 6-122.

Related references

7.11 Condition code suffixes on page 7-150.
15.34 VPOP (floating-point) on page 15-785.

15.36 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

7.11 Condition code suffixes on page 7-150.

15.37 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

7.13 Comparison of condition code meanings in integer and floating-point code on page 7-152.
7.11 Condition code suffixes on page 7-150.

15.38 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

7.11 Condition code suffixes on page 7-150.

15.39 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 concepts

6.16 Stack implementation using LDM and STM on page 6-122.

Related references

7.11 Condition code suffixes on page 7-150.

15.40 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 concepts

12.5 Register-relative and PC-relative expressions on page 12-302.

Related references

15.17 VLDR pseudo-instruction (floating-point) on page 15-768.
7.11 Condition code suffixes on page 7-150.

15.41 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

15.40 VSTR (floating-point) on page 15-791.
15.39 VSTM (floating-point) on page 15-790.
7.11 Condition code suffixes on page 7-150.

15.42 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

7.11 Condition code suffixes on page 7-150.

Chapter 16
A64 General Instructions

Describes the A64 general instructions.

It contains the following sections:

16.1 A64 instructions in alphabetical order

A summary of the A64 instructions and pseudo-instructions that are supported.

Table 16-1 Summary of A64 general instructions

Mnemonic	Brief description	See
ADC	Add with Carry	*16.3 ADC* on page 16-805
ADCS	Add with Carry, setting flags	*16.4 ADCS* on page 16-806
ADD (extended register)	Add (extended register)	*16.5 ADD (extended register)* on page 16-807
ADD (immediate)	Add (immediate)	*16.6 ADD (immediate)* on page 16-809
ADD (shifted register)	Add (shifted register)	*16.7 ADD (shifted register)* on page 16-810
ADDS (extended register)	Add (extended register), setting flags	*16.8 ADDS (extended register)* on page 16-811
ADDS (immediate)	Add (immediate), setting flags	*16.9 ADDS (immediate)* on page 16-813
ADDS (shifted register)	Add (shifted register), setting flags	*16.10 ADDS (shifted register)* on page 16-814
ADR	Form PC-relative address	*16.11 ADR* on page 16-815
ADRL pseudo-instruction	Load a PC-relative address into a register	*16.12 ADRL pseudo-instruction* on page 16-816
ADRP	Form PC-relative address to 4KB page	*16.13 ADRP* on page 16-817
AND (immediate)	Bitwise AND (immediate)	*16.14 AND (immediate)* on page 16-818
AND (shifted register)	Bitwise AND (shifted register)	*16.15 AND (shifted register)* on page 16-819
ANDS (immediate)	Bitwise AND (immediate), setting flags	*16.16 ANDS (immediate)* on page 16-820
ANDS (shifted register)	Bitwise AND (shifted register), setting flags	*16.17 ANDS (shifted register)* on page 16-821
ASR (register)	Arithmetic Shift Right (register)	*16.18 ASR (register)* on page 16-822
ASR (immediate)	Arithmetic Shift Right (immediate)	*16.19 ASR (immediate)* on page 16-823
ASRV	Arithmetic Shift Right Variable	*16.20 ASRV* on page 16-824
AT	Address Translate	*16.21 AT* on page 16-825
AUTDA, AUTDZA	Authenticate Data address, using key A	*16.22 AUTDA, AUTDZA* on page 16-827
AUTDB, AUTDZB	Authenticate Data address, using key B	*16.23 AUTDB, AUTDZB* on page 16-828
AUTIA, AUTIZA, AUTIA1716, AUTIASP, AUTIAZ	Authenticate Instruction address, using key A	*16.24 AUTIA, AUTIZA, AUTIA1716, AUTIASP, AUTIAZ* on page 16-829
AUTIB, AUTIZB, AUTIB1716, AUTIBSP, AUTIBZ	Authenticate Instruction address, using key B	*16.25 AUTIB, AUTIZB, AUTIB1716, AUTIBSP, AUTIBZ* on page 16-830
B.cond	Branch conditionally	*16.26 B.cond* on page 16-831
B	Branch	*16.27 B* on page 16-832
BFC	Bitfield Clear, leaving other bits unchanged	*16.28 BFC* on page 16-833
BFI	Bitfield Insert	*16.29 BFI* on page 16-834
BFM	Bitfield Move	*16.30 BFM* on page 16-835
BFXIL	Bitfield extract and insert at low end	*16.31 BFXIL* on page 16-836
BIC (shifted register)	Bitwise Bit Clear (shifted register)	*16.32 BIC (shifted register)* on page 16-837

Table 16-1 Summary of A64 general instructions (continued)

Mnemonic	Brief description	See
BICS (shifted register)	Bitwise Bit Clear (shifted register), setting flags	*16.33 BICS (shifted register)* on page 16-838
BL	Branch with Link	*16.34 BL* on page 16-839
BLR	Branch with Link to Register	*16.35 BLR* on page 16-840
BLRAA, BLRAAZ, BLRAB, BLRABZ	Branch with Link to Register, with pointer authentication	*16.36 BLRAA, BLRAAZ, BLRAB, BLRABZ* on page 16-841
BR	Branch to Register	*16.37 BR* on page 16-842
BRAA, BRAAZ, BRAB, BRABZ	Branch to Register, with pointer authentication	*16.38 BRAA, BRAAZ, BRAB, BRABZ* on page 16-843
BRK	Breakpoint instruction	*16.39 BRK* on page 16-844
CBNZ	Compare and Branch on Nonzero	*16.40 CBNZ* on page 16-845
CBZ	Compare and Branch on Zero	*16.41 CBZ* on page 16-846
CCMN (immediate)	Conditional Compare Negative (immediate)	*16.42 CCMN (immediate)* on page 16-847
CCMN (register)	Conditional Compare Negative (register)	*16.43 CCMN (register)* on page 16-848
CCMP (immediate)	Conditional Compare (immediate)	*16.44 CCMP (immediate)* on page 16-849
CCMP (register)	Conditional Compare (register)	*16.45 CCMP (register)* on page 16-850
CINC	Conditional Increment	*16.46 CINC* on page 16-851
CINV	Conditional Invert	*16.47 CINV* on page 16-852
CLREX	Clear Exclusive	*16.48 CLREX* on page 16-853
CLS	Count leading sign bits	*16.49 CLS* on page 16-854
CLZ	Count leading zero bits	*16.50 CLZ* on page 16-855
CMN (extended register)	Compare Negative (extended register)	*16.51 CMN (extended register)* on page 16-856
CMN (immediate)	Compare Negative (immediate)	*16.52 CMN (immediate)* on page 16-858
CMN (shifted register)	Compare Negative (shifted register)	*16.53 CMN (shifted register)* on page 16-859
CMP (extended register)	Compare (extended register)	*16.54 CMP (extended register)* on page 16-860
CMP (immediate)	Compare (immediate)	*16.55 CMP (immediate)* on page 16-862
CMP (shifted register)	Compare (shifted register)	*16.56 CMP (shifted register)* on page 16-863
CNEG	Conditional Negate	*16.57 CNEG* on page 16-864
CRC32B, CRC32H, CRC32W, CRC32X	CRC32 checksum performs a cyclic redundancy check (CRC) calculation on a value held in a general-purpose register	*16.58 CRC32B, CRC32H, CRC32W, CRC32X* on page 16-865
CRC32CB, CRC32CH, CRC32CW, CRC32CX	CRC32C checksum performs a cyclic redundancy check (CRC) calculation on a value held in a general-purpose register	*16.59 CRC32CB, CRC32CH, CRC32CW, CRC32CX* on page 16-866
CSEL	Conditional Select	*16.60 CSEL* on page 16-867
CSET	Conditional Set	*16.61 CSET* on page 16-868

Table 16-1 Summary of A64 general instructions (continued)

Mnemonic	Brief description	See
CSETM	Conditional Set Mask	*16.62 CSETM* on page 16-869
CSINC	Conditional Select Increment	*16.63 CSINC* on page 16-870
CSINV	Conditional Select Invert	*16.64 CSINV* on page 16-871
CSNEG	Conditional Select Negation	*16.65 CSNEG* on page 16-872
DC	Data Cache operation	*16.66 DC* on page 16-873
DCPS1	Debug Change PE State to EL1	*16.67 DCPS1* on page 16-874
DCPS2	Debug Change PE State to EL2	*16.68 DCPS2* on page 16-875
DCPS3	Debug Change PE State to EL3	*16.69 DCPS3* on page 16-876
DMB	Data Memory Barrier	*16.70 DMB* on page 16-877
DRPS	Debug restore process state	*16.71 DRPS* on page 16-879
DSB	Data Synchronization Barrier	*16.72 DSB* on page 16-880
EON (shifted register)	Bitwise Exclusive OR NOT (shifted register)	*16.73 EON (shifted register)* on page 16-882
EOR (immediate)	Bitwise Exclusive OR (immediate)	*16.74 EOR (immediate)* on page 16-883
EOR (shifted register)	Bitwise Exclusive OR (shifted register)	*16.75 EOR (shifted register)* on page 16-884
ERET	Returns from an exception	*16.76 ERET* on page 16-885
ERETAA, ERETAB	Exception Return, with pointer authentication	*16.77 ERETAA, ERETAB* on page 16-886
ESB	Error Synchronization Barrier	*16.78 ESB* on page 16-887
EXTR	Extract register	*16.79 EXTR* on page 16-888
HINT	Hint instruction	*16.80 HINT* on page 16-889
HLT	Halt instruction	*16.81 HLT* on page 16-890
HVC	Hypervisor call to allow OS code to call the Hypervisor	*16.82 HVC* on page 16-891
IC	Instruction Cache operation	*16.83 IC* on page 16-892
ISB	Instruction Synchronization Barrier	*16.84 ISB* on page 16-893
LSL (register)	Logical Shift Left (register)	*16.85 LSL (register)* on page 16-894
LSL (immediate)	Logical Shift Left (immediate)	*16.86 LSL (immediate)* on page 16-895
LSLV	Logical Shift Left Variable	*16.87 LSLV* on page 16-896
LSR (register)	Logical Shift Right (register)	*16.88 LSR (register)* on page 16-897
LSR (immediate)	Logical Shift Right (immediate)	*16.89 LSR (immediate)* on page 16-898
LSRV	Logical Shift Right Variable	*16.90 LSRV* on page 16-899
MADD	Multiply-Add	*16.91 MADD* on page 16-900
MNEG	Multiply-Negate	*16.92 MNEG* on page 16-901
MOV (to or from SP)	Move between register and stack pointer	*16.93 MOV (to or from SP)* on page 16-902

Table 16-1 Summary of A64 general instructions (continued)

Mnemonic	Brief description	See
MOV (inverted wide immediate)	Move (inverted wide immediate)	*16.94 MOV (inverted wide immediate)* on page 16-903
MOV (wide immediate)	Move (wide immediate)	*16.95 MOV (wide immediate)* on page 16-904
MOV (bitmask immediate)	Move (bitmask immediate)	*16.96 MOV (bitmask immediate)* on page 16-905
MOV (register)	Move (register)	*16.97 MOV (register)* on page 16-906
MOVK	Move wide with keep	*16.98 MOVK* on page 16-907
MOVL pseudo-instruction	Load a register with either a 32-bit or 64-bit immediate value or any address	*16.99 MOVL pseudo-instruction* on page 16-908
MOVN	Move wide with NOT	*16.100 MOVN* on page 16-909
MOVZ	Move wide with zero	*16.101 MOVZ* on page 16-910
MRS	Move System Register	*16.102 MRS* on page 16-911
MSR (immediate)	Move immediate value to Special Register	*16.103 MSR (immediate)* on page 16-912
MSR (register)	Move general-purpose register to System Register	*16.104 MSR (register)* on page 16-913
MSUB	Multiply-Subtract	*16.105 MSUB* on page 16-914
MUL	Multiply	*16.106 MUL* on page 16-915
MVN	Bitwise NOT	*16.107 MVN* on page 16-916
NEG (shifted register)	Negate (shifted register)	*16.108 NEG (shifted register)* on page 16-917
NEGS	Negate, setting flags	*16.109 NEGS* on page 16-918
NGC	Negate with Carry	*16.110 NGC* on page 16-919
NGCS	Negate with Carry, setting flags	*16.111 NGCS* on page 16-920
NOP	No Operation	*16.112 NOP* on page 16-921
ORN (shifted register)	Bitwise OR NOT (shifted register)	*16.113 ORN (shifted register)* on page 16-922
ORR (immediate)	Bitwise OR (immediate)	*16.114 ORR (immediate)* on page 16-923
ORR (shifted register)	Bitwise OR (shifted register)	*16.115 ORR (shifted register)* on page 16-924
PACDA, PACDZA	Pointer Authentication Code for Data address, using key A	*16.116 PACDA, PACDZA* on page 16-925
PACDB, PACDZB	Pointer Authentication Code for Data address, using key B	*16.117 PACDB, PACDZB* on page 16-926
PACGA	Pointer Authentication Code, using Generic key	*16.118 PACGA* on page 16-927
PACIA, PACIZA, PACIA1716, PACIASP, PACIAZ	Pointer Authentication Code for Instruction address, using key A	*16.119 PACIA, PACIZA, PACIA1716, PACIASP, PACIAZ* on page 16-928
PACIB, PACIZB, PACIB1716, PACIBSP, PACIBZ	Pointer Authentication Code for Instruction address, using key B	*16.120 PACIB, PACIZB, PACIB1716, PACIBSP, PACIBZ* on page 16-929
PSB	Profiling Synchronization Barrier	*16.121 PSB* on page 16-930
RBIT	Reverse Bits	*16.122 RBIT* on page 16-931

Table 16-1 Summary of A64 general instructions (continued)

Mnemonic	Brief description	See
RET	Return from subroutine	*16.123 RET* on page 16-932
RETAA, RETAB	Return from subroutine, with pointer authentication	*16.124 RETAA, RETAB* on page 16-933
REV16	Reverse bytes in 16-bit halfwords	*16.125 REV16* on page 16-934
REV32	Reverse bytes in 32-bit words	*16.126 REV32* on page 16-935
REV64	Reverse Bytes	*16.127 REV64* on page 16-936
REV	Reverse Bytes	*16.128 REV* on page 16-937
ROR (immediate)	Rotate right (immediate)	*16.129 ROR (immediate)* on page 16-938
ROR (register)	Rotate Right (register)	*16.130 ROR (register)* on page 16-939
RORV	Rotate Right Variable	*16.131 RORV* on page 16-940
SBC	Subtract with Carry	*16.132 SBC* on page 16-941
SBCS	Subtract with Carry, setting flags	*16.133 SBCS* on page 16-942
SBFIZ	Signed Bitfield Insert in Zero	*16.134 SBFIZ* on page 16-943
SBFM	Signed Bitfield Move	*16.135 SBFM* on page 16-944
SBFX	Signed Bitfield Extract	*16.136 SBFX* on page 16-945
SDIV	Signed Divide	*16.137 SDIV* on page 16-946
SEV	Send Event	*16.138 SEV* on page 16-947
SEVL	Send Event Local	*16.139 SEVL* on page 16-948
SMADDL	Signed Multiply-Add Long	*16.140 SMADDL* on page 16-949
SMC	Supervisor call to allow OS or Hypervisor code to call the Secure Monitor	*16.141 SMC* on page 16-950
SMNEGL	Signed Multiply-Negate Long	*16.142 SMNEGL* on page 16-951
SMSUBL	Signed Multiply-Subtract Long	*16.143 SMSUBL* on page 16-952
SMULH	Signed Multiply High	*16.144 SMULH* on page 16-953
SMULL	Signed Multiply Long	*16.145 SMULL* on page 16-954
SUB (extended register)	Subtract (extended register)	*16.146 SUB (extended register)* on page 16-955
SUB (immediate)	Subtract (immediate)	*16.147 SUB (immediate)* on page 16-957
SUB (shifted register)	Subtract (shifted register)	*16.148 SUB (shifted register)* on page 16-958
SUBS (extended register)	Subtract (extended register), setting flags	*16.149 SUBS (extended register)* on page 16-959
SUBS (immediate)	Subtract (immediate), setting flags	*16.150 SUBS (immediate)* on page 16-961
SUBS (shifted register)	Subtract (shifted register), setting flags	*16.151 SUBS (shifted register)* on page 16-962
SVC	Supervisor call to allow application code to call the OS	*16.152 SVC* on page 16-963
SXTB	Signed Extend Byte	*16.153 SXTB* on page 16-964
SXTH	Sign Extend Halfword	*16.154 SXTH* on page 16-965
SXTW	Sign Extend Word	*16.155 SXTW* on page 16-966

Table 16-1 Summary of A64 general instructions (continued)

Mnemonic	Brief description	See
SYS	System instruction	*16.156 SYS* on page 16-967
SYSL	System instruction with result	*16.157 SYSL* on page 16-968
TBNZ	Test bit and Branch if Nonzero	*16.158 TBNZ* on page 16-969
TBZ	Test bit and Branch if Zero	*16.159 TBZ* on page 16-970
TLBI	TLB Invalidate operation	*16.160 TLBI* on page 16-971
TST (immediate)	, setting the condition flags and discarding the result	*16.161 TST (immediate)* on page 16-973
TST (shifted register)	Test (shifted register)	*16.162 TST (shifted register)* on page 16-974
UBFIZ	Unsigned Bitfield Insert in Zero	*16.163 UBFIZ* on page 16-975
UBFM	Unsigned Bitfield Move	*16.164 UBFM* on page 16-976
UBFX	Unsigned Bitfield Extract	*16.165 UBFX* on page 16-977
UDIV	Unsigned Divide	*16.166 UDIV* on page 16-978
UMADDL	Unsigned Multiply-Add Long	*16.167 UMADDL* on page 16-979
UMNEGL	Unsigned Multiply-Negate Long	*16.168 UMNEGL* on page 16-980
UMSUBL	Unsigned Multiply-Subtract Long	*16.169 UMSUBL* on page 16-981
UMULH	Unsigned Multiply High	*16.170 UMULH* on page 16-982
UMULL	Unsigned Multiply Long	*16.171 UMULL* on page 16-983
UXTB	Unsigned Extend Byte	*16.172 UXTB* on page 16-984
UXTH	Unsigned Extend Halfword	*16.173 UXTH* on page 16-985
WFE	Wait For Event	*16.174 WFE* on page 16-986
WFI	Wait For Interrupt	*16.175 WFI* on page 16-987
XPACD, XPACI, XPACLRI	Strip Pointer Authentication Code	*16.176 XPACD, XPACI, XPACLRI* on page 16-988
YIELD	YIELD	*16.177 YIELD* on page 16-989

16.2 Register restrictions for A64 instructions

In A64 instructions, the general-purpose integer registers are W0-W30 for 32-bit registers and X0-X30 for 64-bit registers.

You cannot refer to register 31 by number. In a few instructions, you can refer to it using one of the following names:

`WSP`
: the current stack pointer in a 32-bit context.

`SP`
: the current stack pointer in a 64-bit context.

`WZR`
: the zero register in a 32-bit context.

`XZR`
: the zero register in a 64-bit context.

You can only use one of these names if it is mentioned in the Syntax section for the instruction.

You cannot refer to the Program Counter (PC) explicitly by name or by number.

16.3 ADC

Add with Carry.

Syntax

```
ADC Wd, Wn, Wm ; 32-bit
ADC Xd, Xn, Xm ; 64-bit
```

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

Wn
Is the 32-bit name of the first general-purpose source register.

Wm
Is the 32-bit name of the second general-purpose source register.

Xd
Is the 64-bit name of the general-purpose destination register.

Xn
Is the 64-bit name of the first general-purpose source register.

Xm
Is the 64-bit name of the second general-purpose source register.

Operation

Add with Carry adds two register values and the Carry flag value, and writes the result to the destination register.

`Rd = Rn + Rm + C`, where *R* is either `W` or `X`.

Related references

16.1 A64 instructions in alphabetical order on page 16-798.

16.4 ADCS

Add with Carry, setting flags.

Syntax

```
ADCS Wd, Wn, Wm ; 32-bit

ADCS Xd, Xn, Xm ; 64-bit
```

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

Wn
Is the 32-bit name of the first general-purpose source register.

Wm
Is the 32-bit name of the second general-purpose source register.

Xd
Is the 64-bit name of the general-purpose destination register.

Xn
Is the 64-bit name of the first general-purpose source register.

Xm
Is the 64-bit name of the second general-purpose source register.

Operation

Add with Carry, setting flags, adds two register values and the Carry flag value, and writes the result to the destination register. It updates the condition flags based on the result.

`Rd = Rn + Rm + C`, where `R` is either `W` or `X`.

Related references

16.1 A64 instructions in alphabetical order on page 16-798.

16.5 ADD (extended register)

Add (extended register).

Syntax

```
ADD Wd|WSP, Wn|WSP, Wm{, extend {#amount}} ; 32-bit
```

```
ADD Xd|SP, Xn|SP, Rm{, extend {#amount}} ; 64-bit
```

Where:

Wd|WSP
Is the 32-bit name of the destination general-purpose register or stack pointer.

Wn|WSP
Is the 32-bit name of the first source general-purpose register or stack pointer.

Wm
Is the 32-bit name of the second general-purpose source register.

extend
Is the extension to be applied to the second source operand:

32-bit general registers
Can be one of UXTB, UXTH, LSL|UXTW, UXTX, SXTB, SXTH, SXTW or SXTX.

If *Rd* or *Rn* is WSP then LSL is preferred rather than UXTW, and can be omitted when *amount* is 0. In all other cases *extend* is required and must be UXTW rather than LSL.

64-bit general registers
Can be one of UXTB, UXTH, UXTW, LSL|UXTX, SXTB, SXTH, SXTW or SXTX.

If *Rd* or *Rn* is SP then LSL is preferred rather than UXTX, and can be omitted when *amount* is 0. In all other cases *extend* is required and must be UXTX rather than LSL.

Xd|SP
Is the 64-bit name of the destination general-purpose register or stack pointer.

Xn|SP
Is the 64-bit name of the first source general-purpose register or stack pointer.

R
Is a width specifier, and can be either W or X.

m
Is the number [0-30] of the second general-purpose source register or the name ZR (31).

amount
Is the left shift amount to be applied after extension in the range 0 to 4, defaulting to 0. It must be absent when *extend* is absent, is required when *extend* is LSL, and is optional when *extend* is present but not LSL.

Operation

Add (extended register) adds a register value and a sign or zero-extended register value, followed by an optional left shift amount, and writes the result to the destination register. The argument that is extended from the *Rm* register can be a byte, halfword, word, or doubleword.

Rd = *Rn* + LSL(*extend*(*Rm*), *amount*), where *R* is either W or X.

Usage

Table 16-2 ADD (64-bit general registers) specifier combinations

R	*extend*
W	SXTB
W	SXTH
W	SXTW

Table 16-2 ADD (64-bit general registers) specifier combinations (continued)

R	*extend*
W	UXTB
W	UXTH
W	UXTW
X	LSL|UXTX
X	SXTX

Related references

16.1 A64 instructions in alphabetical order on page 16-798.

16.6 ADD (immediate)

Add (immediate).

This instruction is used by the alias `MOV` (to or from SP).

Syntax

```
ADD Wd|WSP, Wn|WSP, #imm{, shift} ; 32-bit
```

```
ADD Xd|SP, Xn|SP, #imm{, shift} ; 64-bit
```

Where:

Wd|WSP
Is the 32-bit name of the destination general-purpose register or stack pointer.

Wn|WSP
Is the 32-bit name of the source general-purpose register or stack pointer.

Xd|SP
Is the 64-bit name of the destination general-purpose register or stack pointer.

Xn|SP
Is the 64-bit name of the source general-purpose register or stack pointer.

imm
Is an unsigned immediate, in the range 0 to 4095.

shift
Is the optional left shift to apply to the immediate, defaulting to `LSL #0`, and can be either `LSL #0` or `LSL #12`.

Operation

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

`Rd = Rn + shift(imm)`, where *R* is either `W` or `X`.

Related references

16.93 MOV (to or from SP) on page 16-902.
16.1 A64 instructions in alphabetical order on page 16-798.

16.7 ADD (shifted register)

Add (shifted register).

Syntax

```
ADD Wd, Wn, Wm{, shift #amount} ; 32-bit
```

```
ADD Xd, Xn, Xm{, shift #amount} ; 64-bit
```

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

Wn
Is the 32-bit name of the first general-purpose source register.

Wm
Is the 32-bit name of the second general-purpose source register.

amount
Depends on the instruction variant:

32-bit general registers
Is the shift amount, in the range 0 to 31, defaulting to 0.

64-bit general registers
Is the shift amount, in the range 0 to 63, defaulting to 0.

Xd
Is the 64-bit name of the general-purpose destination register.

Xn
Is the 64-bit name of the first general-purpose source register.

Xm
Is the 64-bit name of the second general-purpose source register.

shift
Is the optional shift type to be applied to the second source operand, defaulting to LSL, and can be one of `LSL`, `LSR`, or `ASR`.

Operation

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

`Rd = Rn + shift(Rm, amount)`, where `R` is either `W` or `X`.

Related references

16.1 A64 instructions in alphabetical order on page 16-798.

16.8 ADDS (extended register)

Add (extended register), setting flags.

This instruction is used by the alias `CMN` (extended register).

Syntax

```
ADDS Wd, Wn|WSP, Wm{, extend {#amount}} ; 32-bit

ADDS Xd, Xn|SP, Rm{, extend {#amount}} ; 64-bit
```

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

Wn|WSP
Is the 32-bit name of the first source general-purpose register or stack pointer.

Wm
Is the 32-bit name of the second general-purpose source register.

extend
Is the extension to be applied to the second source operand:

32-bit general registers
Can be one of `UXTB`, `UXTH`, `LSL|UXTW`, `UXTX`, `SXTB`, `SXTH`, `SXTW` or `SXTX`.

If *Rn* is WSP then `LSL` is preferred rather than `UXTW`, and can be omitted when *amount* is 0. In all other cases *extend* is required and must be `UXTW` rather than `LSL`.

64-bit general registers
Can be one of `UXTB`, `UXTH`, `UXTW`, `LSL|UXTX`, `SXTB`, `SXTH`, `SXTW` or `SXTX`.

If *Rn* is SP then `LSL` is preferred rather than `UXTX`, and can be omitted when *amount* is 0. In all other cases *extend* is required and must be `UXTX` rather than `LSL`.

Xd
Is the 64-bit name of the general-purpose destination register.

Xn|SP
Is the 64-bit name of the first source general-purpose register or stack pointer.

R
Is a width specifier, and can be either `W` or `X`.

m
Is the number [0-30] of the second general-purpose source register or the name ZR (31).

amount
Is the left shift amount to be applied after extension in the range 0 to 4, defaulting to 0. It must be absent when *extend* is absent, is required when *extend* is LSL, and is optional when *extend* is present but not LSL.

Operation

Add (extended register), setting flags, adds a register value and a sign or zero-extended register value, followed by an optional left shift amount, and writes the result to the destination register. The argument that is extended from the *Rm* register can be a byte, halfword, word, or doubleword. It updates the condition flags based on the result.

`Rd = Rn + LSL(extend(Rm), amount)`, where *R* is either `W` or `X`.

Usage

Table 16-3 ADDS (64-bit general registers) specifier combinations

R	*extend*
W	SXTB
W	SXTH
W	SXTW
W	UXTB
W	UXTH
W	UXTW
X	LSL|UXTX
X	SXTX

Related references

16.51 CMN (extended register) on page 16-856.
16.1 A64 instructions in alphabetical order on page 16-798.

16.9 ADDS (immediate)

Add (immediate), setting flags.

This instruction is used by the alias `CMN` (immediate).

Syntax

`ADDS` *`Wd`*`,` *`Wn|WSP`*`, #`*`imm`*`{,` *`shift`*`}` ; 32-bit

`ADDS` *`Xd`*`,` *`Xn|SP`*`, #`*`imm`*`{,` *`shift`*`}` ; 64-bit

Where:

`Wd`
Is the 32-bit name of the general-purpose destination register.

`Wn|WSP`
Is the 32-bit name of the source general-purpose register or stack pointer.

`Xd`
Is the 64-bit name of the general-purpose destination register.

`Xn|SP`
Is the 64-bit name of the source general-purpose register or stack pointer.

`imm`
Is an unsigned immediate, in the range 0 to 4095.

`shift`
Is the optional left shift to apply to the immediate, defaulting to `LSL #0`, and can be either `LSL #0` or `LSL #12`.

Operation

Add (immediate), setting flags, adds a register value and an optionally-shifted immediate value, and writes the result to the destination register. It updates the condition flags based on the result.

`R``d = `*`R`*`n + `*`shift`*`(imm)`, where *`R`* is either `W` or `X`.

Related references

16.52 CMN (immediate) on page 16-858.
16.1 A64 instructions in alphabetical order on page 16-798.

16.10 ADDS (shifted register)

Add (shifted register), setting flags.

This instruction is used by the alias `CMN` (shifted register).

Syntax

```
ADDS Wd, Wn, Wm{, shift #amount} ; 32-bit

ADDS Xd, Xn, Xm{, shift #amount} ; 64-bit
```

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

Wn
Is the 32-bit name of the first general-purpose source register.

Wm
Is the 32-bit name of the second general-purpose source register.

amount
Depends on the instruction variant:

32-bit general registers
Is the shift amount, in the range 0 to 31, defaulting to 0.

64-bit general registers
Is the shift amount, in the range 0 to 63, defaulting to 0.

Xd
Is the 64-bit name of the general-purpose destination register.

Xn
Is the 64-bit name of the first general-purpose source register.

Xm
Is the 64-bit name of the second general-purpose source register.

shift
Is the optional shift type to be applied to the second source operand, defaulting to LSL, and can be one of `LSL`, `LSR`, or `ASR`.

Operation

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

`Rd = Rn + shift(Rm, amount)`, where `R` is either `W` or `X`.

Related references

16.53 CMN (shifted register) on page 16-859.
16.1 A64 instructions in alphabetical order on page 16-798.

16.11 ADR

Form PC-relative address.

Syntax

```
ADR Xd, label
```

Where:

Xd
Is the 64-bit name of the general-purpose destination register.

label
Is the program label whose address is to be calculated. Its offset from the address of this instruction, in the range ±1MB.

Usage

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

Related references

16.1 A64 instructions in alphabetical order on page 16-798.

16.12 ADRL pseudo-instruction

Load a PC-relative address into a register. It is similar to the ADR instruction but ADRL can load a wider range of addresses than ADR because it generates two data processing instructions.

Syntax

ADRL *Wd,label*

ADRL *Xd,label*

where:

Wd
Is the register to load with a 32-bit address.

Xd
Is the register to load with a 64-bit address.

label
Is a PC-relative expression.

Usage

ADRL assembles to two instructions, an ADRP followed by ADD.

If the assembler cannot construct the address in two instructions, it generates a relocation. The linker then generates the correct offsets.

ADRL produces position-independent code, because the address is calculated relative to PC.

Example

```
     ADRL  x0, mylabel   ; loads address of mylabel into x0
```

Related concepts

12.5 Register-relative and PC-relative expressions on page 12-302.

Related information

Armv8-A Architecture Reference Manual.

16.13 ADRP

Form PC-relative address to 4KB page.

Syntax

`ADRP Xd, label`

Where:

Xd
Is the 64-bit name of the general-purpose destination register.

label
Is the program label whose 4KB page address is to be calculated. Its offset from the page address of this instruction, in the range ±4GB.

Usage

Form PC-relative address to 4KB page adds an immediate value that is shifted left by 12 bits, to the PC value to form a PC-relative address, with the bottom 12 bits masked out, and writes the result to the destination register.

Related references

16.1 A64 instructions in alphabetical order on page 16-798.

16.14 AND (immediate)

Bitwise AND (immediate).

Syntax

```
AND Wd|WSP, Wn, #imm ; 32-bit
```

```
AND Xd|SP, Xn, #imm ; 64-bit
```

Where:

Wd|WSP
Is the 32-bit name of the destination general-purpose register or stack pointer.

Wn
Is the 32-bit name of the general-purpose source register.

imm
The bitmask immediate.

Xd|SP
Is the 64-bit name of the destination general-purpose register or stack pointer.

Xn
Is the 64-bit name of the general-purpose source register.

Operation

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

`Rd = Rn AND imm`, where `R` is either `W` or `X`.

Related references

16.1 A64 instructions in alphabetical order on page 16-798.

16.15 AND (shifted register)

Bitwise AND (shifted register).

Syntax

```
AND Wd, Wn, Wm{, shift #amount} ; 32-bit
AND Xd, Xn, Xm{, shift #amount} ; 64-bit
```

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

Wn
Is the 32-bit name of the first general-purpose source register.

Wm
Is the 32-bit name of the second general-purpose source register.

amount
Depends on the instruction variant:

32-bit general registers
Is the shift amount, in the range 0 to 31, defaulting to 0.

64-bit general registers
Is the shift amount, in the range 0 to 63, defaulting to 0.

Xd
Is the 64-bit name of the general-purpose destination register.

Xn
Is the 64-bit name of the first general-purpose source register.

Xm
Is the 64-bit name of the second general-purpose source register.

shift
Is the optional shift to be applied to the final source, defaulting to LSL, and can be one of `LSL`, `LSR`, `ASR`, or `ROR`.

Operation

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

`Rd = Rn AND shift(Rm, amount)`, where `R` is either `W` or `X`.

Related references

16.1 A64 instructions in alphabetical order on page 16-798.

16.16 ANDS (immediate)

Bitwise AND (immediate), setting flags.

This instruction is used by the alias `TST` (immediate).

Syntax

```
ANDS Wd, Wn, #imm ; 32-bit

ANDS Xd, Xn, #imm ; 64-bit
```

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

Wn
Is the 32-bit name of the general-purpose source register.

imm
The bitmask immediate.

Xd
Is the 64-bit name of the general-purpose destination register.

Xn
Is the 64-bit name of the general-purpose source register.

Operation

Bitwise AND (immediate), setting flags, performs a bitwise AND of a register value and an immediate value, and writes the result to the destination register. It updates the condition flags based on the result.

`Rd = Rn AND imm`, where *R* is either `W` or `X`.

Related references

16.161 TST (immediate) on page 16-973.
16.1 A64 instructions in alphabetical order on page 16-798.

16.17 ANDS (shifted register)

Bitwise AND (shifted register), setting flags.

This instruction is used by the alias `TST` (shifted register).

Syntax

```
ANDS Wd, Wn, Wm{, shift #amount} ; 32-bit
```

```
ANDS Xd, Xn, Xm{, shift #amount} ; 64-bit
```

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

Wn
Is the 32-bit name of the first general-purpose source register.

Wm
Is the 32-bit name of the second general-purpose source register.

amount
Depends on the instruction variant:

32-bit general registers
Is the shift amount, in the range 0 to 31, defaulting to 0.

64-bit general registers
Is the shift amount, in the range 0 to 63, defaulting to 0.

Xd
Is the 64-bit name of the general-purpose destination register.

Xn
Is the 64-bit name of the first general-purpose source register.

Xm
Is the 64-bit name of the second general-purpose source register.

shift
Is the optional shift to be applied to the final source, defaulting to LSL, and can be one of `LSL`, `LSR`, `ASR`, or `ROR`.

Operation

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

`Rd = Rn AND shift(Rm, amount)`, where `R` is either `W` or `X`.

Related references

16.162 TST (shifted register) on page 16-974.
16.1 A64 instructions in alphabetical order on page 16-798.

16.18 ASR (register)

Arithmetic Shift Right (register).

This instruction is an alias of ASRV.

The equivalent instruction is ASRV *Wd, Wn, Wm*.

Syntax

```
ASR Wd, Wn, Wm ; 32-bit

ASR Xd, Xn, Xm ; 64-bit
```

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

Wn
Is the 32-bit name of the first general-purpose source register.

Wm
Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits.

Xd
Is the 64-bit name of the general-purpose destination register.

Xn
Is the 64-bit name of the first general-purpose source register.

Xm
Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits.

Operation

Arithmetic Shift Right (register) shifts a register value right by a variable number of bits, shifting in copies of its sign bit, and writes the result to the destination register. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is right-shifted.

Rd = ASR(*Rn*, *Rm*), where *R* is either W or X.

Related references

16.20 ASRV on page 16-824.
16.1 A64 instructions in alphabetical order on page 16-798.

16.19 ASR (immediate)

Arithmetic Shift Right (immediate).

This instruction is an alias of `SBFM`.

The equivalent instruction is `SBFM Wd, Wn, #shift, #31`.

Syntax

```
ASR Wd, Wn, #shift ; 32-bit
ASR Xd, Xn, #shift ; 64-bit
```

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

Wn
Is the 32-bit name of the general-purpose source register.

shift
Depends on the instruction variant:

32-bit general registers
Is the shift amount, in the range 0 to 31.

64-bit general registers
Is the shift amount, in the range 0 to 63.

Xd
Is the 64-bit name of the general-purpose destination register.

Xn
Is the 64-bit name of the general-purpose source register.

Operation

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

`Rd = ASR(Rn, shift)`, where `R` is either `W` or `X`.

Related references

16.135 SBFM on page 16-944.
16.1 A64 instructions in alphabetical order on page 16-798.

16.20 ASRV

Arithmetic Shift Right Variable.

This instruction is used by the alias `ASR` (register).

Syntax

```
ASRV Wd, Wn, Wm ; 32-bit general registers

ASRV Xd, Xn, Xm ; 64-bit general registers
```

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

Wn
Is the 32-bit name of the first general-purpose source register.

Wm
Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits.

Xd
Is the 64-bit name of the general-purpose destination register.

Xn
Is the 64-bit name of the first general-purpose source register.

Xm
Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits.

Operation

Arithmetic Shift Right Variable shifts a register value right by a variable number of bits, shifting in copies of its sign bit, and writes the result to the destination register. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is right-shifted.

`Rd = ASR(Rn, Rm)`, where `R` is either `W` or `X`.

Related references

16.18 ASR (register) on page 16-822.
16.1 A64 instructions in alphabetical order on page 16-798.

16.21 AT

Address Translate.

This instruction is an alias of SYS.

The equivalent instruction is `SYS #op1, C7, Cm, #op2, Xt`.

Syntax

`AT at_op, Xt`

Where:

at_op
Is an AT instruction name, as listed for the AT system instruction group, and can be one of the values shown in Usage.

op1
Is a 3-bit unsigned immediate, in the range 0 to 7.

Cm
Is a name `Cm`, with `m` in the range 0 to 15.

op2
Is a 3-bit unsigned immediate, in the range 0 to 7.

Xt
Is the 64-bit name of the general-purpose source register.

Usage

Address Translate. For more information, see *A64 system instructions for address translation* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

The following table shows the valid specifier combinations:

Table 16-4 SYS parameter values corresponding to AT operations

at_op	*op1*	*Cm*	*op2*
S12E0R	4	0	6
S12E0W	4	0	7
S12E1R	4	0	4
S12E1W	4	0	5
S1E0R	0	0	2
S1E0W	0	0	3
S1E1R	0	0	0
S1E1RP	0	1	0
S1E1W	0	0	1
S1E1WP	0	1	1
S1E2R	4	0	0
S1E2W	4	0	1
S1E3R	6	0	0
S1E3W	6	0	1

Related references

16.156 SYS on page 16-967.
16.1 A64 instructions in alphabetical order on page 16-798.

16.22 AUTDA, AUTDZA

Authenticate Data address, using key A.

Syntax

```
AUTDA Xd, Xn|SP ; AUTDA general registers

AUTDZA Xd ; AUTDZA general registers
```

Where:

Xn|SP
Is the 64-bit name of the general-purpose source register or stack pointer.

Xd
Is the 64-bit name of the general-purpose destination register.

Architectures supported

Supported in the Armv8.3-A architecture and later.

Usage

Authenticate Data address, using key A. This instruction authenticates a data address, using a modifier and key A.

The address is in the general-purpose register that is specified by *Xd*.

The modifier is:

- In the general-purpose register or stack pointer that is specified by *Xn|SP* for AUTDA.
- The value zero, for AUTDZA.

If the authentication passes, the upper bits of the address are restored to enable subsequent use of the address. If the authentication fails, the upper bits are corrupted and any subsequent use of the address results in a Translation fault.

Related references

16.1 A64 instructions in alphabetical order on page 16-798.

16.23 AUTDB, AUTDZB

Authenticate Data address, using key B.

Syntax

```
AUTDB Xd, Xn|SP ; AUTDB general registers

AUTDZB Xd ; AUTDZB general registers
```

Where:

Xn|SP
Is the 64-bit name of the general-purpose source register or stack pointer.

Xd
Is the 64-bit name of the general-purpose destination register.

Architectures supported

Supported in the Armv8.3-A architecture and later.

Usage

Authenticate Data address, using key B. This instruction authenticates a data address, using a modifier and key B.

The address is in the general-purpose register that is specified by *Xd*.

The modifier is:

- In the general-purpose register or stack pointer that is specified by *Xn|SP* for AUTDB.
- The value zero, for AUTDZB.

If the authentication passes, the upper bits of the address are restored to enable subsequent use of the address. If the authentication fails, the upper bits are corrupted and any subsequent use of the address results in a Translation fault.

Related references

16.1 A64 instructions in alphabetical order on page 16-798.

16.24 AUTIA, AUTIZA, AUTIA1716, AUTIASP, AUTIAZ

Authenticate Instruction address, using key A.

Syntax

`AUTIA Xd, Xn|SP`

`AUTIZA Xd`

`AUTIA1716`

`AUTIASP`

`AUTIAZ`

Where:

Xd
Is the 64-bit name of the general-purpose destination register.

Xn|SP
Is the 64-bit name of the general-purpose source register or stack pointer.

Architectures supported

Supported in the Armv8.3-A architecture and later.

Usage

Authenticate Instruction address, using key A. This instruction authenticates an instruction address, using a modifier and key A.

The address is:

- In the general-purpose register that is specified by *Xd* for `AUTIA` and `AUTIZA`.
- In X17, for `AUTIA1716`.
- In X30, for `AUTIASP` and `AUTIAZ`.

The modifier is:

- In the general-purpose register or stack pointer that is specified by *Xn|SP* for `AUTIA`.
- The value zero, for `AUTIZA` and `AUTIAZ`.
- In X16, for `AUTIA1716`.
- In SP, for `AUTIASP`.

If the authentication passes, the upper bits of the address are restored to enable subsequent use of the address. If the authentication fails, the upper bits are corrupted and any subsequent use of the address results in a Translation fault.

Related references

16.1 A64 instructions in alphabetical order on page 16-798.

16.25 AUTIB, AUTIZB, AUTIB1716, AUTIBSP, AUTIBZ

Authenticate Instruction address, using key B.

Syntax

`AUTIB Xd, Xn|SP`

`AUTIZB Xd`

`AUTIB1716`

`AUTIBSP`

`AUTIBZ`

Where:

Xd
Is the 64-bit name of the general-purpose destination register.

Xn|SP
Is the 64-bit name of the general-purpose source register or stack pointer.

Architectures supported

Supported in the Armv8.3-A architecture and later.

Usage

Authenticate Instruction address, using key B. This instruction authenticates an instruction address, using a modifier and key B.

The address is:

- In the general-purpose register that is specified by *Xd* for `AUTIB` and `AUTIZB`.
- In X17, for `AUTIB1716`.
- In X30, for `AUTIBSP` and `AUTIBZ`.

The modifier is:

- In the general-purpose register or stack pointer that is specified by *Xn|SP* for `AUTIB`.
- The value zero, for `AUTIZB` and `AUTIBZ`.
- In X16, for `AUTIB1716`.
- In SP, for `AUTIBSP`.

If the authentication passes, the upper bits of the address are restored to enable subsequent use of the address. If the authentication fails, the upper bits are corrupted and any subsequent use of the address results in a Translation fault.

Related references

16.1 A64 instructions in alphabetical order on page 16-798.

16.26 B.cond

Branch conditionally.

Syntax

`B.cond label`

Where:

cond
Is one of the standard conditions.

label
Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range ±1MB.

Usage

Branch conditionally to a label at a PC-relative offset, with a hint that this is not a subroutine call or return.

Related references

7.12 Condition code suffixes and related flags on page 7-151.
16.1 A64 instructions in alphabetical order on page 16-798.

16.27 B

Branch.

Syntax

`B label`

Where:

label

Is the program label to be unconditionally branched to. Its offset from the address of this instruction, in the range ±128MB. The branch can be forward or backward within 128MB.

Usage

Branch causes an unconditional branch to a label at a PC-relative offset, with a hint that this is not a subroutine call or return.

Related references

16.1 A64 instructions in alphabetical order on page 16-798.

16.28 BFC

Bitfield Clear, leaving other bits unchanged.

This instruction is an alias of `BFM`.

The equivalent instruction is `BFM Wd, WZR, #(-lsb MOD 32), #(width-1)`.

Syntax

```
BFC Wd, #lsb, #width ; 32-bit

BFC Xd, #lsb, #width ; 64-bit
```

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

lsb
Depends on the instruction variant:

32-bit general registers
Is the bit number of the lsb of the destination bitfield, in the range 0 to 31.

64-bit general registers
Is the bit number of the lsb of the destination bitfield, in the range 0 to 63.

width
Depends on the instruction variant:

32-bit general registers
Is the width of the bitfield, in the range 1 to 32-*lsb*.

64-bit general registers
Is the width of the bitfield, in the range 1 to 64-*lsb*.

Xd
Is the 64-bit name of the general-purpose destination register.

Architectures supported

Supported in the Armv8.2 architecture and later.

Related references

16.30 BFM on page 16-835.
16.1 A64 instructions in alphabetical order on page 16-798.

16.29 BFI

Bitfield Insert.

This instruction is an alias of `BFM`.

The equivalent instruction is `BFM Wd, Wn, #(-lsb MOD 32), #(width-1)`.

Syntax

```
BFI Wd, Wn, #lsb, #width ; 32-bit

BFI Xd, Xn, #lsb, #width ; 64-bit
```

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

Wn
Is the 32-bit name of the general-purpose source register.

lsb
Depends on the instruction variant:

32-bit general registers
Is the bit number of the lsb of the destination bitfield, in the range 0 to 31.

64-bit general registers
Is the bit number of the lsb of the destination bitfield, in the range 0 to 63.

width
Depends on the instruction variant:

32-bit general registers
Is the width of the bitfield, in the range 1 to 32-*lsb*.

64-bit general registers
Is the width of the bitfield, in the range 1 to 64-*lsb*.

Xd
Is the 64-bit name of the general-purpose destination register.

Xn
Is the 64-bit name of the general-purpose source register.

Usage

Bitfield Insert copies any number of low-order bits from a source register into the same number of adjacent bits at any position in the destination register, leaving other bits unchanged.

Related references

16.30 BFM on page 16-835.
16.1 A64 instructions in alphabetical order on page 16-798.

16.30 BFM

Bitfield Move.

This instruction is used by the aliases:

- `BFC`.
- `BFI`.
- `BFXIL`.

Syntax

```
BFM Wd, Wn, #<immr>, #<imms> ; 32-bit

BFM Xd, Xn, #<immr>, #<imms> ; 64-bit
```

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

Wn
Is the 32-bit name of the general-purpose source register.

<immr>
Depends on the instruction variant:

32-bit general registers
Is the right rotate amount, in the range 0 to 31.

64-bit general registers
Is the right rotate amount, in the range 0 to 63.

<imms>
Depends on the instruction variant:

32-bit general registers
Is the leftmost bit number to be moved from the source, in the range 0 to 31.

64-bit general registers
Is the leftmost bit number to be moved from the source, in the range 0 to 63.

Xd
Is the 64-bit name of the general-purpose destination register.

Xn
Is the 64-bit name of the general-purpose source register.

Usage

Bitfield Move copies any number of low-order bits from a source register into the same number of adjacent bits at any position in the destination register, leaving other bits unchanged.

Related references

16.28 BFC on page 16-833.
16.29 BFI on page 16-834.
16.31 BFXIL on page 16-836.
16.1 A64 instructions in alphabetical order on page 16-798.

16.31 BFXIL

Bitfield extract and insert at low end.

This instruction is an alias of `BFM`.

The equivalent instruction is `BFM Wd, Wn, #lsb, #(lsb+width-1)`.

Syntax

```
BFXIL Wd, Wn, #lsb, #width ; 32-bit
BFXIL Xd, Xn, #lsb, #width ; 64-bit
```

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

Wn
Is the 32-bit name of the general-purpose source register.

lsb
Depends on the instruction variant:

32-bit general registers
Is the bit number of the lsb of the source bitfield, in the range 0 to 31.

64-bit general registers
Is the bit number of the lsb of the source bitfield, in the range 0 to 63.

width
Depends on the instruction variant:

32-bit general registers
Is the width of the bitfield, in the range 1 to 32-*lsb*.

64-bit general registers
Is the width of the bitfield, in the range 1 to 64-*lsb*.

Xd
Is the 64-bit name of the general-purpose destination register.

Xn
Is the 64-bit name of the general-purpose source register.

Usage

Bitfield extract and insert at low end copies any number of low-order bits from a source register into the same number of adjacent bits at the low end in the destination register, leaving other bits unchanged.

Related references

16.30 BFM on page 16-835.
16.1 A64 instructions in alphabetical order on page 16-798.

16.32 BIC (shifted register)

Bitwise Bit Clear (shifted register).

Syntax

```
BIC Wd, Wn, Wm{, shift #amount} ; 32-bit
BIC Xd, Xn, Xm{, shift #amount} ; 64-bit
```

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

Wn
Is the 32-bit name of the first general-purpose source register.

Wm
Is the 32-bit name of the second general-purpose source register.

amount
Depends on the instruction variant:

32-bit general registers
Is the shift amount, in the range 0 to 31, defaulting to 0.

64-bit general registers
Is the shift amount, in the range 0 to 63, defaulting to 0.

Xd
Is the 64-bit name of the general-purpose destination register.

Xn
Is the 64-bit name of the first general-purpose source register.

Xm
Is the 64-bit name of the second general-purpose source register.

shift
Is the optional shift to be applied to the final source, defaulting to LSL, and can be one of `LSL`, `LSR`, `ASR`, or `ROR`.

Operation

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

`Rd = Rn AND NOT shift(Rm, amount)`, where `R` is either `W` or `X`.

Related references

16.1 A64 instructions in alphabetical order on page 16-798.

16.33 BICS (shifted register)

Bitwise Bit Clear (shifted register), setting flags.

Syntax

```
BICS Wd, Wn, Wm{, shift #amount} ; 32-bit

BICS Xd, Xn, Xm{, shift #amount} ; 64-bit
```

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

Wn
Is the 32-bit name of the first general-purpose source register.

Wm
Is the 32-bit name of the second general-purpose source register.

amount
Depends on the instruction variant:

32-bit general registers
Is the shift amount, in the range 0 to 31, defaulting to 0.

64-bit general registers
Is the shift amount, in the range 0 to 63, defaulting to 0.

Xd
Is the 64-bit name of the general-purpose destination register.

Xn
Is the 64-bit name of the first general-purpose source register.

Xm
Is the 64-bit name of the second general-purpose source register.

shift
Is the optional shift to be applied to the final source, defaulting to LSL, and can be one of `LSL`, `LSR`, `ASR`, or `ROR`.

Operation

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

`Rd = Rn AND NOT shift(Rm, amount)`, where `R` is either `W` or `X`.

Related references

16.1 A64 instructions in alphabetical order on page 16-798.

16.34 BL

Branch with Link.

Syntax

`BL label`

Where:

label

Is the program label to be unconditionally branched to. Its offset from the address of this instruction, in the range ±128MB. The branch can be forward or backward within 128MB.

Usage

Branch with Link branches to a PC-relative offset, setting the register X30 to PC+4. It provides a hint that this is a subroutine call.

Related references

16.1 A64 instructions in alphabetical order on page 16-798.

16.35 BLR

Branch with Link to Register.

Syntax

`BLR` *Xn*

Where:

Xn
Is the 64-bit name of the general-purpose register holding the address to be branched to.

Usage

Branch with Link to Register calls a subroutine at an address in a register, setting register X30 to PC+4.

Related references

16.1 A64 instructions in alphabetical order on page 16-798.

16.36 BLRAA, BLRAAZ, BLRAB, BLRABZ

Branch with Link to Register, with pointer authentication.

Syntax

```
BLRAA Xn, Xm|SP ; BLRAA general registers

BLRAAZ Xn ; BLRAAZ general registers

BLRAB Xn, Xm|SP ; BLRAB general registers

BLRABZ Xn ; BLRABZ general registers
```

Where:

Xn
Is the 64-bit name of the general-purpose register holding the address to be branched to.

Xm|SP
Is the 64-bit name of the general-purpose source register or stack pointer holding the modifier.

Architectures supported

Supported in the Armv8.3-A architecture and later.

Usage

Branch with Link to Register, with pointer authentication. This instruction authenticates the address in the general-purpose register that is specified by *Xn*, using a modifier and the specified key, and calls a subroutine at the authenticated address, setting register X30 to PC+4.

The modifier is:

- In the general-purpose register or stack pointer that is specified by *Xm|SP* for BLRAA and BLRAB.
- The value zero, for BLRAAZ and BLRABZ.

Key A is used for BLRAA and BLRAAZ, and key B is used for BLRAB and BLRABZ.

If the authentication passes, the PE continues execution at the target of the branch. If the authentication fails, a Translation fault is generated.

The authenticated address is not written back to the general-purpose register.

Related references

16.1 A64 instructions in alphabetical order on page 16-798.

16.37 BR

Branch to Register.

Syntax

BR *Xn*

Where:

Xn
Is the 64-bit name of the general-purpose register holding the address to be branched to.

Usage

Branch to Register branches unconditionally to an address in a register, with a hint that this is not a subroutine return.

Related references

16.1 A64 instructions in alphabetical order on page 16-798.

16.38 BRAA, BRAAZ, BRAB, BRABZ

Branch to Register, with pointer authentication.

Syntax

```
BRAA Xn, Xm|SP ; BRAA general registers

BRAAZ Xn ; BRAAZ general registers

BRAB Xn, Xm|SP ; BRAB general registers

BRABZ Xn ; BRABZ general registers
```

Where:

Xn
Is the 64-bit name of the general-purpose register holding the address to be branched to.

Xm|SP
Is the 64-bit name of the general-purpose source register or stack pointer holding the modifier.

Architectures supported

Supported in the Armv8.3-A architecture and later.

Usage

Branch to Register, with pointer authentication. This instruction authenticates the address in the general-purpose register that is specified by *Xn*, using a modifier and the specified key, and branches to the authenticated address.

The modifier is:

- In the general-purpose register or stack pointer that is specified by *Xm|SP* for BRAA and BRAB.
- The value zero, for BRAAZ and BRABZ.

Key A is used for BRAA and BRAAZ, and key B is used for BRAB and BRABZ.

If the authentication passes, the PE continues execution at the target of the branch. If the authentication fails, a Translation fault is generated.

The authenticated address is not written back to the general-purpose register.

Related references

16.1 A64 instructions in alphabetical order on page 16-798.

16.39 BRK

Breakpoint instruction.

Syntax

`BRK #imm`

Where:

imm
Is a 16-bit unsigned immediate, in the range 0 to 65535.

Usage

Breakpoint instruction generates a Breakpoint Instruction exception. The PE records the exception in ESR_ELx, using the EC value `0x3c`, and captures the value of the immediate argument in ESR_ELx.ISS.

Related references

16.1 A64 instructions in alphabetical order on page 16-798.

16.40 CBNZ

Compare and Branch on Nonzero.

Syntax

```
CBNZ Wt, label ; 32-bit
CBNZ Xt, label ; 64-bit
```

Where:

Wt
Is the 32-bit name of the general-purpose register to be tested.

Xt
Is the 64-bit name of the general-purpose register to be tested.

label
Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range ±1MB.

Usage

Compare and Branch on Nonzero compares the value in a register with zero, and conditionally branches to a label at a PC-relative offset if the comparison is not equal. It provides a hint that this is not a subroutine call or return. This instruction does not affect the condition flags.

Related references

16.1 A64 instructions in alphabetical order on page 16-798.

16.41 CBZ

Compare and Branch on Zero.

Syntax

```
CBZ Wt, label ; 32-bit

CBZ Xt, label ; 64-bit
```

Where:

Wt
Is the 32-bit name of the general-purpose register to be tested.

Xt
Is the 64-bit name of the general-purpose register to be tested.

label
Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range ±1MB.

Usage

Compare and Branch on Zero compares the value in a register with zero, and conditionally branches to a label at a PC-relative offset if the comparison is equal. It provides a hint that this is not a subroutine call or return. This instruction does not affect condition flags.

Related references

16.1 A64 instructions in alphabetical order on page 16-798.

16.42 CCMN (immediate)

Conditional Compare Negative (immediate).

Syntax

```
CCMN Wn, #imm, #nzcv, cond ; 32-bit

CCMN Xn, #imm, #nzcv, cond ; 64-bit
```

Where:

Wn
Is the 32-bit name of the first general-purpose source register.

Xn
Is the 64-bit name of the first general-purpose source register.

imm
Is a five bit unsigned immediate.

nzcv
Is the flag bit specifier, an immediate in the range 0 to 15, giving the alternative state for the 4-bit NZCV condition flags.

cond
Is one of the standard conditions.

Operation

Conditional Compare Negative (immediate) sets the value of the condition flags to the result of the comparison of a register value and a negated immediate value if the condition is TRUE, and an immediate value otherwise.

`flags = if cond then compare(Rn, #-imm) else #nzcv`, where `R` is either `W` or `X`.

Related references

7.12 Condition code suffixes and related flags on page 7-151.
16.1 A64 instructions in alphabetical order on page 16-798.

16.43 CCMN (register)

Conditional Compare Negative (register).

Syntax

```
CCMN Wn, Wm, #nzcv, cond ; 32-bit
CCMN Xn, Xm, #nzcv, cond ; 64-bit
```

Where:

Wn
Is the 32-bit name of the first general-purpose source register.

Wm
Is the 32-bit name of the second general-purpose source register.

Xn
Is the 64-bit name of the first general-purpose source register.

Xm
Is the 64-bit name of the second general-purpose source register.

nzcv
Is the flag bit specifier, an immediate in the range 0 to 15, giving the alternative state for the 4-bit NZCV condition flags.

cond
Is one of the standard conditions.

Operation

Conditional Compare Negative (register) sets the value of the condition flags to the result of the comparison of a register value and the inverse of another register value if the condition is TRUE, and an immediate value otherwise.

`flags = if cond then compare(Rn, -Rm) else #nzcv`, where `R` is either `W` or `X`.

Related references

7.12 Condition code suffixes and related flags on page 7-151.
16.1 A64 instructions in alphabetical order on page 16-798.

16.44 CCMP (immediate)

Conditional Compare (immediate).

Syntax

```
CCMP Wn, #imm, #nzcv, cond ; 32-bit

CCMP Xn, #imm, #nzcv, cond ; 64-bit
```

Where:

Wn
Is the 32-bit name of the first general-purpose source register.

Xn
Is the 64-bit name of the first general-purpose source register.

imm
Is a five bit unsigned immediate.

nzcv
Is the flag bit specifier, an immediate in the range 0 to 15, giving the alternative state for the 4-bit NZCV condition flags.

cond
Is one of the standard conditions.

Operation

Conditional Compare (immediate) sets the value of the condition flags to the result of the comparison of a register value and an immediate value if the condition is TRUE, and an immediate value otherwise.

`flags = if cond then compare(Rn, #imm) else #nzcv`, where `R` is either `W` or `X`.

Related references

16.1 A64 instructions in alphabetical order on page 16-798.

16.45 CCMP (register)

Conditional Compare (register).

Syntax

```
CCMP Wn, Wm, #nzcv, cond ; 32-bit
CCMP Xn, Xm, #nzcv, cond ; 64-bit
```

Where:

Wn
Is the 32-bit name of the first general-purpose source register.

Wm
Is the 32-bit name of the second general-purpose source register.

Xn
Is the 64-bit name of the first general-purpose source register.

Xm
Is the 64-bit name of the second general-purpose source register.

nzcv
Is the flag bit specifier, an immediate in the range 0 to 15, giving the alternative state for the 4-bit NZCV condition flags.

cond
Is one of the standard conditions.

Operation

Conditional Compare (register) sets the value of the condition flags to the result of the comparison of two registers if the condition is TRUE, and an immediate value otherwise.

`flags = if cond then compare(Rn, Rm) else #nzcv`, where *R* is either `W` or `X`.

Related references

7.12 Condition code suffixes and related flags on page 7-151.
16.1 A64 instructions in alphabetical order on page 16-798.

16.46 CINC

Conditional Increment.

This instruction is an alias of `CSINC`.

The equivalent instruction is `CSINC Wd, Wn, Wn, invert(cond)`.

Syntax

```
CINC Wd, Wn, cond ; 32-bit
CINC Xd, Xn, cond ; 64-bit
```

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

Wn
Is the 32-bit name of the general-purpose source register.

Xd
Is the 64-bit name of the general-purpose destination register.

Xn
Is the 64-bit name of the general-purpose source register.

cond
Is one of the standard conditions, excluding AL and NV.

Operation

Conditional Increment returns, in the destination register, the value of the source register incremented by 1 if the condition is TRUE, and otherwise returns the value of the source register.

`Rd = if cond then Rn+1 else Rn`, where `R` is either `W` or `X`.

Related references

16.63 CSINC on page 16-870.
7.12 Condition code suffixes and related flags on page 7-151.
16.1 A64 instructions in alphabetical order on page 16-798.

16.47 CINV

Conditional Invert.

This instruction is an alias of `CSINV`.

The equivalent instruction is `CSINV Wd, Wn, Wn, invert(cond)`.

Syntax

```
CINV Wd, Wn, cond ; 32-bit
CINV Xd, Xn, cond ; 64-bit
```

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

Wn
Is the 32-bit name of the general-purpose source register.

Xd
Is the 64-bit name of the general-purpose destination register.

Xn
Is the 64-bit name of the general-purpose source register.

cond
Is one of the standard conditions, excluding AL and NV.

Operation

Conditional Invert returns, in the destination register, the bitwise inversion of the value of the source register if the condition is TRUE, and otherwise returns the value of the source register.

`Rd = if cond then NOT(Rn) else Rn`, where `R` is either `W` or `X`.

Related references

16.64 CSINV on page 16-871.
7.12 Condition code suffixes and related flags on page 7-151.
16.1 A64 instructions in alphabetical order on page 16-798.

16.48 CLREX

Clear Exclusive.

Syntax

```
CLREX {#imm}
```

Where:

imm
Is an optional 4-bit unsigned immediate, in the range 0 to 15, defaulting to 15.

Usage

Clear Exclusive clears the local monitor of the executing PE.

Related references

16.1 A64 instructions in alphabetical order on page 16-798.

16.49 CLS

Count leading sign bits.

Syntax

```
CLS Wd, Wn ; 32-bit
CLS Xd, Xn ; 64-bit
```

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

Wn
Is the 32-bit name of the general-purpose source register.

Xd
Is the 64-bit name of the general-purpose destination register.

Xn
Is the 64-bit name of the general-purpose source register.

Operation

`Rd = CLS(Rn)`, where `R` is either `W` or `X`.

Related references

16.1 A64 instructions in alphabetical order on page 16-798.

16.50 CLZ

Count leading zero bits.

Syntax

```
CLZ Wd, Wn ; 32-bit
CLZ Xd, Xn ; 64-bit
```

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

Wn
Is the 32-bit name of the general-purpose source register.

Xd
Is the 64-bit name of the general-purpose destination register.

Xn
Is the 64-bit name of the general-purpose source register.

Operation

`Rd = CLZ(Rn)`, where `R` is either `W` or `X`.

Related references

16.1 A64 instructions in alphabetical order on page 16-798.

16.51 CMN (extended register)

Compare Negative (extended register).

This instruction is an alias of ADDS (extended register).

The equivalent instruction is ADDS WZR, *Wn|WSP*, *Wm*{, *extend* {#*amount*}}.

Syntax

CMN *Wn|WSP*, *Wm*{, *extend* {#*amount*}} ; 32-bit

CMN *Xn|SP*, *Rm*{, *extend* {#*amount*}} ; 64-bit

Where:

Wn|WSP
Is the 32-bit name of the first source general-purpose register or stack pointer.

Wm
Is the 32-bit name of the second general-purpose source register.

extend
Is the extension to be applied to the second source operand:

32-bit general registers
Can be one of UXTB, UXTH, LSL|UXTW, UXTX, SXTB, SXTH, SXTW or SXTX.

If *Rn* is WSP then LSL is preferred rather than UXTW, and can be omitted when *amount* is 0. In all other cases *extend* is required and must be UXTW rather than LSL.

64-bit general registers
Can be one of UXTB, UXTH, UXTW, LSL|UXTX, SXTB, SXTH, SXTW or SXTX.

If *Rn* is SP then LSL is preferred rather than UXTX, and can be omitted when *amount* is 0. In all other cases *extend* is required and must be UXTX rather than LSL.

Xn|SP
Is the 64-bit name of the first source general-purpose register or stack pointer.

R
Is a width specifier, and can be either W or X.

m
Is the number [0-30] of the second general-purpose source register or the name ZR (31).

amount
Is the left shift amount to be applied after extension in the range 0 to 4, defaulting to 0. It must be absent when *extend* is absent, is required when *extend* is LSL, and is optional when *extend* is present but not LSL.

Operation

Compare Negative (extended register) adds a register value and a sign or zero-extended register value, followed by an optional left shift amount. The argument that is extended from the *Rm* register can be a byte, halfword, word, or doubleword. It updates the condition flags based on the result, and discards the result.

*R*n + LSL(*extend*(*R*m), *amount*), where *R* is either W or X.

Usage

Table 16-5 CMN (64-bit general registers) specifier combinations

R	*extend*
W	SXTB
W	SXTH

Table 16-5 CMN (64-bit general registers) specifier combinations (continued)

R	*extend*
W	SXTW
W	UXTB
W	UXTH
W	UXTW
X	LSL|UXTX
X	SXTX

Related references

16.8 ADDS (extended register) on page 16-811.
16.1 A64 instructions in alphabetical order on page 16-798.

16.52 CMN (immediate)

Compare Negative (immediate).

This instruction is an alias of ADDS (immediate).

The equivalent instruction is `ADDS WZR, Wn|WSP, #imm {, shift}`.

Syntax

`CMN Wn|WSP, #imm{, shift} ; 32-bit`

`CMN Xn|SP, #imm{, shift} ; 64-bit`

Where:

Wn|WSP
Is the 32-bit name of the source general-purpose register or stack pointer.

Xn|SP
Is the 64-bit name of the source general-purpose register or stack pointer.

imm
Is an unsigned immediate, in the range 0 to 4095.

shift
Is the optional left shift to apply to the immediate, defaulting to `LSL #0`, and can be either `LSL #0` or `LSL #12`.

Operation

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

`Rn + shift(imm)`, where `R` is either `W` or `X`.

Related references

16.9 ADDS (immediate) on page 16-813.
16.1 A64 instructions in alphabetical order on page 16-798.

16.53 CMN (shifted register)

Compare Negative (shifted register).

This instruction is an alias of ADDS (shifted register).

The equivalent instruction is `ADDS WZR, Wn, Wm {, shift #amount}`.

Syntax

```
CMN Wn, Wm{, shift #amount} ; 32-bit
CMN Xn, Xm{, shift #amount} ; 64-bit
```

Where:

Wn
Is the 32-bit name of the first general-purpose source register.

Wm
Is the 32-bit name of the second general-purpose source register.

amount
Depends on the instruction variant:

32-bit general registers
Is the shift amount, in the range 0 to 31, defaulting to 0.

64-bit general registers
Is the shift amount, in the range 0 to 63, defaulting to 0.

Xn
Is the 64-bit name of the first general-purpose source register.

Xm
Is the 64-bit name of the second general-purpose source register.

shift
Is the optional shift type to be applied to the second source operand, defaulting to LSL, and can be one of LSL, LSR, or ASR.

Operation

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

`Rn + shift(Rm, amount)`, where *R* is either W or X.

Related references

16.10 ADDS (shifted register) on page 16-814.
16.1 A64 instructions in alphabetical order on page 16-798.

16.54 CMP (extended register)

Compare (extended register).

This instruction is an alias of SUBS (extended register).

The equivalent instruction is `SUBS WZR, Wn|WSP, Wm{, extend {#amount}}`.

Syntax

```
CMP Wn|WSP, Wm{, extend {#amount}} ; 32-bit
CMP Xn|SP, Rm{, extend {#amount}} ; 64-bit
```

Where:

Wn|WSP
Is the 32-bit name of the first source general-purpose register or stack pointer.

Wm
Is the 32-bit name of the second general-purpose source register.

extend
Is the extension to be applied to the second source operand:

32-bit general registers
Can be one of UXTB, UXTH, LSL|UXTW, UXTX, SXTB, SXTH, SXTW or SXTX.

If *Rn* is WSP then LSL is preferred rather than UXTW, and can be omitted when *amount* is 0. In all other cases *extend* is required and must be UXTW rather than LSL.

64-bit general registers
Can be one of UXTB, UXTH, UXTW, LSL|UXTX, SXTB, SXTH, SXTW or SXTX.

If *Rn* is SP then LSL is preferred rather than UXTX, and can be omitted when *amount* is 0. In all other cases *extend* is required and must be UXTX rather than LSL.

Xn|SP
Is the 64-bit name of the first source general-purpose register or stack pointer.

R
Is a width specifier, and can be either W or X.

m
Is the number [0-30] of the second general-purpose source register or the name ZR (31).

amount
Is the left shift amount to be applied after extension in the range 0 to 4, defaulting to 0. It must be absent when *extend* is absent, is required when *extend* is LSL, and is optional when *extend* is present but not LSL.

Operation

Compare (extended register) subtracts a sign or zero-extended register value, followed by an optional left shift amount, from a register value. The argument that is extended from the *Rm* register can be a byte, halfword, word, or doubleword. It updates the condition flags based on the result, and discards the result.

`Rn - LSL(extend(Rm), amount)`, where *R* is either W or X.

Usage

Table 16-6 CMP (64-bit general registers) specifier combinations

R	*extend*
W	SXTB
W	SXTH
W	SXTW

Table 16-6 CMP (64-bit general registers) specifier combinations (continued)

R	*extend*
W	UXTB
W	UXTH
W	UXTW
X	LSL|UXTX
X	SXTX

Related references

16.149 SUBS (extended register) on page 16-959.
16.1 A64 instructions in alphabetical order on page 16-798.

16.55 CMP (immediate)

Compare (immediate).

This instruction is an alias of SUBS (immediate).

The equivalent instruction is `SUBS WZR, Wn|WSP, #imm {, shift}`.

Syntax

```
CMP Wn|WSP, #imm{, shift} ; 32-bit
CMP Xn|SP, #imm{, shift} ; 64-bit
```

Where:

Wn|WSP
Is the 32-bit name of the source general-purpose register or stack pointer.

Xn|SP
Is the 64-bit name of the source general-purpose register or stack pointer.

imm
Is an unsigned immediate, in the range 0 to 4095.

shift
Is the optional left shift to apply to the immediate, defaulting to `LSL #0`, and can be either `LSL #0` or `LSL #12`.

Operation

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

`Rn - shift(imm)`, where `R` is either `W` or `X`.

Related references

16.150 SUBS (immediate) on page 16-961.
16.1 A64 instructions in alphabetical order on page 16-798.

16.56 CMP (shifted register)

Compare (shifted register).

This instruction is an alias of SUBS (shifted register).

The equivalent instruction is `SUBS WZR, Wn, Wm {, shift #amount}`.

Syntax

```
CMP Wn, Wm{, shift #amount} ; 32-bit
CMP Xn, Xm{, shift #amount} ; 64-bit
```

Where:

Wn
Is the 32-bit name of the first general-purpose source register.

Wm
Is the 32-bit name of the second general-purpose source register.

amount
Depends on the instruction variant:

32-bit general registers
Is the shift amount, in the range 0 to 31, defaulting to 0.

64-bit general registers
Is the shift amount, in the range 0 to 63, defaulting to 0.

Xn
Is the 64-bit name of the first general-purpose source register.

Xm
Is the 64-bit name of the second general-purpose source register.

shift
Is the optional shift type to be applied to the second source operand, defaulting to LSL, and can be one of LSL, LSR, or ASR.

Operation

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

`Rn - shift(Rm,amount)`, where *R* is either W or X.

Related references

16.151 SUBS (shifted register) on page 16-962.
16.1 A64 instructions in alphabetical order on page 16-798.

16.57 CNEG

Conditional Negate.

This instruction is an alias of `CSNEG`.

The equivalent instruction is `CSNEG Wd, Wn, Wn, invert(cond)`.

Syntax

```
CNEG Wd, Wn, cond ; 32-bit

CNEG Xd, Xn, cond ; 64-bit
```

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

Wn
Is the 32-bit name of the general-purpose source register.

Xd
Is the 64-bit name of the general-purpose destination register.

Xn
Is the 64-bit name of the general-purpose source register.

cond
Is one of the standard conditions, excluding AL and NV.

Operation

Conditional Negate returns, in the destination register, the negated value of the source register if the condition is TRUE, and otherwise returns the value of the source register.

`Rd = if cond then -Rn else Rn`, where `R` is either `W` or `X`.

Related references

16.65 CSNEG on page 16-872.
7.12 Condition code suffixes and related flags on page 7-151.
16.1 A64 instructions in alphabetical order on page 16-798.

16.58 CRC32B, CRC32H, CRC32W, CRC32X

CRC32 checksum performs a cyclic redundancy check (CRC) calculation on a value held in a general-purpose register.

Syntax

```
CRC32B Wd, Wn, Wm ; Wd = CRC32(Wn, Rm[<7:0>])

CRC32H Wd, Wn, Wm ; Wd = CRC32(Wn, Rm[<15:0>])

CRC32W Wd, Wn, Wm ; Wd = CRC32(Wn, Rm[<31:0>])

CRC32X Wd, Wn, Xm ; Wd = CRC32(Wn, Rm[<63:0>])
```

Where:

Wm
Is the 32-bit name of the general-purpose data source register.

Xm
Is the 64-bit name of the general-purpose data source register.

Wd
Is the 32-bit name of the general-purpose accumulator output register.

Wn
Is the 32-bit name of the general-purpose accumulator input register.

Operation

This instruction 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, 32, or 64 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_AA64ISAR0_EL1.CRC32 indicates whether this instruction is supported. See *ID_AA64ISAR0_EL1* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

```
Wd = CRC32(Wn, Rm<n:0>) // n = 7, 15, 31, 63.
```

Architectures supported

Supported in architecture Armv8.1 and later. Optionally supported in Armv8-A.

Related references

16.1 A64 instructions in alphabetical order on page 16-798.

16.59 CRC32CB, CRC32CH, CRC32CW, CRC32CX

CRC32C checksum performs a cyclic redundancy check (CRC) calculation on a value held in a general-purpose register.

Syntax

```
CRC32CB Wd, Wn, Wm ; Wd = CRC32C(Wn, Rm[<7:0>])

CRC32CH Wd, Wn, Wm ; Wd = CRC32C(Wn, Rm[<15:0>])

CRC32CW Wd, Wn, Wm ; Wd = CRC32C(Wn, Rm[<31:0>])

CRC32CX Wd, Wn, Xm ; Wd = CRC32C(Wn, Rm[<63:0>])
```

Where:

Wm
Is the 32-bit name of the general-purpose data source register.

Xm
Is the 64-bit name of the general-purpose data source register.

Wd
Is the 32-bit name of the general-purpose accumulator output register.

Wn
Is the 32-bit name of the general-purpose accumulator input register.

Operation

This instruction 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, 32, or 64 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_AA64ISAR0_EL1.CRC32 indicates whether this instruction is supported. See *ID_AA64ISAR0_EL1* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

```
Wd = CRC32C(Wn, Rm<n:0>) // n = 7, 15, 31, 63.
```

Architectures supported

Supported in architecture Armv8.1 and later. Optionally supported in Armv8-A.

Related references

16.1 A64 instructions in alphabetical order on page 16-798.

16.60 CSEL

Conditional Select.

Syntax

```
CSEL Wd, Wn, Wm, cond ; 32-bit
CSEL Xd, Xn, Xm, cond ; 64-bit
```

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

Wn
Is the 32-bit name of the first general-purpose source register.

Wm
Is the 32-bit name of the second general-purpose source register.

Xd
Is the 64-bit name of the general-purpose destination register.

Xn
Is the 64-bit name of the first general-purpose source register.

Xm
Is the 64-bit name of the second general-purpose source register.

cond
Is one of the standard conditions.

Operation

Conditional Select returns, in the destination register, the value of the first source register if the condition is TRUE, and otherwise returns the value of the second source register.

`Rd = if cond then Rn else Rm`, where *R* is either `W` or `X`.

Related references

7.12 Condition code suffixes and related flags on page 7-151.
16.1 A64 instructions in alphabetical order on page 16-798.

16.61 CSET

Conditional Set.

This instruction is an alias of `CSINC`.

The equivalent instruction is `CSINC Wd, WZR, WZR, invert(cond)`.

Syntax

```
CSET Wd, cond ; 32-bit
CSET Xd, cond ; 64-bit
```

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

Xd
Is the 64-bit name of the general-purpose destination register.

cond
Is one of the standard conditions, excluding AL and NV.

Operation

Conditional Set sets the destination register to 1 if the condition is TRUE, and otherwise sets it to 0.

`Rd = if cond then 1 else 0`, where *R* is either `W` or `X`.

Related references

16.63 CSINC on page 16-870.
7.12 Condition code suffixes and related flags on page 7-151.
16.1 A64 instructions in alphabetical order on page 16-798.

16.62 CSETM

Conditional Set Mask.

This instruction is an alias of `CSINV`.

The equivalent instruction is `CSINV Wd, WZR, WZR, invert(cond)`.

Syntax

```
CSETM Wd, cond ; 32-bit

CSETM Xd, cond ; 64-bit
```

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

Xd
Is the 64-bit name of the general-purpose destination register.

cond
Is one of the standard conditions, excluding AL and NV.

Operation

Conditional Set Mask sets all bits of the destination register to 1 if the condition is TRUE, and otherwise sets all bits to 0.

`Rd = if cond then -1 else 0`, where *R* is either `W` or `X`.

Related references

16.64 CSINV on page 16-871.
7.12 Condition code suffixes and related flags on page 7-151.
16.1 A64 instructions in alphabetical order on page 16-798.

16.63 CSINC

Conditional Select Increment.

This instruction is used by the aliases:

- `CINC`.
- `CSET`.

Syntax

`CSINC` *`Wd`*`,` *`Wn`*`,` *`Wm`*`,` *`cond`* `; 32-bit`

`CSINC` *`Xd`*`,` *`Xn`*`,` *`Xm`*`,` *`cond`* `; 64-bit`

Where:

`Wd`
Is the 32-bit name of the general-purpose destination register.

`Wn`
Is the 32-bit name of the first general-purpose source register.

`Wm`
Is the 32-bit name of the second general-purpose source register.

`Xd`
Is the 64-bit name of the general-purpose destination register.

`Xn`
Is the 64-bit name of the first general-purpose source register.

`Xm`
Is the 64-bit name of the second general-purpose source register.

`cond`
Is one of the standard conditions.

Operation

Conditional Select Increment returns, in the destination register, the value of the first source register if the condition is TRUE, and otherwise returns the value of the second source register incremented by 1.

`R``d = if` *`cond`* `then` *`R`*`n else (`*`R`*`m + 1)`, where *`R`* is either `W` or `X`.

Related references

16.46 CINC on page 16-851.
16.61 CSET on page 16-868.
7.12 Condition code suffixes and related flags on page 7-151.
16.1 A64 instructions in alphabetical order on page 16-798.

16.64 CSINV

Conditional Select Invert.

This instruction is used by the aliases:

- `CINV`.
- `CSETM`.

Syntax

`CSINV` *`Wd`*`,` *`Wn`*`,` *`Wm`*`,` *`cond`* `; 32-bit`

`CSINV` *`Xd`*`,` *`Xn`*`,` *`Xm`*`,` *`cond`* `; 64-bit`

Where:

`Wd`
Is the 32-bit name of the general-purpose destination register.

`Wn`
Is the 32-bit name of the first general-purpose source register.

`Wm`
Is the 32-bit name of the second general-purpose source register.

`Xd`
Is the 64-bit name of the general-purpose destination register.

`Xn`
Is the 64-bit name of the first general-purpose source register.

`Xm`
Is the 64-bit name of the second general-purpose source register.

`cond`
Is one of the standard conditions.

Operation

Conditional Select Invert returns, in the destination register, the value of the first source register if the condition is TRUE, and otherwise returns the bitwise inversion value of the second source register.

`R`*`d`* `= if` *`cond`* `then` *`R`*`n else NOT (`*`R`*`m)`, where *`R`* is either `W` or `X`.

Related references

16.47 CINV on page 16-852.
16.62 CSETM on page 16-869.
7.12 Condition code suffixes and related flags on page 7-151.
16.1 A64 instructions in alphabetical order on page 16-798.

16.65 CSNEG

Conditional Select Negation.

This instruction is used by the alias `CNEG`.

Syntax

```
CSNEG Wd, Wn, Wm, cond ; 32-bit

CSNEG Xd, Xn, Xm, cond ; 64-bit
```

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

Wn
Is the 32-bit name of the first general-purpose source register.

Wm
Is the 32-bit name of the second general-purpose source register.

Xd
Is the 64-bit name of the general-purpose destination register.

Xn
Is the 64-bit name of the first general-purpose source register.

Xm
Is the 64-bit name of the second general-purpose source register.

cond
Is one of the standard conditions.

Operation

Conditional Select Negation returns, in the destination register, the value of the first source register if the condition is TRUE, and otherwise returns the negated value of the second source register.

`Rd = if cond then Rn else -Rm`, where `R` is either `W` or `X`.

Related references

16.57 CNEG on page 16-864.
7.12 Condition code suffixes and related flags on page 7-151.
16.1 A64 instructions in alphabetical order on page 16-798.

16.66 DC

Data Cache operation.

This instruction is an alias of SYS.

The equivalent instruction is `SYS #op1, C7, Cm, #op2, Xt`.

Syntax

`DC <dc_op>, Xt`

Where:

`<dc_op>`
Is a DC instruction name, as listed for the DC system instruction group, and can be one of the values shown in Usage.

`op1`
Is a 3-bit unsigned immediate, in the range 0 to 7.

`Cm`
Is a name `Cm`, with `m` in the range 0 to 15.

`op2`
Is a 3-bit unsigned immediate, in the range 0 to 7.

`Xt`
Is the 64-bit name of the general-purpose source register.

Usage

Data Cache operation. For more information, see *A64 system instructions for cache maintenance* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

The following table shows the valid specifier combinations:

Table 16-7 SYS parameter values corresponding to DC operations

<dc_op>	*op1*	*Cm*	*op2*
CISW	0	14	2
CIVAC	3	14	1
CSW	0	10	2
CVAC	3	10	1
CVAP	3	12	1
CVAU	3	11	1
ISW	0	6	2
IVAC	0	6	1
ZVA	3	4	1

Related references

16.156 SYS on page 16-967.
16.1 A64 instructions in alphabetical order on page 16-798.

16.67 DCPS1

Debug Change PE State to EL1.

Syntax

```
DCPS1 {#imm}
```

Where:

imm
Is an optional 16-bit unsigned immediate, in the range 0 to 65535, defaulting to 0.

Usage

Debug Change PE State to EL1, when executed in Debug state:

- If executed at EL0 changes the current Exception level and SP to EL1 using SP_EL1.
- Otherwise, if executed at ELx, selects SP_ELx.

The target exception level of a DCPS1 instruction is:

- EL1 if the instruction is executed at EL0.
- Otherwise, the Exception level at which the instruction is executed.

When the target Exception level of a DCPS1 instruction is ELx, on executing this instruction:

- *ELR_ELx* becomes UNKNOWN.
- *SPSR_ELx* becomes UNKNOWN.
- *ESR_ELx* becomes UNKNOWN.
- *DLR_EL0* and *DSPSR_EL0* become UNKNOWN.
- The endianness is set according to *SCTLR_ELx*.EE.

This instruction is UNDEFINED at EL0 in Non-secure state if EL2 is implemented and *HCR_EL2*.TGE == 1.

This instruction is always UNDEFINED in Non-debug state.

For more information on the operation of the DCPSn instructions, see *DCPS* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

16.1 A64 instructions in alphabetical order on page 16-798.

16.68 DCPS2

Debug Change PE State to EL2.

Syntax

```
DCPS2 {#imm}
```

Where:

imm
Is an optional 16-bit unsigned immediate, in the range 0 to 65535, defaulting to 0.

Usage

Debug Change PE State to EL2, when executed in Debug state:

- If executed at EL0 or EL1 changes the current Exception level and SP to EL2 using SP_EL2.
- Otherwise, if executed at ELx, selects SP_ELx.

The target exception level of a DCPS2 instruction is:

- EL2 if the instruction is executed at an exception level that is not EL3.
- EL3 if the instruction is executed at EL3.

When the target Exception level of a DCPS2 instruction is ELx, on executing this instruction:

- *ELR_ELx* becomes UNKNOWN.
- *SPSR_ELx* becomes UNKNOWN.
- *ESR_ELx* becomes UNKNOWN.
- *DLR_EL0* and *DSPSR_EL0* become UNKNOWN.
- The endianness is set according to *SCTLR_ELx*.EE.

This instruction is UNDEFINED at the following exception levels:

- All exception levels if EL2 is not implemented.
- At EL0 and EL1 in Secure state if EL2 is implemented.

This instruction is always UNDEFINED in Non-debug state.

For more information on the operation of the DCPSn instructions, see *DCPS* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

16.1 A64 instructions in alphabetical order on page 16-798.

16.69 DCPS3

Debug Change PE State to EL3.

Syntax

```
DCPS3 {#imm}
```

Where:

imm
Is an optional 16-bit unsigned immediate, in the range 0 to 65535, defaulting to 0.

Usage

Debug Change PE State to EL3, when executed in Debug state:

- If executed at EL3 selects SP_EL3.
- Otherwise, changes the current Exception level and SP to EL3 using SP_EL3.

The target exception level of a DCPS3 instruction is EL3.

On executing a DCPS3 instruction:

- *ELR_EL3* becomes UNKNOWN.
- *SPSR_EL3* becomes UNKNOWN.
- *ESR_EL3* becomes UNKNOWN.
- *DLR_EL0* and *DSPSR_EL0* become UNKNOWN.
- The endianness is set according to *SCTLR_EL3*.EE.

This instruction is UNDEFINED at all exception levels if either:

- *EDSCR.SDD* == 1.
- EL3 is not implemented.

This instruction is always UNDEFINED in Non-debug state.

For more information on the operation of the DCPSn instructions, see *DCPS* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

16.1 A64 instructions in alphabetical order on page 16-798.

16.70 DMB

Data Memory Barrier.

Syntax

`DMB option|#imm`

Where:

option

Specifies the limitation on the barrier operation. Values are:

SY

Full system is the required shareability domain, reads and writes are the required access types in both *Group A* and *Group B*. This option is referred to as the full system DMB.

ST

Full system is the required shareability domain, writes are the required access type in both *Group A* and *Group B*.

LD

Full system is the required shareability domain, reads are the required access type in *Group A*, and reads and writes are the required access types in *Group B*.

ISH

Inner Shareable is the required shareability domain, reads and writes are the required access types in both *Group A* and *Group B*.

ISHST

Inner Shareable is the required shareability domain, writes are the required access type in both *Group A* and *Group B*.

ISHLD

Inner Shareable is the required shareability domain, reads are the required access type in *Group A*, and reads and writes are the required access types in *Group B*.

NSH

Non-shareable is the required shareability domain, reads and writes are the required access types in both *Group A* and *Group B*.

NSHST

Non-shareable is the required shareability domain, writes are the required access type in both *Group A* and *Group B*.

NSHLD

Non-shareable is the required shareability domain, reads are the required access type in *Group A*, and reads and writes are the required access types in *Group B*.

OSH

Outer Shareable is the required shareability domain, reads and writes are the required access types in both *Group A* and *Group B*.

OSHST

Outer Shareable is the required shareability domain, writes are the required access type in both *Group A* and *Group B*.

OSHLD

Outer Shareable is the required shareability domain, reads are the required access type in *Group A*, and reads and writes are the required access types in *Group B*.

imm

Is a 4-bit unsigned immediate, in the range 0 to 15.

Usage

Data Memory Barrier is a memory barrier that ensures the ordering of observations of memory accesses, see *Data Memory Barrier* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

16.1 A64 instructions in alphabetical order on page 16-798.

16.71 DRPS

Debug restore process state.

Syntax

```
DRPS
```

Related references

16.1 A64 instructions in alphabetical order on page 16-798.

16.72 DSB

Data Synchronization Barrier.

Syntax

`DSB option|#imm`

Where:

option

Specifies the limitation on the barrier operation. Values are:

SY

Full system is the required shareability domain, reads and writes are the required access types in both *Group A* and *Group B*. This option is referred to as the full system DMB.

ST

Full system is the required shareability domain, writes are the required access type in both *Group A* and *Group B*.

LD

Full system is the required shareability domain, reads are the required access type in *Group A*, and reads and writes are the required access types in *Group B*.

ISH

Inner Shareable is the required shareability domain, reads and writes are the required access types in both *Group A* and *Group B*.

ISHST

Inner Shareable is the required shareability domain, writes are the required access type in both *Group A* and *Group B*.

ISHLD

Inner Shareable is the required shareability domain, reads are the required access type in *Group A*, and reads and writes are the required access types in *Group B*.

NSH

Non-shareable is the required shareability domain, reads and writes are the required access types in both *Group A* and *Group B*.

NSHST

Non-shareable is the required shareability domain, writes are the required access type in both *Group A* and *Group B*.

NSHLD

Non-shareable is the required shareability domain, reads are the required access type in *Group A*, and reads and writes are the required access types in *Group B*.

OSH

Outer Shareable is the required shareability domain, reads and writes are the required access types in both *Group A* and *Group B*.

OSHST

Outer Shareable is the required shareability domain, writes are the required access type in both *Group A* and *Group B*.

OSHLD

Outer Shareable is the required shareability domain, reads are the required access type in *Group A*, and reads and writes are the required access types in *Group B*.

imm

Is a 4-bit unsigned immediate, in the range 0 to 15.

Usage

Data Synchronization Barrier is a memory barrier that ensures the completion of memory accesses, see *Data Synchronization Barrier* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

16.1 A64 instructions in alphabetical order on page 16-798.

16.73 EON (shifted register)

Bitwise Exclusive OR NOT (shifted register).

Syntax

```
EON Wd, Wn, Wm{, shift #amount} ; 32-bit
EON Xd, Xn, Xm{, shift #amount} ; 64-bit
```

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

Wn
Is the 32-bit name of the first general-purpose source register.

Wm
Is the 32-bit name of the second general-purpose source register.

amount
Depends on the instruction variant:

32-bit general registers
Is the shift amount, in the range 0 to 31, defaulting to 0.

64-bit general registers
Is the shift amount, in the range 0 to 63, defaulting to 0.

Xd
Is the 64-bit name of the general-purpose destination register.

Xn
Is the 64-bit name of the first general-purpose source register.

Xm
Is the 64-bit name of the second general-purpose source register.

shift
Is the optional shift to be applied to the final source, defaulting to LSL, and can be one of `LSL`, `LSR`, `ASR`, or `ROR`.

Operation

Bitwise Exclusive OR NOT (shifted register) performs a bitwise Exclusive OR NOT of a register value and an optionally-shifted register value, and writes the result to the destination register.

`Rd = Rn EOR NOT shift(Rm, amount)`, where `R` is either `W` or `X`.

Related references

16.1 A64 instructions in alphabetical order on page 16-798.

16.74 EOR (immediate)

Bitwise Exclusive OR (immediate).

Syntax

```
EOR Wd|WSP, Wn, #imm ; 32-bit
```

```
EOR Xd|SP, Xn, #imm ; 64-bit
```

Where:

Wd|WSP
Is the 32-bit name of the destination general-purpose register or stack pointer.

Wn
Is the 32-bit name of the general-purpose source register.

imm
The bitmask immediate.

Xd|SP
Is the 64-bit name of the destination general-purpose register or stack pointer.

Xn
Is the 64-bit name of the general-purpose source register.

Operation

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

`Rd = Rn EOR imm`, where `R` is either `W` or `X`.

Related references

16.1 A64 instructions in alphabetical order on page 16-798.

16.75 EOR (shifted register)

Bitwise Exclusive OR (shifted register).

Syntax

```
EOR Wd, Wn, Wm{, shift #amount} ; 32-bit
EOR Xd, Xn, Xm{, shift #amount} ; 64-bit
```

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

Wn
Is the 32-bit name of the first general-purpose source register.

Wm
Is the 32-bit name of the second general-purpose source register.

amount
Depends on the instruction variant:

32-bit general registers
Is the shift amount, in the range 0 to 31, defaulting to 0.

64-bit general registers
Is the shift amount, in the range 0 to 63, defaulting to 0.

Xd
Is the 64-bit name of the general-purpose destination register.

Xn
Is the 64-bit name of the first general-purpose source register.

Xm
Is the 64-bit name of the second general-purpose source register.

shift
Is the optional shift to be applied to the final source, defaulting to LSL, and can be one of `LSL`, `LSR`, `ASR`, or `ROR`.

Operation

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

`Rd = Rn EOR shift(Rm, amount)`, where `R` is either `W` or `X`.

Related references

16.1 A64 instructions in alphabetical order on page 16-798.

16.76 ERET

Returns from an exception. It restores the processor state based on SPSR_EL*n* and branches to ELR_EL*n*, where *n* is the current exception level..

Syntax

```
ERET
```

Usage

Exception Return using the ELR and SPSR for the current Exception level. When executed, the PE restores PSTATE from the SPSR, and branches to the address held in the ELR.

The PE checks the SPSR for the current Exception level for an illegal return event. See *Illegal return events from AArch64 state* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

`ERET` is UNDEFINED at EL0.

Related references

16.1 A64 instructions in alphabetical order on page 16-798.

16.77 ERETAA, ERETAB

Exception Return, with pointer authentication.

Syntax

ERETAA

ERETAB

Architectures supported

Supported in the Armv8.3-A architecture and later.

Usage

Exception Return, with pointer authentication. This instruction authenticates the address in ELR, using SP as the modifier and the specified key, the PE restores PSTATE from the SPSR for the current Exception level, and branches to the authenticated address.

Key A is used for ERETAA, and key B is used for ERETAB.

If the authentication passes, the PE continues execution at the target of the branch. If the authentication fails, a Translation fault is generated.

The authenticated address is not written back to ELR.

The PE checks the SPSR for the current Exception level for an illegal return event. See *Illegal return events from AArch64 state* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

ERET is UNDEFINED at EL0.

Related references

16.1 A64 instructions in alphabetical order on page 16-798.

16.78 ESB

Error Synchronization Barrier.

Syntax

```
ESB
```

Architectures supported

Supported in the Armv8.2 architecture and later.

Usage

Error Synchronization Barrier.

Related references

16.1 A64 instructions in alphabetical order on page 16-798.

16.79 EXTR

Extract register.

This instruction is used by the alias `ROR` (immediate).

Syntax

```
EXTR Wd, Wn, Wm, #lsb ; 32-bit

EXTR Xd, Xn, Xm, #lsb ; 64-bit
```

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

Wn
Is the 32-bit name of the first general-purpose source register.

Wm
Is the 32-bit name of the second general-purpose source register.

lsb
Depends on the instruction variant:

32-bit general registers
Is the least significant bit position from which to extract, in the range 0 to 31.

64-bit general registers
Is the least significant bit position from which to extract, in the range 0 to 63.

Xd
Is the 64-bit name of the general-purpose destination register.

Xn
Is the 64-bit name of the first general-purpose source register.

Xm
Is the 64-bit name of the second general-purpose source register.

Usage

Extract register extracts a register from a pair of registers.

Related references

16.129 ROR (immediate) on page 16-938.
16.1 A64 instructions in alphabetical order on page 16-798.

16.80 HINT

Hint instruction.

Syntax

```
HINT #imm ; Hints 6 and 7

HINT #imm ; Hints 8 to 15, and 24 to 127

HINT #imm ; Hints 17 to 23
```

Where:

imm

Is a 7-bit unsigned immediate, in the range 0 to 127, but excludes the following:

0

NOP.

1

YIELD.

2

WFE.

3

WFI.

4

SEV.

5

SEVL.

Usage

Hint instruction is for the instruction set space that is reserved for architectural hint instructions.

The encoding variants described here are unallocated in this revision of the architecture, and behave as a NOP. These encodings might be allocated to other hint functionality in future revisions of the architecture.

Related references

16.1 A64 instructions in alphabetical order on page 16-798.

16.81 HLT

Halt instruction.

Syntax

`HLT #imm`

Where:

imm
Is a 16-bit unsigned immediate, in the range 0 to 65535.

Usage

Halt instruction generates a Halt Instruction debug event.

Related references

16.1 A64 instructions in alphabetical order on page 16-798.

16.82 HVC

Hypervisor call to allow OS code to call the Hypervisor. It generates an exception targeting exception level 2 (EL2).

Syntax

```
HVC #imm
```

Where:

imm
: Is a 16-bit unsigned immediate, in the range 0 to 65535. This value is made available to the handler in the Exception Syndrome Register.

Usage

Hypervisor Call causes an exception to EL2. Non-secure software executing at EL1 can use this instruction to call the hypervisor to request a service.

The HVC instruction is UNDEFINED:

- At EL0, and Secure EL1.
- When SCR_EL3.HCE is set to 0.

On executing an HVC instruction, the PE records the exception as a Hypervisor Call exception in *ESR_ELx*, using the EC value 0x16, and the value of the immediate argument.

Related references

16.1 A64 instructions in alphabetical order on page 16-798.

16.83 IC

Instruction Cache operation.

This instruction is an alias of SYS.

The equivalent instruction is `SYS #op1, C7, Cm, #op2{, Xt}`.

Syntax

```
IC <ic_op>{, Xt}
```

Where:

<ic_op>
Is an IC instruction name, as listed for the IC system instruction pages, and can be one of the values shown in Usage.

op1
Is a 3-bit unsigned immediate, in the range 0 to 7.

Cm
Is a name Cm, with m in the range 0 to 15.

op2
Is a 3-bit unsigned immediate, in the range 0 to 7.

Xt
Is the 64-bit name of the optional general-purpose source register, defaulting to 31.

Usage

Instruction Cache operation. For more information, see *A64 system instructions for cache maintenance* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

The following table shows the valid specifier combinations:

Table 16-8 SYS parameter values corresponding to IC operations

<ic_op>	*op1*	*Cm*	*op2*
IALLU	0	5	0
IALLUIS	0	1	0
IVAU	3	5	1

Related references

16.156 SYS on page 16-967.
16.1 A64 instructions in alphabetical order on page 16-798.

16.84 ISB

Instruction Synchronization Barrier.

Syntax

`ISB {option|#imm}`

Where:

option

Specifies an optional limitation on the barrier operation. Values are:

SY

Full system barrier operation. Can be omitted.

imm

Is an optional 4-bit unsigned immediate, in the range 0 to 15, defaulting to 15.

Usage

Instruction Synchronization Barrier flushes the pipeline in the PE, 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 changing operations executed before the `ISB` instruction are visible to the instructions fetched after the `ISB`. Context changing operations include changing the ASID, TLB maintenance instructions, and all changes to the System registers. In addition, any branches that appear in program order after the `ISB` instruction are written into the branch prediction logic with the context that is visible after the `ISB` instruction. This is needed to ensure correct execution of the instruction stream. For more information, see *Instruction Synchronization Barrier (ISB)* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

16.1 A64 instructions in alphabetical order on page 16-798.

16.85 LSL (register)

Logical Shift Left (register).

This instruction is an alias of `LSLV`.

The equivalent instruction is `LSLV Wd, Wn, Wm`.

Syntax

```
LSL Wd, Wn, Wm ; 32-bit

LSL Xd, Xn, Xm ; 64-bit
```

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

Wn
Is the 32-bit name of the first general-purpose source register.

Wm
Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits.

Xd
Is the 64-bit name of the general-purpose destination register.

Xn
Is the 64-bit name of the first general-purpose source register.

Xm
Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits.

Operation

Logical Shift Left (register) shifts a register value left by a variable number of bits, shifting in zeros, and writes the result to the destination register. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is left-shifted.

`Rd = LSL(Rn, Rm)`, where *R* is either `W` or `X`.

Related references

16.87 LSLV on page 16-896.
16.1 A64 instructions in alphabetical order on page 16-798.

16.86 LSL (immediate)

Logical Shift Left (immediate).

This instruction is an alias of `UBFM`.

The equivalent instruction is `UBFM Wd, Wn, #(-shift MOD 32), #(31-shift)`.

Syntax

```
LSL Wd, Wn, #shift ; 32-bit

LSL Xd, Xn, #shift ; 64-bit
```

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

Wn
Is the 32-bit name of the general-purpose source register.

shift
Depends on the instruction variant:

32-bit general registers
Is the shift amount, in the range 0 to 31.

64-bit general registers
Is the shift amount, in the range 0 to 63.

Xd
Is the 64-bit name of the general-purpose destination register.

Xn
Is the 64-bit name of the general-purpose source register.

Operation

Logical Shift Left (immediate) shifts a register value left by an immediate number of bits, shifting in zeros, and writes the result to the destination register.

`Rd = LSL(Rn, shift)`, where `R` is either `W` or `X`.

Related references

16.164 UBFM on page 16-976.
16.1 A64 instructions in alphabetical order on page 16-798.

16.87 LSLV

Logical Shift Left Variable.

This instruction is used by the alias `LSL` (register).

Syntax

`LSLV` *`Wd`*`,` *`Wn`*`,` *`Wm`* `; 32-bit`

`LSLV` *`Xd`*`,` *`Xn`*`,` *`Xm`* `; 64-bit`

Where:

`Wd`
Is the 32-bit name of the general-purpose destination register.

`Wn`
Is the 32-bit name of the first general-purpose source register.

`Wm`
Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits.

`Xd`
Is the 64-bit name of the general-purpose destination register.

`Xn`
Is the 64-bit name of the first general-purpose source register.

`Xm`
Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits.

Operation

Logical Shift Left Variable shifts a register value left by a variable number of bits, shifting in zeros, and writes the result to the destination register. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is left-shifted.

`Rd = LSL(Rn, Rm)`, where *`R`* is either `W` or `X`.

Related references

16.85 LSL (register) on page 16-894.
16.1 A64 instructions in alphabetical order on page 16-798.

16.88 LSR (register)

Logical Shift Right (register).

This instruction is an alias of `LSRV`.

The equivalent instruction is `LSRV Wd, Wn, Wm`.

Syntax

```
LSR Wd, Wn, Wm ; 32-bit

LSR Xd, Xn, Xm ; 64-bit
```

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

Wn
Is the 32-bit name of the first general-purpose source register.

Wm
Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits.

Xd
Is the 64-bit name of the general-purpose destination register.

Xn
Is the 64-bit name of the first general-purpose source register.

Xm
Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits.

Operation

Logical Shift Right (register) shifts a register value right by a variable number of bits, shifting in zeros, and writes the result to the destination register. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is right-shifted.

`Rd = LSR(Rn, Rm)`, where `R` is either `W` or `X`.

Related references

16.90 LSRV on page 16-899.
16.1 A64 instructions in alphabetical order on page 16-798.

16.89 LSR (immediate)

Logical Shift Right (immediate).

This instruction is an alias of `UBFM`.

The equivalent instruction is `UBFM Wd, Wn, #shift, #31`.

Syntax

```
LSR Wd, Wn, #shift ; 32-bit
LSR Xd, Xn, #shift ; 64-bit
```

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

Wn
Is the 32-bit name of the general-purpose source register.

shift
Depends on the instruction variant:

32-bit general registers
Is the shift amount, in the range 0 to 31.

64-bit general registers
Is the shift amount, in the range 0 to 63.

Xd
Is the 64-bit name of the general-purpose destination register.

Xn
Is the 64-bit name of the general-purpose source register.

Operation

Logical Shift Right (immediate) shifts a register value right by an immediate number of bits, shifting in zeros, and writes the result to the destination register.

`Rd = LSR(Rn, shift)`, where `R` is either `W` or `X`.

Related references

16.164 UBFM on page 16-976.
16.1 A64 instructions in alphabetical order on page 16-798.

16.90 LSRV

Logical Shift Right Variable.

This instruction is used by the alias `LSR` (register).

Syntax

```
LSRV Wd, Wn, Wm ; 32-bit

LSRV Xd, Xn, Xm ; 64-bit
```

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

Wn
Is the 32-bit name of the first general-purpose source register.

Wm
Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits.

Xd
Is the 64-bit name of the general-purpose destination register.

Xn
Is the 64-bit name of the first general-purpose source register.

Xm
Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits.

Operation

Logical Shift Right Variable shifts a register value right by a variable number of bits, shifting in zeros, and writes the result to the destination register. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is right-shifted.

`Rd = LSR(Rn, Rm)`, where `R` is either `W` or `X`.

Related references

16.88 LSR (register) on page 16-897.
16.1 A64 instructions in alphabetical order on page 16-798.

16.91 MADD

Multiply-Add.

This instruction is used by the alias `MUL`.

Syntax

```
MADD Wd, Wn, Wm, Wa ; 32-bit

MADD Xd, Xn, Xm, Xa ; 64-bit
```

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

Wn
Is the 32-bit name of the first general-purpose source register holding the multiplicand.

Wm
Is the 32-bit name of the second general-purpose source register holding the multiplier.

Wa
Is the 32-bit name of the third general-purpose source register holding the addend.

Xd
Is the 64-bit name of the general-purpose destination register.

Xn
Is the 64-bit name of the first general-purpose source register holding the multiplicand.

Xm
Is the 64-bit name of the second general-purpose source register holding the multiplier.

Xa
Is the 64-bit name of the third general-purpose source register holding the addend.

Operation

Multiply-Add multiplies two register values, adds a third register value, and writes the result to the destination register.

`Rd = Ra + Rn * Rm`, where `R` is either `W` or `X`.

Related references

16.106 MUL on page 16-915.
16.1 A64 instructions in alphabetical order on page 16-798.

16.92 MNEG

Multiply-Negate.

This instruction is an alias of `MSUB`.

The equivalent instruction is `MSUB Wd, Wn, Wm, WZR`.

Syntax

```
MNEG Wd, Wn, Wm ; 32-bit

MNEG Xd, Xn, Xm ; 64-bit
```

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

Wn
Is the 32-bit name of the first general-purpose source register holding the multiplicand.

Wm
Is the 32-bit name of the second general-purpose source register holding the multiplier.

Xd
Is the 64-bit name of the general-purpose destination register.

Xn
Is the 64-bit name of the first general-purpose source register holding the multiplicand.

Xm
Is the 64-bit name of the second general-purpose source register holding the multiplier.

Operation

Multiply-Negate multiplies two register values, negates the product, and writes the result to the destination register.

`Rd = -(Rn * Rm)`, where `R` is either `W` or `X`.

Related references

16.105 MSUB on page 16-914.
16.1 A64 instructions in alphabetical order on page 16-798.

16.93 MOV (to or from SP)

Move between register and stack pointer.

This instruction is an alias of ADD (immediate).

The equivalent instruction is `ADD Wd|WSP, Wn|WSP, #0`.

Syntax

```
MOV Wd|WSP, Wn|WSP ; 32-bit
MOV Xd|SP, Xn|SP ; 64-bit
```

Where:

Wd|WSP
Is the 32-bit name of the destination general-purpose register or stack pointer.

Wn|WSP
Is the 32-bit name of the source general-purpose register or stack pointer.

Xd|SP
Is the 64-bit name of the destination general-purpose register or stack pointer.

Xn|SP
Is the 64-bit name of the source general-purpose register or stack pointer.

Operation

`Rd = Rn`, where `R` is either `W` or `X`.

Related references

16.6 ADD (immediate) on page 16-809.
16.1 A64 instructions in alphabetical order on page 16-798.

16.94 MOV (inverted wide immediate)

Move (inverted wide immediate).

This instruction is an alias of `MOVN`.

The equivalent instruction is `MOVN Wd, #imm16, LSL #shift`.

Syntax

```
MOV Wd, #imm ; 32-bit
MOV Xd, #imm ; 64-bit
```

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

imm
Depends on the instruction variant:

32-bit general registers
Is a 32-bit immediate.

64-bit general registers
Is a 64-bit immediate.

Xd
Is the 64-bit name of the general-purpose destination register.

Operation

Move (inverted wide immediate) moves an inverted 16-bit immediate value to a register.

`Rd = imm`, where `R` is either `W` or `X`.

Related references

16.100 MOVN on page 16-909.
16.1 A64 instructions in alphabetical order on page 16-798.

16.95 MOV (wide immediate)

Move (wide immediate).

This instruction is an alias of `MOVZ`.

The equivalent instruction is `MOVZ Wd, #imm16, LSL #shift`.

Syntax

```
MOV Wd, #imm ; 32-bit
MOV Xd, #imm ; 64-bit
```

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

imm
Depends on the instruction variant:

32-bit general registers
Is a 32-bit immediate.

64-bit general registers
Is a 64-bit immediate.

Xd
Is the 64-bit name of the general-purpose destination register.

Operation

Move (wide immediate) moves a 16-bit immediate value to a register.

`Rd = imm`, where `R` is either `W` or `X`.

Related references

16.101 MOVZ on page 16-910.
16.1 A64 instructions in alphabetical order on page 16-798.

16.96 MOV (bitmask immediate)

Move (bitmask immediate).

This instruction is an alias of ORR (immediate).

The equivalent instruction is `ORR Wd|WSP, WZR, #imm`.

Syntax

```
MOV Wd|WSP, #imm ; 32-bit
MOV Xd|SP, #imm ; 64-bit
```

Where:

Wd|WSP
Is the 32-bit name of the destination general-purpose register or stack pointer.

imm
The bitmask immediate but excluding values which could be encoded by MOVZ or MOVN.

Xd|SP
Is the 64-bit name of the destination general-purpose register or stack pointer.

Operation

Move (bitmask immediate) writes a bitmask immediate value to a register.

`Rd = imm`, where *R* is either W or X.

Related references

16.114 ORR (immediate) on page 16-923.
16.1 A64 instructions in alphabetical order on page 16-798.

16.97 MOV (register)

Move (register).

This instruction is an alias of `ORR` (shifted register).

The equivalent instruction is `ORR Wd, WZR, Wm`.

Syntax

```
MOV Wd, Wm ; 32-bit
MOV Xd, Xm ; 64-bit
```

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

Wm
Is the 32-bit name of the general-purpose source register.

Xd
Is the 64-bit name of the general-purpose destination register.

Xm
Is the 64-bit name of the general-purpose source register.

Operation

Move (register) copies the value in a source register to the destination register.

`Rd = Rm`, where *R* is either `W` or `X`.

Related references

16.115 ORR (shifted register) on page 16-924.
16.1 A64 instructions in alphabetical order on page 16-798.

16.98 MOVK

Move wide with keep.

Syntax

```
MOVK Wd, #imm{, LSL #shift} ; 32-bit

MOVK Xd, #imm{, LSL #shift} ; 64-bit
```

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

shift
Depends on the instruction variant:

32-bit general registers
Is the amount by which to shift the immediate left, either 0 (the default) or 16.

64-bit general registers
Is the amount by which to shift the immediate left, either 0 (the default), 16, 32 or 48.

Xd
Is the 64-bit name of the general-purpose destination register.

imm
Is the 16-bit unsigned immediate, in the range 0 to 65535.

Operation

Move wide with keep moves an optionally-shifted 16-bit immediate value into a register, keeping other bits unchanged.

`R`d<shift+15:shift> = imm16, where `R` is either `W` or `X`.

Related references

16.1 A64 instructions in alphabetical order on page 16-798.

16.99 MOVL pseudo-instruction

Load a register with either a 32-bit or 64-bit immediate value or any address.

`MOVL` generates either two or four instructions. If a *Wd* register is specified, `MOVL` generates a `MOV`, `MOVK` pair. If an *Xd* register is specified, `MOVL` generates a `MOV` followed by three `MOVK` instructions. If the assembler can load the register using a single `MOV` instruction, it additionally generates either one or three `NOP`s.

Syntax

`MOVL` *Wd,expr*

`MOVL` *Xd,expr*

where:

Wd
Is the register to load with a 32-bit value.

Xd
Is the register to load with a 64-bit value.

expr
Can be any one of the following:

symbol
A label in this or another program area.

#constant
Any 32-bit or 64-bit immediate value.

symbol + constant
A label plus a 32-bit or 64-bit immediate value.

Usage

Use the `MOVL` pseudo-instruction to:

- Generate literal constants when an immediate value cannot be generated in a single instruction.
- Load a PC-relative or external address into a register. The address remains valid regardless of where the linker places the ELF section containing the `MOVL`.

  Note

  An address loaded in this way is fixed at link time, so the code is not position-independent.

Examples

```
    MOVL w3, #0xABCDEF12   ; loads 0xABCDEF12 into w3
    MOVL x1, Trigger+12    ; loads the address that is 12 bytes higher than
                           ; the address Trigger into x1
```

Related concepts

12.5 Register-relative and PC-relative expressions on page 12-302.

12.10 Numeric local labels on page 12-307.

6.7 Load immediate values using LDR Rd, =const on page 6-110.

Related references

13.77 ORR on page 13-449.

Related information

Armv8-A Architecture Reference Manual.

16.100 MOVN

Move wide with NOT.

This instruction is used by the alias `MOV` (inverted wide immediate).

Syntax

```
MOVN Wd, #imm{, LSL #shift} ; 32-bit

MOVN Xd, #imm{, LSL #shift} ; 64-bit
```

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

shift
Depends on the instruction variant:

32-bit general registers
Is the amount by which to shift the immediate left, either 0 (the default) or 16.

64-bit general registers
Is the amount by which to shift the immediate left, either 0 (the default), 16, 32 or 48.

Xd
Is the 64-bit name of the general-purpose destination register.

imm
Is the 16-bit unsigned immediate, in the range 0 to 65535.

Operation

Move wide with NOT moves the inverse of an optionally-shifted 16-bit immediate value to a register.

`Rd = NOT (LSL (imm16, shift))`, where `R` is either `W` or `X`.

Related references

16.94 MOV (inverted wide immediate) on page 16-903.
16.1 A64 instructions in alphabetical order on page 16-798.

16.101 MOVZ

Move wide with zero.

This instruction is used by the alias `MOV` (wide immediate).

Syntax

```
MOVZ Wd, #imm{, LSL #shift} ; 32-bit

MOVZ Xd, #imm{, LSL #shift} ; 64-bit
```

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

shift
Depends on the instruction variant:

32-bit general registers
Is the amount by which to shift the immediate left, either 0 (the default) or 16.

64-bit general registers
Is the amount by which to shift the immediate left, either 0 (the default), 16, 32 or 48.

Xd
Is the 64-bit name of the general-purpose destination register.

imm
Is the 16-bit unsigned immediate, in the range 0 to 65535.

Operation

Move wide with zero moves an optionally-shifted 16-bit immediate value to a register.

`Rd = LSL (imm16, shift)`, where `R` is either `W` or `X`.

Related references

16.95 MOV (wide immediate) on page 16-904.
16.1 A64 instructions in alphabetical order on page 16-798.

16.102 MRS

Move System Register.

Syntax

```
MRS Xt, (systemreg|Sop0_op1_Cn_Cm_op2)
```

Where:

Xt
Is the 64-bit name of the general-purpose destination register.

systemreg
Is a System register name.

op0
Is an unsigned immediate, and can be either `2` or `3`.

op1
Is a 3-bit unsigned immediate, in the range 0 to 7.

Cn
Is a name `Cn`, with `n` in the range 0 to 15.

Cm
Is a name `Cm`, with `m` in the range 0 to 15.

op2
Is a 3-bit unsigned immediate, in the range 0 to 7.

Usage

Move System Register allows the PE to read an AArch64 System register into a general-purpose register.

Related references

16.1 A64 instructions in alphabetical order on page 16-798.

16.103 MSR (immediate)

Move immediate value to Special Register.

Syntax

```
MSR pstatefield, #imm
```

Where:

pstatefield
Is a PSTATE field name, and can be one of `UAO`, `PAN`, `SPSel`, `DAIFSet` or `DAIFClr`.

imm
Is a 4-bit unsigned immediate, in the range 0 to 15.

Usage

Move immediate value to Special Register moves an immediate value to selected bits of the PSTATE. For more information, see *Process state, PSTATE* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

The bits that can be written are D, A, I, F, and SP. This set of bits is expanded in extensions to the architecture as follows:

- Armv8.1 adds the PAN bit.
- Armv8.2 adds the UAO bit.

Related references

16.1 A64 instructions in alphabetical order on page 16-798.

16.104 MSR (register)

Move general-purpose register to System Register.

Syntax

`MSR (systemreg|Sop0_op1_Cn_Cm_op2), Xt`

Where:

systemreg
Is a System register name.

op0
Is an unsigned immediate, and can be either `2` or `3`.

op1
Is a 3-bit unsigned immediate, in the range 0 to 7.

Cn
Is a name `Cn`, with `n` in the range 0 to 15.

Cm
Is a name `Cm`, with `m` in the range 0 to 15.

op2
Is a 3-bit unsigned immediate, in the range 0 to 7.

Xt
Is the 64-bit name of the general-purpose source register.

Usage

Move general-purpose register to System Register allows the PE to write an AArch64 System register from a general-purpose register.

Related references

16.1 A64 instructions in alphabetical order on page 16-798.

16.105 MSUB

Multiply-Subtract.

This instruction is used by the alias `MNEG`.

Syntax

```
MSUB Wd, Wn, Wm, Wa ; 32-bit

MSUB Xd, Xn, Xm, Xa ; 64-bit
```

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

Wn
Is the 32-bit name of the first general-purpose source register holding the multiplicand.

Wm
Is the 32-bit name of the second general-purpose source register holding the multiplier.

Wa
Is the 32-bit name of the third general-purpose source register holding the minuend.

Xd
Is the 64-bit name of the general-purpose destination register.

Xn
Is the 64-bit name of the first general-purpose source register holding the multiplicand.

Xm
Is the 64-bit name of the second general-purpose source register holding the multiplier.

Xa
Is the 64-bit name of the third general-purpose source register holding the minuend.

Operation

Multiply-Subtract multiplies two register values, subtracts the product from a third register value, and writes the result to the destination register.

`Rd = Ra - Rn * Rm`, where `R` is either `W` or `X`.

Related references

16.92 MNEG on page 16-901.
16.1 A64 instructions in alphabetical order on page 16-798.

16.106 MUL

Multiply.

This instruction is an alias of `MADD`.

The equivalent instruction is `MADD Wd, Wn, Wm, WZR`.

Syntax

```
MUL Wd, Wn, Wm ; 32-bit
MUL Xd, Xn, Xm ; 64-bit
```

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

Wn
Is the 32-bit name of the first general-purpose source register holding the multiplicand.

Wm
Is the 32-bit name of the second general-purpose source register holding the multiplier.

Xd
Is the 64-bit name of the general-purpose destination register.

Xn
Is the 64-bit name of the first general-purpose source register holding the multiplicand.

Xm
Is the 64-bit name of the second general-purpose source register holding the multiplier.

Operation

`Rd = Rn * Rm`, where `R` is either `W` or `X`.

Related references

16.91 MADD on page 16-900.
16.1 A64 instructions in alphabetical order on page 16-798.

16.107 MVN

Bitwise NOT.

This instruction is an alias of `ORN` (shifted register).

The equivalent instruction is `ORN Wd, WZR, Wm{, shift #amount}`.

Syntax

```
MVN Wd, Wm{, shift #amount} ; 32-bit
MVN Xd, Xm{, shift #amount} ; 64-bit
```

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

Wm
Is the 32-bit name of the general-purpose source register.

amount
Depends on the instruction variant:

32-bit general registers
Is the shift amount, in the range 0 to 31, defaulting to 0.

64-bit general registers
Is the shift amount, in the range 0 to 63, defaulting to 0.

Xd
Is the 64-bit name of the general-purpose destination register.

Xm
Is the 64-bit name of the general-purpose source register.

shift
Is the optional shift to be applied to the final source, defaulting to LSL, and can be one of `LSL`, `LSR`, `ASR`, or `ROR`.

Operation

Bitwise NOT writes the bitwise inverse of a register value to the destination register.

`Rd = NOT shift(Rm, amount)`, where `R` is either `W` or `X`.

Related references

16.113 ORN (shifted register) on page 16-922.
16.1 A64 instructions in alphabetical order on page 16-798.

16.108 NEG (shifted register)

Negate (shifted register).

This instruction is an alias of SUB (shifted register).

The equivalent instruction is `SUB Wd, WZR, Wm {, shift #amount}`.

Syntax

```
NEG Wd, Wm{, shift #amount} ; 32-bit

NEG Xd, Xm{, shift #amount} ; 64-bit
```

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

Wm
Is the 32-bit name of the general-purpose source register.

amount
Depends on the instruction variant:

32-bit general registers
Is the shift amount, in the range 0 to 31, defaulting to 0.

64-bit general registers
Is the shift amount, in the range 0 to 63, defaulting to 0.

Xd
Is the 64-bit name of the general-purpose destination register.

Xm
Is the 64-bit name of the general-purpose source register.

shift
Is the optional shift type to be applied to the second source operand, defaulting to LSL, and can be one of LSL, LSR, or ASR.

Operation

Negate (shifted register) negates an optionally-shifted register value, and writes the result to the destination register.

`Rd = 0 - shift(Rm, amount)`, where `R` is either `W` or `X`.

Related references

16.148 SUB (shifted register) on page 16-958.
16.1 A64 instructions in alphabetical order on page 16-798.

16.109 NEGS

Negate, setting flags.

This instruction is an alias of SUBS (shifted register).

The equivalent instruction is `SUBS Wd, WZR, Wm {, shift #amount}`.

Syntax

```
NEGS Wd, Wm{, shift #amount} ; 32-bit

NEGS Xd, Xm{, shift #amount} ; 64-bit
```

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

Wm
Is the 32-bit name of the general-purpose source register.

amount
Depends on the instruction variant:

32-bit general registers
Is the shift amount, in the range 0 to 31, defaulting to 0.

64-bit general registers
Is the shift amount, in the range 0 to 63, defaulting to 0.

Xd
Is the 64-bit name of the general-purpose destination register.

Xm
Is the 64-bit name of the general-purpose source register.

shift
Is the optional shift type to be applied to the second source operand, defaulting to LSL, and can be one of LSL, LSR, or ASR.

Operation

Negate, setting flags, negates an optionally-shifted register value, and writes the result to the destination register. It updates the condition flags based on the result.

`Rd = 0 - shift(Rm, amount)`, where *R* is either W or X.

Related references

16.151 SUBS (shifted register) on page 16-962.
16.1 A64 instructions in alphabetical order on page 16-798.

16.110 NGC

Negate with Carry.

This instruction is an alias of `SBC`.

The equivalent instruction is `SBC Wd, WZR, Wm`.

Syntax

```
NGC Wd, Wm ; 32-bit
NGC Xd, Xm ; 64-bit
```

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

Wm
Is the 32-bit name of the general-purpose source register.

Xd
Is the 64-bit name of the general-purpose destination register.

Xm
Is the 64-bit name of the general-purpose source register.

Operation

Negate with Carry negates the sum of a register value and the value of NOT (Carry flag), and writes the result to the destination register.

`Rd = 0 - Rm - 1 + C`, where *R* is either `W` or `X`.

Related references

16.132 SBC on page 16-941.
16.1 A64 instructions in alphabetical order on page 16-798.

16.111 NGCS

Negate with Carry, setting flags.

This instruction is an alias of `SBCS`.

The equivalent instruction is `SBCS Wd, WZR, Wm`.

Syntax

```
NGCS Wd, Wm ; 32-bit

NGCS Xd, Xm ; 64-bit
```

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

Wm
Is the 32-bit name of the general-purpose source register.

Xd
Is the 64-bit name of the general-purpose destination register.

Xm
Is the 64-bit name of the general-purpose source register.

Operation

Negate with Carry, setting flags, negates the sum of a register value and the value of NOT (Carry flag), and writes the result to the destination register. It updates the condition flags based on the result.

`Rd = 0 - Rm - 1 + C`, where *R* is either `W` or `X`.

Related references

16.133 SBCS on page 16-942.
16.1 A64 instructions in alphabetical order on page 16-798.

16.112 NOP

No Operation.

Usage

No Operation does nothing, other than advance the value of the program counter by 4. This instruction can be used for instruction alignment purposes.

Note

The timing effects of including a `NOP` instruction in a program are not guaranteed. It can increase execution time, leave it unchanged, or even reduce it. Therefore, `NOP` instructions are not suitable for timing loops.

Related references

16.1 A64 instructions in alphabetical order on page 16-798.

16.113 ORN (shifted register)

Bitwise OR NOT (shifted register).

This instruction is used by the alias `MVN`.

Syntax

```
ORN Wd, Wn, Wm{, shift #amount} ; 32-bit
ORN Xd, Xn, Xm{, shift #amount} ; 64-bit
```

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

Wn
Is the 32-bit name of the first general-purpose source register.

Wm
Is the 32-bit name of the second general-purpose source register.

amount
Depends on the instruction variant:

32-bit general registers
Is the shift amount, in the range 0 to 31, defaulting to 0.

64-bit general registers
Is the shift amount, in the range 0 to 63, defaulting to 0.

Xd
Is the 64-bit name of the general-purpose destination register.

Xn
Is the 64-bit name of the first general-purpose source register.

Xm
Is the 64-bit name of the second general-purpose source register.

shift
Is the optional shift to be applied to the final source, defaulting to LSL, and can be one of `LSL`, `LSR`, `ASR`, or `ROR`.

Operation

Bitwise OR NOT (shifted register) performs a bitwise (inclusive) OR of a register value and the complement of an optionally-shifted register value, and writes the result to the destination register.

`Rd = Rn OR NOT shift(Rm, amount)`, where `R` is either `W` or `X`.

Related references

16.107 MVN on page 16-916.
16.1 A64 instructions in alphabetical order on page 16-798.

16.114 ORR (immediate)

Bitwise OR (immediate).

This instruction is used by the alias `MOV` (bitmask immediate).

Syntax

```
ORR Wd|WSP, Wn, #imm ; 32-bit

ORR Xd|SP, Xn, #imm ; 64-bit
```

Where:

Wd|WSP
Is the 32-bit name of the destination general-purpose register or stack pointer.

Wn
Is the 32-bit name of the general-purpose source register.

imm
The bitmask immediate.

Xd|SP
Is the 64-bit name of the destination general-purpose register or stack pointer.

Xn
Is the 64-bit name of the general-purpose source register.

Operation

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

`Rd = Rn OR imm`, where *R* is either `W` or `X`.

Related references

16.96 MOV (bitmask immediate) on page 16-905.
16.1 A64 instructions in alphabetical order on page 16-798.

16.115 ORR (shifted register)

Bitwise OR (shifted register).

This instruction is used by the alias `MOV` (register).

Syntax

```
ORR Wd, Wn, Wm{, shift #amount} ; 32-bit
```

```
ORR Xd, Xn, Xm{, shift #amount} ; 64-bit
```

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

Wn
Is the 32-bit name of the first general-purpose source register.

Wm
Is the 32-bit name of the second general-purpose source register.

amount
Depends on the instruction variant:

32-bit general registers
Is the shift amount, in the range 0 to 31, defaulting to 0.

64-bit general registers
Is the shift amount, in the range 0 to 63, defaulting to 0.

Xd
Is the 64-bit name of the general-purpose destination register.

Xn
Is the 64-bit name of the first general-purpose source register.

Xm
Is the 64-bit name of the second general-purpose source register.

shift
Is the optional shift to be applied to the final source, defaulting to LSL, and can be one of `LSL`, `LSR`, `ASR`, or `ROR`.

Operation

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

`Rd = Rn OR shift(Rm, amount)`, where `R` is either `W` or `X`.

Related references

16.97 MOV (register) on page 16-906.
16.1 A64 instructions in alphabetical order on page 16-798.

16.116 PACDA, PACDZA

Pointer Authentication Code for Data address, using key A.

Syntax

```
PACDA Xd, Xn|SP ; PACDA general registers

PACDZA Xd ; PACDZA general registers
```

Where:

Xn|SP
Is the 64-bit name of the general-purpose source register or stack pointer.

Xd
Is the 64-bit name of the general-purpose destination register.

Architectures supported

Supported in the Armv8.3-A architecture and later.

Usage

Pointer Authentication Code for Data address, using key A. This instruction computes and inserts a pointer authentication code for a data address, using a modifier and key A.

The address is in the general-purpose register that is specified by *Xd*.

The modifier is:

- In the general-purpose register or stack pointer that is specified by *Xn|SP* for PACDA.
- The value zero, for PACDZA.

Related references

16.1 A64 instructions in alphabetical order on page 16-798.

16.117 PACDB, PACDZB

Pointer Authentication Code for Data address, using key B.

Syntax

```
PACDB Xd, Xn|SP ; PACDB general registers

PACDZB Xd ; PACDZB general registers
```

Where:

Xn|SP
Is the 64-bit name of the general-purpose source register or stack pointer.

Xd
Is the 64-bit name of the general-purpose destination register.

Architectures supported

Supported in the Armv8.3-A architecture and later.

Usage

Pointer Authentication Code for Data address, using key B. This instruction computes and inserts a pointer authentication code for a data address, using a modifier and key B.

The address is in the general-purpose register that is specified by *Xd*.

The modifier is:

- In the general-purpose register or stack pointer that is specified by *Xn|SP* for PACDB.
- The value zero, for PACDZB.

Related references

16.1 A64 instructions in alphabetical order on page 16-798.

16.118 PACGA

Pointer Authentication Code, using Generic key.

Syntax

`PACGA Xd, Xn, Xm|SP`

Where:

Xd
Is the 64-bit name of the general-purpose destination register.

Xn
Is the 64-bit name of the first general-purpose source register.

Xm|SP
Is the 64-bit name of the second general-purpose source register or stack pointer.

Architectures supported

Supported in the Armv8.3-A architecture and later.

Usage

Pointer Authentication Code, using Generic key. This instruction computes the pointer authentication code for an address in the first source register, using a modifier in the second source register, and the Generic key. The computed pointer authentication code is returned in the upper 32 bits of the destination register.

Related references

16.1 A64 instructions in alphabetical order on page 16-798.

16.119 PACIA, PACIZA, PACIA1716, PACIASP, PACIAZ

Pointer Authentication Code for Instruction address, using key A.

Syntax

```
PACIA Xd, Xn|SP ; PACIA general registers

PACIZA Xd ; PACIZA general registers

PACIA1716

PACIASP

PACIAZ
```

Where:

Xd
Is the 64-bit name of the general-purpose destination register.

Xn|SP
Is the 64-bit name of the general-purpose source register or stack pointer.

Architectures supported

Supported in the Armv8.3-A architecture and later.

Usage

Pointer Authentication Code for Instruction address, using key A. This instruction computes and inserts a pointer authentication code for an instruction address, using a modifier and key A.

The address is:

- In the general-purpose register that is specified by *Xd* for `PACIA` and `PACIZA`.
- In X17, for `PACIA1716`.
- In X30, for `PACIASP` and `PACIAZ`.

The modifier is:

- In the general-purpose register or stack pointer that is specified by *Xn|SP* for `PACIA`.
- The value zero, for `PACIZA` and `PACIAZ`.
- In X16, for `PACIA1716`.
- In SP, for `PACIASP`.

Related references

16.1 A64 instructions in alphabetical order on page 16-798.

16.120 PACIB, PACIZB, PACIB1716, PACIBSP, PACIBZ

Pointer Authentication Code for Instruction address, using key B.

Syntax

```
PACIB Xd, Xn|SP ; PACIB general registers

PACIZB Xd ; PACIZB general registers

PACIB1716

PACIBSP

PACIBZ
```

Where:

Xd
Is the 64-bit name of the general-purpose destination register.

Xn|SP
Is the 64-bit name of the general-purpose source register or stack pointer.

Architectures supported

Supported in the Armv8.3-A architecture and later.

Usage

Pointer Authentication Code for Instruction address, using key B. This instruction computes and inserts a pointer authentication code for an instruction address, using a modifier and key B.

The address is:

- In the general-purpose register that is specified by *Xd* for `PACIB` and `PACIZB`.
- In X17, for `PACIB1716`.
- In X30, for `PACIBSP` and `PACIBZ`.

The modifier is:

- In the general-purpose register or stack pointer that is specified by *Xn|SP* for `PACIB`.
- The value zero, for `PACIZB` and `PACIBZ`.
- In X16, for `PACIB1716`.
- In SP, for `PACIBSP`.

Related references

16.1 A64 instructions in alphabetical order on page 16-798.

16.121 PSB

Profiling Synchronization Barrier.

Syntax

```
PSB CSYNC
```

Architectures supported

Supported in the Armv8.2 architecture and later.

Usage

Profiling Synchronization Barrier. This instruction is a barrier that ensures that all existing profiling data for the current PE has been formatted, and profiling buffer addresses have been translated such that all writes to the profiling buffer have been initiated. A following `DSB` instruction completes when the writes to the profiling buffer have completed.

Related references

16.1 A64 instructions in alphabetical order on page 16-798.

16.122 RBIT

Reverse Bits.

Syntax

```
RBIT Wd, Wn ; 32-bit

RBIT Xd, Xn ; 64-bit
```

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

Wn
Is the 32-bit name of the general-purpose source register.

Xd
Is the 64-bit name of the general-purpose destination register.

Xn
Is the 64-bit name of the general-purpose source register.

Usage

Reverse Bits reverses the bit order in a register.

Related references

16.1 A64 instructions in alphabetical order on page 16-798.

16.123 RET

Return from subroutine.

Syntax

`RET {Xn}`

Where:

Xn

Is the 64-bit name of the general-purpose register holding the address to be branched to. Defaults to X30 if absent.

Usage

Return from subroutine branches unconditionally to an address in a register, with a hint that this is a subroutine return.

Related references

16.1 A64 instructions in alphabetical order on page 16-798.

16.124 RETAA, RETAB

Return from subroutine, with pointer authentication.

Syntax

`RETAA`

`RETAB`

Architectures supported

Supported in the Armv8.3-A architecture and later.

Usage

Return from subroutine, with pointer authentication. This instruction authenticates the address that is held in LR, using SP as the modifier and the specified key, branches to the authenticated address, with a hint that this instruction is a subroutine return.

Key A is used for `RETAA`, and key B is used for `RETAB`.

If the authentication passes, the PE continues execution at the target of the branch. If the authentication fails, a Translation fault is generated.

The authenticated address is not written back to LR.

Related references

16.1 A64 instructions in alphabetical order on page 16-798.

16.125 REV16

Reverse bytes in 16-bit halfwords.

Syntax

```
REV16 Wd, Wn ; 32-bit

REV16 Xd, Xn ; 64-bit
```

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

Wn
Is the 32-bit name of the general-purpose source register.

Xd
Is the 64-bit name of the general-purpose destination register.

Xn
Is the 64-bit name of the general-purpose source register.

Usage

Reverse bytes in 16-bit halfwords reverses the byte order in each 16-bit halfword of a register.

Related references

16.1 A64 instructions in alphabetical order on page 16-798.

16.126 REV32

Reverse bytes in 32-bit words.

Syntax

`REV32 Xd, Xn`

Where:

Xd
Is the 64-bit name of the general-purpose destination register.

Xn
Is the 64-bit name of the general-purpose source register.

Usage

Reverse bytes in 32-bit words reverses the byte order in each 32-bit word of a register.

Related references

16.1 A64 instructions in alphabetical order on page 16-798.

16.127 REV64

Reverse Bytes.

This instruction is an alias of REV.

The equivalent instruction is `REV Xd, Xn`.

Syntax

`REV64 Xd, Xn`

Where:

Xd
Is the 64-bit name of the general-purpose destination register.

Xn
Is the 64-bit name of the general-purpose source register.

Usage

Reverse Bytes reverses the byte order in a 64-bit general-purpose register.

When assembling for Armv8.2, an assembler must support this pseudo-instruction. It is OPTIONAL whether an assembler supports this pseudo-instruction when assembling for an architecture earlier than Armv8.2.

Related references

16.128 REV on page 16-937.
16.1 A64 instructions in alphabetical order on page 16-798.

16.128 REV

Reverse Bytes.

This instruction is used by the alias `REV64`.

Syntax

```
REV Wd, Wn ; 32-bit

REV Xd, Xn ; 64-bit
```

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

Wn
Is the 32-bit name of the general-purpose source register.

Xd
Is the 64-bit name of the general-purpose destination register.

Xn
Is the 64-bit name of the general-purpose source register.

Usage

Reverse Bytes reverses the byte order in a register.

Related references

16.127 REV64 on page 16-936.
16.1 A64 instructions in alphabetical order on page 16-798.

16.129 ROR (immediate)

Rotate right (immediate).

This instruction is an alias of EXTR.

The equivalent instruction is `EXTR Wd, Ws, Ws, #shift`.

Syntax

```
ROR Wd, Ws, #shift ; 32-bit

ROR Xd, Xs, #shift ; 64-bit
```

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

Ws
Is the 32-bit name of the general-purpose source register.

shift
Depends on the instruction variant:

32-bit general registers
Is the amount by which to rotate, in the range 0 to 31.

64-bit general registers
Is the amount by which to rotate, in the range 0 to 63.

Xd
Is the 64-bit name of the general-purpose destination register.

Xs
Is the 64-bit name of the general-purpose source register.

Operation

Rotate right (immediate) provides the value of the contents of a register rotated by a variable number of bits. The bits that are rotated off the right end are inserted into the vacated bit positions on the left.

`Rd = ROR(Rs, shift)`, where `R` is either `W` or `X`.

Related references

16.79 EXTR on page 16-888.
16.1 A64 instructions in alphabetical order on page 16-798.

16.130 ROR (register)

Rotate Right (register).

This instruction is an alias of `RORV`.

The equivalent instruction is `RORV Wd, Wn, Wm`.

Syntax

```
ROR Wd, Wn, Wm ; 32-bit

ROR Xd, Xn, Xm ; 64-bit
```

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

Wn
Is the 32-bit name of the first general-purpose source register.

Wm
Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits.

Xd
Is the 64-bit name of the general-purpose destination register.

Xn
Is the 64-bit name of the first general-purpose source register.

Xm
Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits.

Operation

Rotate Right (register) provides the value of the contents of a register rotated by a variable number of bits. The bits that are rotated off the right end are inserted into the vacated bit positions on the left. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is right-shifted.

`Rd = ROR(Rn, Rm)`, where *R* is either `W` or `X`.

Related references

16.131 RORV on page 16-940.
16.1 A64 instructions in alphabetical order on page 16-798.

16.131 RORV

Rotate Right Variable.

This instruction is used by the alias `ROR` (register).

Syntax

```
RORV Wd, Wn, Wm ; 32-bit

RORV Xd, Xn, Xm ; 64-bit
```

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

Wn
Is the 32-bit name of the first general-purpose source register.

Wm
Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits.

Xd
Is the 64-bit name of the general-purpose destination register.

Xn
Is the 64-bit name of the first general-purpose source register.

Xm
Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits.

Operation

Rotate Right Variable provides the value of the contents of a register rotated by a variable number of bits. The bits that are rotated off the right end are inserted into the vacated bit positions on the left. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is right-shifted.

`Rd = ROR(Rn, Rm)`, where `R` is either `W` or `X`.

Related references

16.130 ROR (register) on page 16-939.
16.1 A64 instructions in alphabetical order on page 16-798.

16.132 SBC

Subtract with Carry.

This instruction is used by the alias `NGC`.

Syntax

`SBC Wd, Wn, Wm ; 32-bit`

`SBC Xd, Xn, Xm ; 64-bit`

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

Wn
Is the 32-bit name of the first general-purpose source register.

Wm
Is the 32-bit name of the second general-purpose source register.

Xd
Is the 64-bit name of the general-purpose destination register.

Xn
Is the 64-bit name of the first general-purpose source register.

Xm
Is the 64-bit name of the second general-purpose source register.

Operation

Subtract with Carry subtracts a register value and the value of NOT (Carry flag) from a register value, and writes the result to the destination register.

`Rd = Rn - Rm - 1 + C`, where *R* is either `W` or `X`.

Related references

16.110 NGC on page 16-919.
16.1 A64 instructions in alphabetical order on page 16-798.

16.133 SBCS

Subtract with Carry, setting flags.

This instruction is used by the alias `NGCS`.

Syntax

```
SBCS Wd, Wn, Wm ; 32-bit

SBCS Xd, Xn, Xm ; 64-bit
```

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

Wn
Is the 32-bit name of the first general-purpose source register.

Wm
Is the 32-bit name of the second general-purpose source register.

Xd
Is the 64-bit name of the general-purpose destination register.

Xn
Is the 64-bit name of the first general-purpose source register.

Xm
Is the 64-bit name of the second general-purpose source register.

Operation

Subtract with Carry, setting flags, subtracts a register value and the value of NOT (Carry flag) from a register value, and writes the result to the destination register. It updates the condition flags based on the result.

`Rd = Rn - Rm - 1 + C`, where *R* is either `W` or `X`.

Related references

16.111 NGCS on page 16-920.
16.1 A64 instructions in alphabetical order on page 16-798.

16.134 SBFIZ

Signed Bitfield Insert in Zero.

This instruction is an alias of `SBFM`.

The equivalent instruction is `SBFM Wd, Wn, #(-lsb MOD 32), #(width-1)`.

Syntax

```
SBFIZ Wd, Wn, #lsb, #width ; 32-bit

SBFIZ Xd, Xn, #lsb, #width ; 64-bit
```

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

Wn
Is the 32-bit name of the general-purpose source register.

lsb
Depends on the instruction variant:

32-bit general registers
Is the bit number of the lsb of the destination bitfield, in the range 0 to 31.

64-bit general registers
Is the bit number of the lsb of the destination bitfield, in the range 0 to 63.

width
Depends on the instruction variant:

32-bit general registers
Is the width of the bitfield, in the range 1 to 32-*lsb*.

64-bit general registers
Is the width of the bitfield, in the range 1 to 64-*lsb*.

Xd
Is the 64-bit name of the general-purpose destination register.

Xn
Is the 64-bit name of the general-purpose source register.

Usage

Signed Bitfield Insert in Zero zeros the destination register and copies any number of contiguous bits from a source register into any position in the destination register, sign-extending the most significant bit of the transferred value.

Related references

16.135 SBFM on page 16-944.
16.1 A64 instructions in alphabetical order on page 16-798.

16.135 SBFM

Signed Bitfield Move.

This instruction is used by the aliases:

- `ASR` (immediate).
- `SBFIZ`.
- `SBFX`.
- `SXTB`.
- `SXTH`.
- `SXTW`.

Syntax

```
SBFM Wd, Wn, #<immr>, #<imms> ; 32-bit

SBFM Xd, Xn, #<immr>, #<imms> ; 64-bit
```

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

Wn
Is the 32-bit name of the general-purpose source register.

`<immr>`
Depends on the instruction variant:

32-bit general registers
Is the right rotate amount, in the range 0 to 31.

64-bit general registers
Is the right rotate amount, in the range 0 to 63.

`<imms>`
Depends on the instruction variant:

32-bit general registers
Is the leftmost bit number to be moved from the source, in the range 0 to 31.

64-bit general registers
Is the leftmost bit number to be moved from the source, in the range 0 to 63.

Xd
Is the 64-bit name of the general-purpose destination register.

Xn
Is the 64-bit name of the general-purpose source register.

Usage

Signed Bitfield Move copies any number of low-order bits from a source register into the same number of adjacent bits at any position in the destination register, shifting in copies of the sign bit in the upper bits and zeros in the lower bits.

Related references

16.19 ASR (immediate) on page 16-823.
16.134 SBFIZ on page 16-943.
16.136 SBFX on page 16-945.
16.153 SXTB on page 16-964.
16.154 SXTH on page 16-965.
16.155 SXTW on page 16-966.
16.1 A64 instructions in alphabetical order on page 16-798.

16.136 SBFX

Signed Bitfield Extract.

This instruction is an alias of `SBFM`.

The equivalent instruction is `SBFM Wd, Wn, #lsb, #(lsb+width-1)`.

Syntax

```
SBFX Wd, Wn, #lsb, #width ; 32-bit

SBFX Xd, Xn, #lsb, #width ; 64-bit
```

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

Wn
Is the 32-bit name of the general-purpose source register.

lsb
Depends on the instruction variant:

32-bit general registers
Is the bit number of the lsb of the source bitfield, in the range 0 to 31.

64-bit general registers
Is the bit number of the lsb of the source bitfield, in the range 0 to 63.

width
Depends on the instruction variant:

32-bit general registers
Is the width of the bitfield, in the range 1 to 32-*lsb*.

64-bit general registers
Is the width of the bitfield, in the range 1 to 64-*lsb*.

Xd
Is the 64-bit name of the general-purpose destination register.

Xn
Is the 64-bit name of the general-purpose source register.

Usage

Signed Bitfield Extract extracts any number of adjacent bits at any position from a register, sign-extends them to the size of the register, and writes the result to the destination register.

Related references

16.135 SBFM on page 16-944.
16.1 A64 instructions in alphabetical order on page 16-798.

16.137 SDIV

Signed Divide.

Syntax

```
SDIV Wd, Wn, Wm ; 32-bit

SDIV Xd, Xn, Xm ; 64-bit
```

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

Wn
Is the 32-bit name of the first general-purpose source register.

Wm
Is the 32-bit name of the second general-purpose source register.

Xd
Is the 64-bit name of the general-purpose destination register.

Xn
Is the 64-bit name of the first general-purpose source register.

Xm
Is the 64-bit name of the second general-purpose source register.

Operation

Signed Divide divides a signed integer register value by another signed integer register value, and writes the result to the destination register. The condition flags are not affected.

`Rd = Rn / Rm`, where `R` is either `W` or `X`.

Related references

16.1 A64 instructions in alphabetical order on page 16-798.

16.138 SEV

Send Event.

Usage

Send Event is a hint instruction. It causes an event to be signaled to all PEs in the multiprocessor system. For more information, see *Wait for Event mechanism and Send event* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

16.1 A64 instructions in alphabetical order on page 16-798.

16.139 SEVL

Send Event Local.

Usage

Send Event Local is a hint instruction. It causes an event to be signaled locally without the requirement to affect other PEs in the multiprocessor system. It can prime a wait-loop which starts with a `WFE` instruction.

Related references

16.1 A64 instructions in alphabetical order on page 16-798.

16.140 SMADDL

Signed Multiply-Add Long.

This instruction is used by the alias `SMULL`.

Syntax

`SMADDL Xd, Wn, Wm, Xa`

Where:

Xd
Is the 64-bit name of the general-purpose destination register.

Wn
Is the 32-bit name of the first general-purpose source register holding the multiplicand.

Wm
Is the 32-bit name of the second general-purpose source register holding the multiplier.

Xa
Is the 64-bit name of the third general-purpose source register holding the addend.

Operation

Signed Multiply-Add Long multiplies two 32-bit register values, adds a 64-bit register value, and writes the result to the 64-bit destination register.

```
Xd = Xa + Wn * Wm.
```

Related references

16.145 SMULL on page 16-954.
16.1 A64 instructions in alphabetical order on page 16-798.

16.141 SMC

Supervisor call to allow OS or Hypervisor code to call the Secure Monitor. It generates an exception targeting exception level 3 (EL3).

Syntax

`SMC #imm`

Where:

imm

Is a 16-bit unsigned immediate, in the range 0 to 65535. This value is made available to the handler in the Exception Syndrome Register.

Usage

Secure Monitor Call causes an exception to EL3.

`SMC` is available only for software executing at EL1 or higher. It is UNDEFINED in EL0.

If the values of HCR_EL2.TSC and SCR_EL3.SMD are both 0, execution of an `SMC` instruction at EL1 or higher generates a Secure Monitor Call exception, recording it in ESR_ELx, using the EC value `0x17`, that is taken to EL3.

If the value of HCR_EL2.TSC is 1, execution of an `SMC` instruction in a Non-secure EL1 state generates an exception that is taken to EL2, regardless of the value of SCR_EL3.SMD. For more information, see *Traps to EL2 of Non-secure EL1 execution of SMC instructions* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

If the value of HCR_EL2.TSC is 0 and the value of SCR_EL3.SMD is 1, the SMC instruction is UNDEFINED.

Related references

16.1 A64 instructions in alphabetical order on page 16-798.

16.142 SMNEGL

Signed Multiply-Negate Long.

This instruction is an alias of `SMSUBL`.

The equivalent instruction is `SMSUBL Xd, Wn, Wm, XZR`.

Syntax

`SMNEGL Xd, Wn, Wm`

Where:

Xd
Is the 64-bit name of the general-purpose destination register.

Wn
Is the 32-bit name of the first general-purpose source register holding the multiplicand.

Wm
Is the 32-bit name of the second general-purpose source register holding the multiplier.

Operation

Signed Multiply-Negate Long multiplies two 32-bit register values, negates the product, and writes the result to the 64-bit destination register.

```
Xd = -(Wn * Wm).
```

Related references

16.143 SMSUBL on page 16-952.
16.1 A64 instructions in alphabetical order on page 16-798.

16.143 SMSUBL

Signed Multiply-Subtract Long.

This instruction is used by the alias `SMNEGL`.

Syntax

`SMSUBL` *`Xd, Wn, Wm, Xa`*

Where:

Xd
Is the 64-bit name of the general-purpose destination register.

Wn
Is the 32-bit name of the first general-purpose source register holding the multiplicand.

Wm
Is the 32-bit name of the second general-purpose source register holding the multiplier.

Xa
Is the 64-bit name of the third general-purpose source register holding the minuend.

Operation

Signed Multiply-Subtract Long multiplies two 32-bit register values, subtracts the product from a 64-bit register value, and writes the result to the 64-bit destination register.

```
Xd = Xa - Wn * Wm.
```

Related references

16.142 SMNEGL on page 16-951.
16.1 A64 instructions in alphabetical order on page 16-798.

16.144 SMULH

Signed Multiply High.

Syntax

`SMULH Xd, Xn, Xm`

Where:

Xd
Is the 64-bit name of the general-purpose destination register.

Xn
Is the 64-bit name of the first general-purpose source register holding the multiplicand.

Xm
Is the 64-bit name of the second general-purpose source register holding the multiplier.

Operation

Signed Multiply High multiplies two 64-bit register values, and writes bits[127:64] of the 128-bit result to the 64-bit destination register.

```
Xd = bits<127:64> of Xn * Xm.
```

Related references

16.1 A64 instructions in alphabetical order on page 16-798.

16.145 SMULL

Signed Multiply Long.

This instruction is an alias of `SMADDL`.

The equivalent instruction is `SMADDL Xd, Wn, Wm, XZR`.

Syntax

`SMULL Xd, Wn, Wm`

Where:

Xd
Is the 64-bit name of the general-purpose destination register.

Wn
Is the 32-bit name of the first general-purpose source register holding the multiplicand.

Wm
Is the 32-bit name of the second general-purpose source register holding the multiplier.

Operation

Signed Multiply Long multiplies two 32-bit register values, and writes the result to the 64-bit destination register.

```
Xd = Wn * Wm.
```

Related references

16.140 SMADDL on page 16-949.
16.1 A64 instructions in alphabetical order on page 16-798.

16.146 SUB (extended register)

Subtract (extended register).

Syntax

`SUB Wd|WSP, Wn|WSP, Wm{, extend {#amount}} ; 32-bit`

`SUB Xd|SP, Xn|SP, Rm{, extend {#amount}} ; 64-bit`

Where:

Wd|WSP
Is the 32-bit name of the destination general-purpose register or stack pointer.

Wn|WSP
Is the 32-bit name of the first source general-purpose register or stack pointer.

Wm
Is the 32-bit name of the second general-purpose source register.

extend
Is the extension to be applied to the second source operand:

32-bit general registers
Can be one of `UXTB`, `UXTH`, `LSL|UXTW`, `UXTX`, `SXTB`, `SXTH`, `SXTW` or `SXTX`.

If *Rd* or *Rn* is WSP then `LSL` is preferred rather than `UXTW`, and can be omitted when *amount* is 0. In all other cases *extend* is required and must be `UXTW` rather than `LSL`.

64-bit general registers
Can be one of `UXTB`, `UXTH`, `UXTW`, `LSL|UXTX`, `SXTB`, `SXTH`, `SXTW` or `SXTX`.

If *Rd* or *Rn* is SP then `LSL` is preferred rather than `UXTX`, and can be omitted when *amount* is 0. In all other cases *extend* is required and must be `UXTX` rather than `LSL`.

Xd|SP
Is the 64-bit name of the destination general-purpose register or stack pointer.

Xn|SP
Is the 64-bit name of the first source general-purpose register or stack pointer.

R
Is a width specifier, and can be either `W` or `X`.

m
Is the number [0-30] of the second general-purpose source register or the name ZR (31).

amount
Is the left shift amount to be applied after extension in the range 0 to 4, defaulting to 0. It must be absent when *extend* is absent, is required when *extend* is LSL, and is optional when *extend* is present but not LSL.

Operation

Subtract (extended register) subtracts a sign or zero-extended register value, followed by an optional left shift amount, from a register value, and writes the result to the destination register. The argument that is extended from the *Rm* register can be a byte, halfword, word, or doubleword.

`Rd = Rn - LSL(extend(Rm), amount)`, where *R* is either `W` or `X`.

Usage

Table 16-9 SUB (64-bit general registers) specifier combinations

R	*extend*
W	SXTB
W	SXTH
W	SXTW

Table 16-9 SUB (64-bit general registers) specifier combinations (continued)

R	*extend*
W	UXTB
W	UXTH
W	UXTW
X	LSL|UXTX
X	SXTX

Related references

16.1 A64 instructions in alphabetical order on page 16-798.

16.147 SUB (immediate)

Subtract (immediate).

Syntax

`SUB Wd|WSP, Wn|WSP, #imm{, shift} ; 32-bit`

`SUB Xd|SP, Xn|SP, #imm{, shift} ; 64-bit`

Where:

Wd|WSP
Is the 32-bit name of the destination general-purpose register or stack pointer.

Wn|WSP
Is the 32-bit name of the source general-purpose register or stack pointer.

Xd|SP
Is the 64-bit name of the destination general-purpose register or stack pointer.

Xn|SP
Is the 64-bit name of the source general-purpose register or stack pointer.

imm
Is an unsigned immediate, in the range 0 to 4095.

shift
Is the optional left shift to apply to the immediate, defaulting to `LSL #0`, and can be either `LSL #0` or `LSL #12`.

Operation

Subtract (immediate) subtracts an optionally-shifted immediate value from a register value, and writes the result to the destination register.

`Rd = Rn - shift(imm)`, where *R* is either `W` or `X`.

Related references

16.1 A64 instructions in alphabetical order on page 16-798.

16.148 SUB (shifted register)

Subtract (shifted register).

This instruction is used by the alias NEG (shifted register).

Syntax

```
SUB Wd, Wn, Wm{, shift #amount} ; 32-bit

SUB Xd, Xn, Xm{, shift #amount} ; 64-bit
```

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

Wn
Is the 32-bit name of the first general-purpose source register.

Wm
Is the 32-bit name of the second general-purpose source register.

amount
Depends on the instruction variant:

32-bit general registers
Is the shift amount, in the range 0 to 31, defaulting to 0.

64-bit general registers
Is the shift amount, in the range 0 to 63, defaulting to 0.

Xd
Is the 64-bit name of the general-purpose destination register.

Xn
Is the 64-bit name of the first general-purpose source register.

Xm
Is the 64-bit name of the second general-purpose source register.

shift
Is the optional shift type to be applied to the second source operand, defaulting to LSL, and can be one of LSL, LSR, or ASR.

Operation

Subtract (shifted register) subtracts an optionally-shifted register value from a register value, and writes the result to the destination register.

Rd = Rn - *shift*(Rm, *amount*), where *R* is either W or X.

Related references

16.108 NEG (shifted register) on page 16-917.
16.1 A64 instructions in alphabetical order on page 16-798.

16.149 SUBS (extended register)

Subtract (extended register), setting flags.

This instruction is used by the alias CMP (extended register).

Syntax

```
SUBS Wd, Wn|WSP, Wm{, extend {#amount}} ; 32-bit

SUBS Xd, Xn|SP, Rm{, extend {#amount}} ; 64-bit
```

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

Wn|WSP
Is the 32-bit name of the first source general-purpose register or stack pointer.

Wm
Is the 32-bit name of the second general-purpose source register.

extend
Is the extension to be applied to the second source operand:

32-bit general registers
Can be one of UXTB, UXTH, LSL|UXTW, UXTX, SXTB, SXTH, SXTW or SXTX.

If *Rn* is WSP then LSL is preferred rather than UXTW, and can be omitted when *amount* is 0. In all other cases *extend* is required and must be UXTW rather than LSL.

64-bit general registers
Can be one of UXTB, UXTH, UXTW, LSL|UXTX, SXTB, SXTH, SXTW or SXTX.

If *Rn* is SP then LSL is preferred rather than UXTX, and can be omitted when *amount* is 0. In all other cases *extend* is required and must be UXTX rather than LSL.

Xd
Is the 64-bit name of the general-purpose destination register.

Xn|SP
Is the 64-bit name of the first source general-purpose register or stack pointer.

R
Is a width specifier, and can be either W or X.

m
Is the number [0-30] of the second general-purpose source register or the name ZR (31).

amount
Is the left shift amount to be applied after extension in the range 0 to 4, defaulting to 0. It must be absent when *extend* is absent, is required when *extend* is LSL, and is optional when *extend* is present but not LSL.

Operation

Subtract (extended register), setting flags, subtracts a sign or zero-extended register value, followed by an optional left shift amount, from a register value, and writes the result to the destination register. The argument that is extended from the *Rm* register can be a byte, halfword, word, or doubleword. It updates the condition flags based on the result.

*R*d = *R*n - LSL(*extend*(*R*m), *amount*), where *R* is either W or X.

Usage

Table 16-10 SUBS (64-bit general registers) specifier combinations

R	*extend*
W	SXTB
W	SXTH
W	SXTW
W	UXTB
W	UXTH
W	UXTW
X	LSL|UXTX
X	SXTX

Related references

16.54 CMP (extended register) on page 16-860.
16.1 A64 instructions in alphabetical order on page 16-798.

16.150 SUBS (immediate)

Subtract (immediate), setting flags.

This instruction is used by the alias `CMP` (immediate).

Syntax

`SUBS Wd, Wn|WSP, #imm{, shift} ; 32-bit`

`SUBS Xd, Xn|SP, #imm{, shift} ; 64-bit`

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

Wn|WSP
Is the 32-bit name of the source general-purpose register or stack pointer.

Xd
Is the 64-bit name of the general-purpose destination register.

Xn|SP
Is the 64-bit name of the source general-purpose register or stack pointer.

imm
Is an unsigned immediate, in the range 0 to 4095.

shift
Is the optional left shift to apply to the immediate, defaulting to `LSL #0`, and can be either `LSL #0` or `LSL #12`.

Operation

Subtract (immediate), setting flags, subtracts an optionally-shifted immediate value from a register value, and writes the result to the destination register. It updates the condition flags based on the result.

`Rd = Rn - shift(imm)`, where `R` is either `W` or `X`.

Related references

16.55 CMP (immediate) on page 16-862.
16.1 A64 instructions in alphabetical order on page 16-798.

16.151 SUBS (shifted register)

Subtract (shifted register), setting flags.

This instruction is used by the aliases:

- `CMP` (shifted register).
- `NEGS`.

Syntax

```
SUBS Wd, Wn, Wm{, shift #amount} ; 32-bit
```

```
SUBS Xd, Xn, Xm{, shift #amount} ; 64-bit
```

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

Wn
Is the 32-bit name of the first general-purpose source register.

Wm
Is the 32-bit name of the second general-purpose source register.

amount
Depends on the instruction variant:

32-bit general registers
Is the shift amount, in the range 0 to 31, defaulting to 0.

64-bit general registers
Is the shift amount, in the range 0 to 63, defaulting to 0.

Xd
Is the 64-bit name of the general-purpose destination register.

Xn
Is the 64-bit name of the first general-purpose source register.

Xm
Is the 64-bit name of the second general-purpose source register.

shift
Is the optional shift type to be applied to the second source operand, defaulting to LSL, and can be one of `LSL`, `LSR`, or `ASR`.

Operation

Subtract (shifted register), setting flags, subtracts an optionally-shifted register value from a register value, and writes the result to the destination register. It updates the condition flags based on the result.

`Rd = Rn - shift(Rm, amount)`, where `R` is either `W` or `X`.

Related references

16.56 CMP (shifted register) on page 16-863.
16.109 NEGS on page 16-918.
16.1 A64 instructions in alphabetical order on page 16-798.

16.152 SVC

Supervisor call to allow application code to call the OS. It generates an exception targeting exception level 1 (EL1).

Syntax

`SVC #imm`

Where:

imm

Is a 16-bit unsigned immediate, in the range 0 to 65535. This value is made available to the handler in the Exception Syndrome Register.

Usage

Supervisor Call causes an exception to be taken to EL1.

On executing an `SVC` instruction, the PE records the exception as a Supervisor Call exception in *ESR_ELx*, using the EC value `0x15`, and the value of the immediate argument.

Related references

16.1 A64 instructions in alphabetical order on page 16-798.

16.153 SXTB

Signed Extend Byte.

This instruction is an alias of `SBFM`.

The equivalent instruction is `SBFM Wd, Wn, #0, #7`.

Syntax

```
SXTB Wd, Wn ; 32-bit

SXTB Xd, Wn ; 64-bit
```

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

Xd
Is the 64-bit name of the general-purpose destination register.

Wn
Is the 32-bit name of the general-purpose source register.

Operation

Signed Extend Byte extracts an 8-bit value from a register, sign-extends it to the size of the register, and writes the result to the destination register.

`Rd = SignExtend(Wn<7:0>)`, where *R* is either `W` or `X`.

Related references

16.135 SBFM on page 16-944.
16.1 A64 instructions in alphabetical order on page 16-798.

16.154 SXTH

Sign Extend Halfword.

This instruction is an alias of `SBFM`.

The equivalent instruction is `SBFM Wd, Wn, #0, #15`.

Syntax

```
SXTH Wd, Wn ; 32-bit

SXTH Xd, Wn ; 64-bit
```

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

Xd
Is the 64-bit name of the general-purpose destination register.

Wn
Is the 32-bit name of the general-purpose source register.

Operation

Sign Extend Halfword extracts a 16-bit value, sign-extends it to the size of the register, and writes the result to the destination register.

`Rd = SignExtend(Wn<15:0>)`, where *R* is either `W` or `X`.

Related references

16.135 SBFM on page 16-944.
16.1 A64 instructions in alphabetical order on page 16-798.

16.155 SXTW

Sign Extend Word.

This instruction is an alias of `SBFM`.

The equivalent instruction is `SBFM Xd, Xn, #0, #31`.

Syntax

`SXTW Xd, Wn`

Where:

Xd
Is the 64-bit name of the general-purpose destination register.

Xn
Is the 64-bit name of the general-purpose source register.

Wn
Is the 32-bit name of the general-purpose source register.

Operation

Sign Extend Word sign-extends a word to the size of the register, and writes the result to the destination register.

```
Xd = SignExtend(Wn<31:0>).
```

Related references

16.135 SBFM on page 16-944.
16.1 A64 instructions in alphabetical order on page 16-798.

16.156 SYS

System instruction.

This instruction is used by the aliases:

- `AT`.
- `DC`.
- `IC`.
- `TLBI`.

Syntax

`SYS #op1, Cn, Cm, #op2{, Xt}`

Where:

op1
Is a 3-bit unsigned immediate, in the range 0 to 7.

Cn
Is a name `Cn`, with `n` in the range 0 to 15.

Cm
Is a name `Cm`, with `m` in the range 0 to 15.

op2
Is a 3-bit unsigned immediate, in the range 0 to 7.

Xt
Is the 64-bit name of the optional general-purpose source register, defaulting to 31.

Usage

System instruction. For more information, see *Op0 equals `0b01`, cache maintenance, TLB maintenance, and address translation instructions* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile* for the encodings of System instructions.

Related references

16.21 AT on page 16-825.
16.66 DC on page 16-873.
16.83 IC on page 16-892.
16.160 TLBI on page 16-971.
16.1 A64 instructions in alphabetical order on page 16-798.

16.157 SYSL

System instruction with result.

Syntax

```
SYSL Xt, #op1, Cn, Cm, #op2
```

Where:

Xt
Is the 64-bit name of the general-purpose destination register.

op1
Is a 3-bit unsigned immediate, in the range 0 to 7.

Cn
Is a name `Cn`, with `n` in the range 0 to 15.

Cm
Is a name `Cm`, with `m` in the range 0 to 15.

op2
Is a 3-bit unsigned immediate, in the range 0 to 7.

Usage

System instruction with result. For more information, see *Op0 equals `0b01`, cache maintenance, TLB maintenance, and address translation instructions* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile* for the encodings of System instructions.

Related references

16.1 A64 instructions in alphabetical order on page 16-798.

16.158 TBNZ

Test bit and Branch if Nonzero.

Syntax

```
TBNZ R<t>, #imm, label
```

Where:

R

Is a width specifier, and can be either `W` or `X`.

In assembler source code an X specifier is always permitted, but a W specifier is only permitted when the bit number is less than 32.

`<t>`

Is the number [0-30] of the general-purpose register to be tested or the name ZR (31).

imm

Is the bit number to be tested, in the range 0 to 63.

label

Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range ±32KB.

Usage

Test bit and Branch if Nonzero compares the value of a bit in a general-purpose register with zero, and conditionally branches to a label at a PC-relative offset if the comparison is not equal. It provides a hint that this is not a subroutine call or return. This instruction does not affect condition flags.

Related references

16.1 A64 instructions in alphabetical order on page 16-798.

16.159 TBZ

Test bit and Branch if Zero.

Syntax

```
TBZ R<t>, #imm, label
```

Where:

R

Is a width specifier, and can be either `W` or `X`.

In assembler source code an X specifier is always permitted, but a W specifier is only permitted when the bit number is less than 32.

`<t>`

Is the number [0-30] of the general-purpose register to be tested or the name ZR (31).

imm

Is the bit number to be tested, in the range 0 to 63.

label

Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range ±32KB.

Usage

Test bit and Branch if Zero compares the value of a test bit with zero, and conditionally branches to a label at a PC-relative offset if the comparison is equal. It provides a hint that this is not a subroutine call or return. This instruction does not affect condition flags.

Related references

16.1 A64 instructions in alphabetical order on page 16-798.

16.160 TLBI

TLB Invalidate operation.

This instruction is an alias of SYS.

The equivalent instruction is `SYS #op1, C8, Cm, #op2{, Xt}`.

Syntax

```
TLBI <tlbi_op>{, Xt}
```

Where:

op1
Is a 3-bit unsigned immediate, in the range 0 to 7.

Cm
Is a name Cm, with m in the range 0 to 15.

op2
Is a 3-bit unsigned immediate, in the range 0 to 7.

<tlbi_op>
Is a TLBI instruction name, as listed for the TLBI system instruction group, and can be one of the values shown in Usage.

Xt
Is the 64-bit name of the optional general-purpose source register, defaulting to 31.

Usage

TLB Invalidate operation. For more information, see *A64 system instructions for TLB maintenance* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

The following table shows the valid specifier combinations:

Table 16-11 SYS parameter values corresponding to TLBI operations

<tlbi_op>	*op1*	*Cm*	*op2*
ALLE1	4	7	4
ALLE1IS	4	3	4
ALLE2	4	7	0
ALLE2IS	4	3	0
ALLE3	6	7	0
ALLE3IS	6	3	0
ASIDE1	0	7	2
ASIDE1IS	0	3	2
IPAS2E1	4	4	1
IPAS2E1IS	4	0	1
IPAS2LE1	4	4	5
IPAS2LE1IS	4	0	5
VAAE1	0	7	3
VAAE1IS	0	3	3
VAALE1	0	7	7
VAALE1IS	0	3	7

Table 16-11 SYS parameter values corresponding to TLBI operations (continued)

<tlbi_op>	op1	Cm	op2
VAE1	0	7	1
VAE1IS	0	3	1
VAE2	4	7	1
VAE2IS	4	3	1
VAE3	6	7	1
VAE3IS	6	3	1
VALE1	0	7	5
VALE1IS	0	3	5
VALE2	4	7	5
VALE2IS	4	3	5
VALE3	6	7	5
VALE3IS	6	3	5
VMALLE1	0	7	0
VMALLE1IS	0	3	0
VMALLS12E1	4	7	6
VMALLS12E1IS	4	3	6

Related references

16.156 SYS on page 16-967.
16.1 A64 instructions in alphabetical order on page 16-798.

16.161 TST (immediate)

, setting the condition flags and discarding the result.

This instruction is an alias of ANDS (immediate).

The equivalent instruction is `ANDS WZR, Wn, #imm`.

Syntax

```
TST Wn, #imm ; 32-bit
TST Xn, #imm ; 64-bit
```

Where:

Wn
Is the 32-bit name of the general-purpose source register.

imm
The bitmask immediate.

Xn
Is the 64-bit name of the general-purpose source register.

Operation

`Rn AND imm`, where `R` is either `W` or `X`.

Related references

16.16 ANDS (immediate) on page 16-820.
16.1 A64 instructions in alphabetical order on page 16-798.

16.162 TST (shifted register)

Test (shifted register).

This instruction is an alias of ANDS (shifted register).

The equivalent instruction is `ANDS WZR, Wn, Wm{, shift #amount}`.

Syntax

```
TST Wn, Wm{, shift #amount} ; 32-bit
TST Xn, Xm{, shift #amount} ; 64-bit
```

Where:

Wn
Is the 32-bit name of the first general-purpose source register.

Wm
Is the 32-bit name of the second general-purpose source register.

amount
Depends on the instruction variant:

32-bit general registers
Is the shift amount, in the range 0 to 31, defaulting to 0.

64-bit general registers
Is the shift amount, in the range 0 to 63, defaulting to 0.

Xn
Is the 64-bit name of the first general-purpose source register.

Xm
Is the 64-bit name of the second general-purpose source register.

shift
Is the optional shift to be applied to the final source, defaulting to LSL, and can be one of LSL, LSR, ASR, or ROR.

Operation

Test (shifted register) performs a bitwise AND operation on a register value and an optionally-shifted register value. It updates the condition flags based on the result, and discards the result.

`Rn AND shift(Rm, amount)`, where *R* is either W or X.

Related references

16.17 ANDS (shifted register) on page 16-821.
16.1 A64 instructions in alphabetical order on page 16-798.

16.163 UBFIZ

Unsigned Bitfield Insert in Zero.

This instruction is an alias of `UBFM`.

The equivalent instruction is `UBFM Wd, Wn, #(-lsb MOD 32), #(width-1)`.

Syntax

```
UBFIZ Wd, Wn, #lsb, #width ; 32-bit
UBFIZ Xd, Xn, #lsb, #width ; 64-bit
```

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

Wn
Is the 32-bit name of the general-purpose source register.

lsb
Depends on the instruction variant:

32-bit general registers
Is the bit number of the lsb of the destination bitfield, in the range 0 to 31.

64-bit general registers
Is the bit number of the lsb of the destination bitfield, in the range 0 to 63.

width
Depends on the instruction variant:

32-bit general registers
Is the width of the bitfield, in the range 1 to 32-*lsb*.

64-bit general registers
Is the width of the bitfield, in the range 1 to 64-*lsb*.

Xd
Is the 64-bit name of the general-purpose destination register.

Xn
Is the 64-bit name of the general-purpose source register.

Usage

Unsigned Bitfield Insert in Zero zeros the destination register and copies any number of contiguous bits from a source register into any position in the destination register.

Related references

16.164 UBFM on page 16-976.
16.1 A64 instructions in alphabetical order on page 16-798.

16.164 UBFM

Unsigned Bitfield Move.

This instruction is used by the aliases:

- `LSL` (immediate).
- `LSR` (immediate).
- `UBFIZ`.
- `UBFX`.
- `UXTB`.
- `UXTH`.

Syntax

```
UBFM Wd, Wn, #<immr>, #<imms> ; 32-bit

UBFM Xd, Xn, #<immr>, #<imms> ; 64-bit
```

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

Wn
Is the 32-bit name of the general-purpose source register.

<immr>
Depends on the instruction variant:

32-bit general registers
Is the right rotate amount, in the range 0 to 31.

64-bit general registers
Is the right rotate amount, in the range 0 to 63.

<imms>
Depends on the instruction variant:

32-bit general registers
Is the leftmost bit number to be moved from the source, in the range 0 to 31.

64-bit general registers
Is the leftmost bit number to be moved from the source, in the range 0 to 63.

Xd
Is the 64-bit name of the general-purpose destination register.

Xn
Is the 64-bit name of the general-purpose source register.

Usage

Unsigned Bitfield Move copies any number of low-order bits from a source register into the same number of adjacent bits at any position in the destination register, with zeros in the upper and lower bits.

Related references

16.86 LSL (immediate) on page 16-895.
16.89 LSR (immediate) on page 16-898.
16.163 UBFIZ on page 16-975.
16.165 UBFX on page 16-977.
16.172 UXTB on page 16-984.
16.173 UXTH on page 16-985.
16.1 A64 instructions in alphabetical order on page 16-798.

16.165 UBFX

Unsigned Bitfield Extract.

This instruction is an alias of `UBFM`.

The equivalent instruction is `UBFM Wd, Wn, #lsb, #(lsb+width-1)`.

Syntax

```
UBFX Wd, Wn, #lsb, #width ; 32-bit
UBFX Xd, Xn, #lsb, #width ; 64-bit
```

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

Wn
Is the 32-bit name of the general-purpose source register.

lsb
Depends on the instruction variant:

32-bit general registers
Is the bit number of the lsb of the source bitfield, in the range 0 to 31.

64-bit general registers
Is the bit number of the lsb of the source bitfield, in the range 0 to 63.

width
Depends on the instruction variant:

32-bit general registers
Is the width of the bitfield, in the range 1 to 32-*lsb*.

64-bit general registers
Is the width of the bitfield, in the range 1 to 64-*lsb*.

Xd
Is the 64-bit name of the general-purpose destination register.

Xn
Is the 64-bit name of the general-purpose source register.

Usage

Unsigned Bitfield Extract extracts any number of adjacent bits at any position from a register, zero-extends them to the size of the register, and writes the result to the destination register.

Related references

16.164 UBFM on page 16-976.
16.1 A64 instructions in alphabetical order on page 16-798.

16.166 UDIV

Unsigned Divide.

Syntax

```
UDIV Wd, Wn, Wm ; 32-bit

UDIV Xd, Xn, Xm ; 64-bit
```

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

Wn
Is the 32-bit name of the first general-purpose source register.

Wm
Is the 32-bit name of the second general-purpose source register.

Xd
Is the 64-bit name of the general-purpose destination register.

Xn
Is the 64-bit name of the first general-purpose source register.

Xm
Is the 64-bit name of the second general-purpose source register.

Operation

Unsigned Divide divides an unsigned integer register value by another unsigned integer register value, and writes the result to the destination register. The condition flags are not affected.

`Rd = Rn / Rm`, where `R` is either `W` or `X`.

Related references

16.1 A64 instructions in alphabetical order on page 16-798.

16.167 UMADDL

Unsigned Multiply-Add Long.

This instruction is used by the alias `UMULL`.

Syntax

`UMADDL Xd, Wn, Wm, Xa`

Where:

Xd
Is the 64-bit name of the general-purpose destination register.

Wn
Is the 32-bit name of the first general-purpose source register holding the multiplicand.

Wm
Is the 32-bit name of the second general-purpose source register holding the multiplier.

Xa
Is the 64-bit name of the third general-purpose source register holding the addend.

Operation

Unsigned Multiply-Add Long multiplies two 32-bit register values, adds a 64-bit register value, and writes the result to the 64-bit destination register.

```
Xd = Xa + Wn * Wm.
```

Related references

16.171 UMULL on page 16-983.
16.1 A64 instructions in alphabetical order on page 16-798.

16.168 UMNEGL

Unsigned Multiply-Negate Long.

This instruction is an alias of `UMSUBL`.

The equivalent instruction is `UMSUBL Xd, Wn, Wm, XZR`.

Syntax

`UMNEGL Xd, Wn, Wm`

Where:

Xd
Is the 64-bit name of the general-purpose destination register.

Wn
Is the 32-bit name of the first general-purpose source register holding the multiplicand.

Wm
Is the 32-bit name of the second general-purpose source register holding the multiplier.

Operation

Unsigned Multiply-Negate Long multiplies two 32-bit register values, negates the product, and writes the result to the 64-bit destination register.

```
Xd = -(Wn * Wm).
```

Related references

16.169 UMSUBL on page 16-981.
16.1 A64 instructions in alphabetical order on page 16-798.

16.169 UMSUBL

Unsigned Multiply-Subtract Long.

This instruction is used by the alias `UMNEGL`.

Syntax

`UMSUBL` *`Xd, Wn, Wm, Xa`*

Where:

Xd
Is the 64-bit name of the general-purpose destination register.

Wn
Is the 32-bit name of the first general-purpose source register holding the multiplicand.

Wm
Is the 32-bit name of the second general-purpose source register holding the multiplier.

Xa
Is the 64-bit name of the third general-purpose source register holding the minuend.

Operation

Unsigned Multiply-Subtract Long multiplies two 32-bit register values, subtracts the product from a 64-bit register value, and writes the result to the 64-bit destination register.

```
Xd = Xa - Wn * Wm.
```

Related references

16.168 UMNEGL on page 16-980.
16.1 A64 instructions in alphabetical order on page 16-798.

16.170 UMULH

Unsigned Multiply High.

Syntax

`UMULH Xd, Xn, Xm`

Where:

Xd
Is the 64-bit name of the general-purpose destination register.

Xn
Is the 64-bit name of the first general-purpose source register holding the multiplicand.

Xm
Is the 64-bit name of the second general-purpose source register holding the multiplier.

Operation

Unsigned Multiply High multiplies two 64-bit register values, and writes bits[127:64] of the 128-bit result to the 64-bit destination register.

```
Xd = bits<127:64> of Xn * Xm.
```

Related references

16.1 A64 instructions in alphabetical order on page 16-798.

16.171 UMULL

Unsigned Multiply Long.

This instruction is an alias of `UMADDL`.

The equivalent instruction is `UMADDL Xd, Wn, Wm, XZR`.

Syntax

`UMULL Xd, Wn, Wm`

Where:

Xd
Is the 64-bit name of the general-purpose destination register.

Wn
Is the 32-bit name of the first general-purpose source register holding the multiplicand.

Wm
Is the 32-bit name of the second general-purpose source register holding the multiplier.

Operation

Unsigned Multiply Long multiplies two 32-bit register values, and writes the result to the 64-bit destination register.

```
Xd = Wn * Wm.
```

Related references

16.167 UMADDL on page 16-979.
16.1 A64 instructions in alphabetical order on page 16-798.

16.172 UXTB

Unsigned Extend Byte.

This instruction is an alias of `UBFM`.

The equivalent instruction is `UBFM Wd, Wn, #0, #7`.

Syntax

`UXTB Wd, Wn`

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

Wn
Is the 32-bit name of the general-purpose source register.

Operation

Unsigned Extend Byte extracts an 8-bit value from a register, zero-extends it to the size of the register, and writes the result to the destination register.

```
Wd = ZeroExtend(Wn<7:0>).
```

Related references

16.164 UBFM on page 16-976.
16.1 A64 instructions in alphabetical order on page 16-798.

16.173 UXTH

Unsigned Extend Halfword.

This instruction is an alias of `UBFM`.

The equivalent instruction is `UBFM Wd, Wn, #0, #15`.

Syntax

`UXTH Wd, Wn`

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

Wn
Is the 32-bit name of the general-purpose source register.

Operation

Unsigned Extend Halfword extracts a 16-bit value from a register, zero-extends it to the size of the register, and writes the result to the destination register.

```
Wd = ZeroExtend(Wn<15:0>).
```

Related references

16.164 UBFM on page 16-976.
16.1 A64 instructions in alphabetical order on page 16-798.

16.174 WFE

Wait For Event.

Usage

Wait For Event is a hint instruction that permits the PE to enter a low-power state until one of a number of events occurs, including events signaled by executing the `SEV` instruction on any PE in the multiprocessor system. For more information, see *Wait For Event mechanism and Send event* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

As described in *Wait For Event mechanism and Send event* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*, the execution of a `WFE` instruction that would otherwise cause entry to a low-power state can be trapped to a higher Exception level. See:

- *Traps to EL1 of EL0 execution of WFE and WFI instructions*.
- *Traps to EL2 of Non-secure EL0 and EL1 execution of WFE and WFI instructions*.
- *Traps to EL3 of EL2, EL1, and EL0 execution of WFE and WFI instructions*.

Related references

16.1 A64 instructions in alphabetical order on page 16-798.

16.175 WFI

Wait For Interrupt.

Usage

Wait For Interrupt is a hint instruction that permits the PE to enter a low-power state until one of a number of asynchronous event occurs. For more information, see *Wait For Interrupt* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

As described in *Wait For Interrupt* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*, the execution of a `WFI` instruction that would otherwise cause entry to a low-power state can be trapped to a higher Exception level. See:

- *Traps to EL1 of EL0 execution of WFE and WFI instructions*.
- *Traps to EL2 of Non-secure EL0 and EL1 execution of WFE and WFI instructions*.
- *Traps to EL3 of EL2, EL1, and EL0 execution of WFE and WFI instructions*.

Related references

16.1 A64 instructions in alphabetical order on page 16-798.

16.176 XPACD, XPACI, XPACLRI

Strip Pointer Authentication Code.

Syntax

```
XPACD Xd ; XPACD general registers

XPACI Xd ; XPACI general registers

XPACLRI
```

Where:

Xd
Is the 64-bit name of the general-purpose destination register.

Architectures supported

Supported in the Armv8.3-A architecture and later.

Usage

Strip Pointer Authentication Code. This instruction removes the pointer authentication code from an address. The address is in the specified general-purpose register for `XPACI` and `XPACD`, and is in LR for `XPACLRI`.

The `XPACD` instruction is used for data addresses, and `XPACI` and `XPACLRI` are used for instruction addresses.

Related references

16.1 A64 instructions in alphabetical order on page 16-798.

16.177 YIELD

YIELD.

Usage

YIELD is a hint instruction. Software with a multithreading capability can use a `YIELD` instruction to indicate to the PE that it is performing a task, for example a spin-lock, that could be swapped out to improve overall system performance. The PE can use this hint to suspend and resume multiple software threads if it supports the capability.

For more information about the recommended use of this instruction, see *The YIELD instruction* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

16.1 A64 instructions in alphabetical order on page 16-798.

Chapter 17
A64 Data Transfer Instructions

Describes the A64 data transfer instructions.

It contains the following sections:

17.1 A64 data transfer instructions in alphabetical order

A summary of the A64 data transfer instructions and pseudo-instructions that are supported.

Table 17-1 Summary of A64 data transfer instructions

Mnemonic	Brief description	See
CASA, CASAL, CAS, CASL, CASAL, CAS, CASL	Compare and Swap word or doubleword in memory	*17.2 CASA, CASAL, CAS, CASL, CASAL, CAS, CASL* on page 17-1000
CASAB, CASALB, CASB, CASLB	Compare and Swap byte in memory	*17.3 CASAB, CASALB, CASB, CASLB* on page 17-1001
CASAH, CASALH, CASH, CASLH	Compare and Swap halfword in memory	*17.4 CASAH, CASALH, CASH, CASLH* on page 17-1002
CASPA, CASPAL, CASP, CASPL, CASPAL, CASP, CASPL	Compare and Swap Pair of words or doublewords in memory	*17.5 CASPA, CASPAL, CASP, CASPL, CASPAL, CASP, CASPL* on page 17-1003
LDADDA, LDADDAL, LDADD, LDADDL, LDADDAL, LDADD, LDADDL	Atomic add on word or doubleword in memory	*17.6 LDADDA, LDADDAL, LDADD, LDADDL, LDADDAL, LDADD, LDADDL* on page 17-1005
LDADDAB, LDADDALB, LDADDB, LDADDLB	Atomic add on byte in memory	*17.7 LDADDAB, LDADDALB, LDADDB, LDADDLB* on page 17-1006
LDADDAH, LDADDALH, LDADDH, LDADDLH	Atomic add on halfword in memory	*17.8 LDADDAH, LDADDALH, LDADDH, LDADDLH* on page 17-1007
LDAPR	Load-Acquire RCpc Register	*17.9 LDAPR* on page 17-1008
LDAPRB	Load-Acquire RCpc Register Byte	*17.10 LDAPRB* on page 17-1009
LDAPRH	Load-Acquire RCpc Register Halfword	*17.11 LDAPRH* on page 17-1010
LDAR	Load-Acquire Register	*17.12 LDAR* on page 17-1011
LDARB	Load-Acquire Register Byte	*17.13 LDARB* on page 17-1012
LDARH	Load-Acquire Register Halfword	*17.14 LDARH* on page 17-1013
LDAXP	Load-Acquire Exclusive Pair of Registers	*17.15 LDAXP* on page 17-1014
LDAXR	Load-Acquire Exclusive Register	*17.16 LDAXR* on page 17-1015
LDAXRB	Load-Acquire Exclusive Register Byte	*17.17 LDAXRB* on page 17-1016
LDAXRH	Load-Acquire Exclusive Register Halfword	*17.18 LDAXRH* on page 17-1017
LDCLRA, LDCLRAL, LDCLR, LDCLRL, LDCLRAL, LDCLR, LDCLRL	Atomic bit clear on word or doubleword in memory	*17.19 LDCLRA, LDCLRAL, LDCLR, LDCLRL, LDCLRAL, LDCLR, LDCLRL* on page 17-1018
LDCLRAB, LDCLRALB, LDCLRB, LDCLRLB	Atomic bit clear on byte in memory	*17.20 LDCLRAB, LDCLRALB, LDCLRB, LDCLRLB* on page 17-1019
LDCLRAH, LDCLRALH, LDCLRH, LDCLRLH	Atomic bit clear on halfword in memory	*17.21 LDCLRAH, LDCLRALH, LDCLRH, LDCLRLH* on page 17-1020
LDEORA, LDEORAL, LDEOR, LDEORL, LDEORAL, LDEOR, LDEORL	Atomic exclusive OR on word or doubleword in memory	*17.22 LDEORA, LDEORAL, LDEOR, LDEORL, LDEORAL, LDEOR, LDEORL* on page 17-1021

Table 17-1 Summary of A64 data transfer instructions (continued)

Mnemonic	Brief description	See
LDEORAB, LDEORALB, LDEORB, LDEORLB	Atomic exclusive OR on byte in memory	*17.23 LDEORAB, LDEORALB, LDEORB, LDEORLB* on page 17-1022
LDEORAH, LDEORALH, LDEORH, LDEORLH	Atomic exclusive OR on halfword in memory	*17.24 LDEORAH, LDEORALH, LDEORH, LDEORLH* on page 17-1023
LDLAR	Load LOAcquire Register	*17.25 LDLAR* on page 17-1024
LDLARB	Load LOAcquire Register Byte	*17.26 LDLARB* on page 17-1025
LDLARH	Load LOAcquire Register Halfword	*17.27 LDLARH* on page 17-1026
LDNP	Load Pair of Registers, with non-temporal hint	*17.28 LDNP* on page 17-1027
LDP	Load Pair of Registers	*17.29 LDP* on page 17-1028
LDPSW	Load Pair of Registers Signed Word	*17.30 LDPSW* on page 17-1029
LDR (immediate)	Load Register (immediate)	*17.31 LDR (immediate)* on page 17-1030
LDR (literal)	Load Register (literal)	*17.32 LDR (literal)* on page 17-1031
LDR pseudo-instruction	Load a register with either a 32-bit or 64-bit immediate value or any address	*17.33 LDR pseudo-instruction* on page 17-1032
LDR (register)	Load Register (register)	*17.34 LDR (register)* on page 17-1034
LDRAA, LDRAB, LDRAB	Load Register, with pointer authentication	*17.35 LDRAA, LDRAB, LDRAB* on page 17-1035
LDRB (immediate)	Load Register Byte (immediate)	*17.36 LDRB (immediate)* on page 17-1036
LDRB (register)	Load Register Byte (register)	*17.37 LDRB (register)* on page 17-1037
LDRH (immediate)	Load Register Halfword (immediate)	*17.38 LDRH (immediate)* on page 17-1038
LDRH (register)	Load Register Halfword (register)	*17.39 LDRH (register)* on page 17-1039
LDRSB (immediate)	Load Register Signed Byte (immediate)	*17.40 LDRSB (immediate)* on page 17-1040
LDRSB (register)	Load Register Signed Byte (register)	*17.41 LDRSB (register)* on page 17-1041
LDRSH (immediate)	Load Register Signed Halfword (immediate)	*17.42 LDRSH (immediate)* on page 17-1042
LDRSH (register)	Load Register Signed Halfword (register)	*17.43 LDRSH (register)* on page 17-1043
LDRSW (immediate)	Load Register Signed Word (immediate)	*17.44 LDRSW (immediate)* on page 17-1044
LDRSW (literal)	Load Register Signed Word (literal)	*17.45 LDRSW (literal)* on page 17-1045
LDRSW (register)	Load Register Signed Word (register)	*17.46 LDRSW (register)* on page 17-1046
LDSETA, LDSETAL, LDSET, LDSETL, LDSETAL, LDSET, LDSETL	Atomic bit set on word or doubleword in memory	*17.47 LDSETA, LDSETAL, LDSET, LDSETL, LDSETAL, LDSET, LDSETL* on page 17-1047
LDSETAB, LDSETALB, LDSETB, LDSETLB	Atomic bit set on byte in memory	*17.48 LDSETAB, LDSETALB, LDSETB, LDSETLB* on page 17-1048
LDSETAH, LDSETALH, LDSETH, LDSETLH	Atomic bit set on halfword in memory	*17.49 LDSETAH, LDSETALH, LDSETH, LDSETLH* on page 17-1049

Table 17-1 Summary of A64 data transfer instructions (continued)

Mnemonic	Brief description	See
LDSMAXA, LDSMAXAL, LDSMAX, LDSMAXL, LDSMAXAL, LDSMAX, LDSMAXL	Atomic signed maximum on word or doubleword in memory	*17.50 LDSMAXA, LDSMAXAL, LDSMAX, LDSMAXL, LDSMAXAL, LDSMAX, LDSMAXL* on page 17-1050
LDSMAXAB, LDSMAXALB, LDSMAXB, LDSMAXLB	Atomic signed maximum on byte in memory	*17.51 LDSMAXAB, LDSMAXALB, LDSMAXB, LDSMAXLB* on page 17-1051
LDSMAXAH, LDSMAXALH, LDSMAXH, LDSMAXLH	Atomic signed maximum on halfword in memory	*17.52 LDSMAXAH, LDSMAXALH, LDSMAXH, LDSMAXLH* on page 17-1052
LDSMINA, LDSMINAL, LDSMIN, LDSMINL, LDSMINAL, LDSMIN, LDSMINL	Atomic signed minimum on word or doubleword in memory	*17.53 LDSMINA, LDSMINAL, LDSMIN, LDSMINL, LDSMINAL, LDSMIN, LDSMINL* on page 17-1053
LDSMINAB, LDSMINALB, LDSMINB, LDSMINLB	Atomic signed minimum on byte in memory	*17.54 LDSMINAB, LDSMINALB, LDSMINB, LDSMINLB* on page 17-1054
LDSMINAH, LDSMINALH, LDSMINH, LDSMINLH	Atomic signed minimum on halfword in memory	*17.55 LDSMINAH, LDSMINALH, LDSMINH, LDSMINLH* on page 17-1055
LDTR	Load Register (unprivileged)	*17.56 LDTR* on page 17-1056
LDTRB	Load Register Byte (unprivileged)	*17.57 LDTRB* on page 17-1057
LDTRH	Load Register Halfword (unprivileged)	*17.58 LDTRH* on page 17-1058
LDTRSB	Load Register Signed Byte (unprivileged)	*17.59 LDTRSB* on page 17-1059
LDTRSH	Load Register Signed Halfword (unprivileged)	*17.60 LDTRSH* on page 17-1060
LDTRSW	Load Register Signed Word (unprivileged)	*17.61 LDTRSW* on page 17-1061
LDUMAXA, LDUMAXAL, LDUMAX, LDUMAXL, LDUMAXAL, LDUMAX, LDUMAXL	Atomic unsigned maximum on word or doubleword in memory	*17.62 LDUMAXA, LDUMAXAL, LDUMAX, LDUMAXL, LDUMAXAL, LDUMAX, LDUMAXL* on page 17-1062
LDUMAXAB, LDUMAXALB, LDUMAXB, LDUMAXLB	Atomic unsigned maximum on byte in memory	*17.63 LDUMAXAB, LDUMAXALB, LDUMAXB, LDUMAXLB* on page 17-1063
LDUMAXAH, LDUMAXALH, LDUMAXH, LDUMAXLH	Atomic unsigned maximum on halfword in memory	*17.64 LDUMAXAH, LDUMAXALH, LDUMAXH, LDUMAXLH* on page 17-1064
LDUMINA, LDUMINAL, LDUMIN, LDUMINL, LDUMINAL, LDUMIN, LDUMINL	Atomic unsigned minimum on word or doubleword in memory	*17.65 LDUMINA, LDUMINAL, LDUMIN, LDUMINL, LDUMINAL, LDUMIN, LDUMINL* on page 17-1065
LDUMINAB, LDUMINALB, LDUMINB, LDUMINLB	Atomic unsigned minimum on byte in memory	*17.66 LDUMINAB, LDUMINALB, LDUMINB, LDUMINLB* on page 17-1066
LDUMINAH, LDUMINALH, LDUMINH, LDUMINLH	Atomic unsigned minimum on halfword in memory	*17.67 LDUMINAH, LDUMINALH, LDUMINH, LDUMINLH* on page 17-1067
LDUR	Load Register (unscaled)	*17.68 LDUR* on page 17-1068
LDURB	Load Register Byte (unscaled)	*17.69 LDURB* on page 17-1069
LDURH	Load Register Halfword (unscaled)	*17.70 LDURH* on page 17-1070

Table 17-1 Summary of A64 data transfer instructions (continued)

Mnemonic	Brief description	See
LDURSB	Load Register Signed Byte (unscaled)	*17.71 LDURSB* on page 17-1071
LDURSH	Load Register Signed Halfword (unscaled)	*17.72 LDURSH* on page 17-1072
LDURSW	Load Register Signed Word (unscaled)	*17.73 LDURSW* on page 17-1073
LDXP	Load Exclusive Pair of Registers	*17.74 LDXP* on page 17-1074
LDXR	Load Exclusive Register	*17.75 LDXR* on page 17-1075
LDXRB	Load Exclusive Register Byte	*17.76 LDXRB* on page 17-1076
LDXRH	Load Exclusive Register Halfword	*17.77 LDXRH* on page 17-1077
PRFM (immediate)	Prefetch Memory (immediate)	*17.78 PRFM (immediate)* on page 17-1078
PRFM (literal)	Prefetch Memory (literal)	*17.79 PRFM (literal)* on page 17-1079
PRFM (register)	Prefetch Memory (register)	*17.80 PRFM (register)* on page 17-1080
PRFUM (unscaled offset)	Prefetch Memory (unscaled offset)	*17.81 PRFUM (unscaled offset)* on page 17-1082
STADD, STADDL, STADDL	Atomic add on word or doubleword in memory, without return	*17.82 STADD, STADDL, STADDL* on page 17-1083
STADDB, STADDLB	Atomic add on byte in memory, without return	*17.83 STADDB, STADDLB* on page 17-1084
STADDH, STADDLH	Atomic add on halfword in memory, without return	*17.84 STADDH, STADDLH* on page 17-1085
STCLR, STCLRL, STCLRL	Atomic bit clear on word or doubleword in memory, without return	*17.85 STCLR, STCLRL, STCLRL* on page 17-1086
STCLRB, STCLRLB	Atomic bit clear on byte in memory, without return	*17.86 STCLRB, STCLRLB* on page 17-1087
STCLRH, STCLRLH	Atomic bit clear on halfword in memory, without return	*17.87 STCLRH, STCLRLH* on page 17-1088
STEOR, STEORL, STEORL	Atomic exclusive OR on word or doubleword in memory, without return	*17.88 STEOR, STEORL, STEORL* on page 17-1089
STEORB, STEORLB	Atomic exclusive OR on byte in memory, without return	*17.89 STEORB, STEORLB* on page 17-1090
STEORH, STEORLH	Atomic exclusive OR on halfword in memory, without return	*17.90 STEORH, STEORLH* on page 17-1091
STLLR	Store LORelease Register	*17.91 STLLR* on page 17-1092
STLLRB	Store LORelease Register Byte	*17.92 STLLRB* on page 17-1093
STLLRH	Store LORelease Register Halfword	*17.93 STLLRH* on page 17-1094
STLR	Store-Release Register	*17.94 STLR* on page 17-1095
STLRB	Store-Release Register Byte	*17.95 STLRB* on page 17-1096
STLRH	Store-Release Register Halfword	*17.96 STLRH* on page 17-1097
STLXP	Store-Release Exclusive Pair of registers	*17.97 STLXP* on page 17-1098
STLXR	Store-Release Exclusive Register	*17.98 STLXR* on page 17-1100
STLXRB	Store-Release Exclusive Register Byte	*17.99 STLXRB* on page 17-1102

Table 17-1 Summary of A64 data transfer instructions (continued)

Mnemonic	Brief description	See
STLXRH	Store-Release Exclusive Register Halfword	*17.100 STLXRH* on page 17-1103
STNP	Store Pair of Registers, with non-temporal hint	*17.101 STNP* on page 17-1104
STP	Store Pair of Registers	*17.102 STP* on page 17-1105
STR (immediate)	Store Register (immediate)	*17.103 STR (immediate)* on page 17-1106
STR (register)	Store Register (register)	*17.104 STR (register)* on page 17-1107
STRB (immediate)	Store Register Byte (immediate)	*17.105 STRB (immediate)* on page 17-1108
STRB (register)	Store Register Byte (register)	*17.106 STRB (register)* on page 17-1109
STRH (immediate)	Store Register Halfword (immediate)	*17.107 STRH (immediate)* on page 17-1110
STRH (register)	Store Register Halfword (register)	*17.108 STRH (register)* on page 17-1111
STSET, STSETL, STSETL	Atomic bit set on word or doubleword in memory, without return	*17.109 STSET, STSETL, STSETL* on page 17-1112
STSETB, STSETLB	Atomic bit set on byte in memory, without return	*17.110 STSETB, STSETLB* on page 17-1113
STSETH, STSETLH	Atomic bit set on halfword in memory, without return	*17.111 STSETH, STSETLH* on page 17-1114
STSMAX, STSMAXL, STSMAXL	Atomic signed maximum on word or doubleword in memory, without return	*17.112 STSMAX, STSMAXL, STSMAXL* on page 17-1115
STSMAXB, STSMAXLB	Atomic signed maximum on byte in memory, without return	*17.113 STSMAXB, STSMAXLB* on page 17-1116
STSMAXH, STSMAXLH	Atomic signed maximum on halfword in memory, without return	*17.114 STSMAXH, STSMAXLH* on page 17-1117
STSMIN, STSMINL, STSMINL	Atomic signed minimum on word or doubleword in memory, without return	*17.115 STSMIN, STSMINL, STSMINL* on page 17-1118
STSMINB, STSMINLB	Atomic signed minimum on byte in memory, without return	*17.116 STSMINB, STSMINLB* on page 17-1119
STSMINH, STSMINLH	Atomic signed minimum on halfword in memory, without return	*17.117 STSMINH, STSMINLH* on page 17-1120
STTR	Store Register (unprivileged)	*17.118 STTR* on page 17-1121
STTRB	Store Register Byte (unprivileged)	*17.119 STTRB* on page 17-1122
STTRH	Store Register Halfword (unprivileged)	*17.120 STTRH* on page 17-1123
STUMAX, STUMAXL, STUMAXL	Atomic unsigned maximum on word or doubleword in memory, without return	*17.121 STUMAX, STUMAXL, STUMAXL* on page 17-1124
STUMAXB, STUMAXLB	Atomic unsigned maximum on byte in memory, without return	*17.122 STUMAXB, STUMAXLB* on page 17-1125
STUMAXH, STUMAXLH	Atomic unsigned maximum on halfword in memory, without return	*17.123 STUMAXH, STUMAXLH* on page 17-1126
STUMIN, STUMINL, STUMINL	Atomic unsigned minimum on word or doubleword in memory, without return	*17.124 STUMIN, STUMINL, STUMINL* on page 17-1127

Table 17-1 Summary of A64 data transfer instructions (continued)

Mnemonic	Brief description	See
STUMINB, STUMINLB	Atomic unsigned minimum on byte in memory, without return	*17.125 STUMINB, STUMINLB* on page 17-1128
STUMINH, STUMINLH	Atomic unsigned minimum on halfword in memory, without return	*17.126 STUMINH, STUMINLH* on page 17-1129
STUR	Store Register (unscaled)	*17.127 STUR* on page 17-1130
STURB	Store Register Byte (unscaled)	*17.128 STURB* on page 17-1131
STURH	Store Register Halfword (unscaled)	*17.129 STURH* on page 17-1132
STXP	Store Exclusive Pair of registers	*17.130 STXP* on page 17-1133
STXR	Store Exclusive Register	*17.131 STXR* on page 17-1135
STXRB	Store Exclusive Register Byte	*17.132 STXRB* on page 17-1136
STXRH	Store Exclusive Register Halfword	*17.133 STXRH* on page 17-1137
SWPA, SWPAL, SWP, SWPL, SWPAL, SWP, SWPL	Swap word or doubleword in memory	*17.134 SWPA, SWPAL, SWP, SWPL, SWPAL, SWP, SWPL* on page 17-1138
SWPAB, SWPALB, SWPB, SWPLB	Swap byte in memory	*17.135 SWPAB, SWPALB, SWPB, SWPLB* on page 17-1139
SWPAH, SWPALH, SWPH, SWPLH	Swap halfword in memory	*17.136 SWPAH, SWPALH, SWPH, SWPLH* on page 17-1140

17.2 CASA, CASAL, CAS, CASL, CASAL, CAS, CASL

Compare and Swap word or doubleword in memory.

Syntax

```
CASA Ws, Wt, [Xn|SP{,#0}] ; 32-bit, acquire general registers

CASAL Ws, Wt, [Xn|SP{,#0}] ; 32-bit, acquire and release general registers

CAS Ws, Wt, [Xn|SP{,#0}] ; 32-bit, no memory ordering general registers

CASL Ws, Wt, [Xn|SP{,#0}] ; 32-bit, release general registers

CASA Xs, Xt, [Xn|SP{,#0}] ; 64-bit, acquire general registers

CASAL Xs, Xt, [Xn|SP{,#0}] ; 64-bit, acquire and release general registers

CAS Xs, Xt, [Xn|SP{,#0}] ; 64-bit, no memory ordering general registers

CASL Xs, Xt, [Xn|SP{,#0}] ; 64-bit, release general registers
```

Where:

Ws
Is the 32-bit name of the general-purpose register to be compared and loaded.

Wt
Is the 32-bit name of the general-purpose register to be conditionally stored.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Xs
Is the 64-bit name of the general-purpose register to be compared and loaded.

Xt
Is the 64-bit name of the general-purpose register to be conditionally stored.

Architectures supported

Supported in the Armv8.1 architecture and later.

Usage

Compare and Swap word or doubleword in memory reads a 32-bit word or 64-bit doubleword from memory, and compares it against the value held in a first register. If the comparison is equal, the value in a second register is written to memory. If the write is performed, the read and write occur atomically such that no other modification of the memory location can take place between the read and write.

- `CASA` and `CASAL` load from memory with acquire semantics.
- `CASL` and `CASAL` store to memory with release semantics.
- `CAS` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

The architecture permits that the data read clears any exclusive monitors associated with that location, even if the compare subsequently fails.

If the instruction generates a synchronous Data Abort, the register which is compared and loaded, that is `Ws`, or `Xs`, is restored to the value held in the register before the instruction was executed.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.3 CASAB, CASALB, CASB, CASLB

Compare and Swap byte in memory.

Syntax

```
CASAB Ws, Wt, [Xn|SP{,#0}] ; Acquire general registers

CASALB Ws, Wt, [Xn|SP{,#0}] ; Acquire and release general registers

CASB Ws, Wt, [Xn|SP{,#0}] ; No memory ordering general registers

CASLB Ws, Wt, [Xn|SP{,#0}] ; Release general registers
```

Where:

Ws
Is the 32-bit name of the general-purpose register to be compared and loaded.

Wt
Is the 32-bit name of the general-purpose register to be conditionally stored.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Architectures supported

Supported in the Armv8.1 architecture and later.

Usage

Compare and Swap byte in memory reads an 8-bit byte from memory, and compares it against the value held in a first register. If the comparison is equal, the value in a second register is written to memory. If the write is performed, the read and write occur atomically such that no other modification of the memory location can take place between the read and write.

- `CASAB` and `CASALB` load from memory with acquire semantics.
- `CASLB` and `CASALB` store to memory with release semantics.
- `CASB` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

The architecture permits that the data read clears any exclusive monitors associated with that location, even if the compare subsequently fails.

If the instruction generates a synchronous Data Abort, the register which is compared and loaded, that is `Ws`, is restored to the values held in the register before the instruction was executed.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.4 CASAH, CASALH, CASH, CASLH

Compare and Swap halfword in memory.

Syntax

```
CASAH Ws, Wt, [Xn|SP{,#0}] ; Acquire general registers

CASALH Ws, Wt, [Xn|SP{,#0}] ; Acquire and release general registers

CASH Ws, Wt, [Xn|SP{,#0}] ; No memory ordering general registers

CASLH Ws, Wt, [Xn|SP{,#0}] ; Release general registers
```

Where:

Ws
Is the 32-bit name of the general-purpose register to be compared and loaded.

Wt
Is the 32-bit name of the general-purpose register to be conditionally stored.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Architectures supported

Supported in the Armv8.1 architecture and later.

Usage

Compare and Swap halfword in memory reads a 16-bit halfword from memory, and compares it against the value held in a first register. If the comparison is equal, the value in a second register is written to memory. If the write is performed, the read and write occur atomically such that no other modification of the memory location can take place between the read and write.

- `CASAH` and `CASALH` load from memory with acquire semantics.
- `CASLH` and `CASALH` store to memory with release semantics.
- `CAS` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

The architecture permits that the data read clears any exclusive monitors associated with that location, even if the compare subsequently fails.

If the instruction generates a synchronous Data Abort, the register which is compared and loaded, that is `Ws`, is restored to the values held in the register before the instruction was executed.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.5 CASPA, CASPAL, CASP, CASPL, CASPAL, CASP, CASPL

Compare and Swap Pair of words or doublewords in memory.

Syntax

```
CASPA Ws, <W(s+1)>, Wt, <W(t+1)>, [Xn|SP{,#0}] ; 32-bit, acquire general registers

CASPAL Ws, <W(s+1)>, Wt, <W(t+1)>, [Xn|SP{,#0}] ; 32-bit, acquire and release general registers

CASP Ws, <W(s+1)>, Wt, <W(t+1)>, [Xn|SP{,#0}] ; 32-bit, no memory ordering general registers

CASPL Ws, <W(s+1)>, Wt, <W(t+1)>, [Xn|SP{,#0}] ; 32-bit, release general registers

CASPA Xs, <X(s+1)>, Xt, <X(t+1)>, [Xn|SP{,#0}] ; 64-bit, acquire general registers

CASPAL Xs, <X(s+1)>, Xt, <X(t+1)>, [Xn|SP{,#0}] ; 64-bit, acquire and release general registers

CASP Xs, <X(s+1)>, Xt, <X(t+1)>, [Xn|SP{,#0}] ; 64-bit, no memory ordering general registers

CASPL Xs, <X(s+1)>, Xt, <X(t+1)>, [Xn|SP{,#0}] ; 64-bit, release general registers
```

Where:

Ws
Is the 32-bit name of the first general-purpose register to be compared and loaded.

<W(s+1)>
Is the 32-bit name of the second general-purpose register to be compared and loaded.

Wt
Is the 32-bit name of the first general-purpose register to be conditionally stored.

<W(t+1)>
Is the 32-bit name of the second general-purpose register to be conditionally stored.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Xs
Is the 64-bit name of the first general-purpose register to be compared and loaded.

<X(s+1)>
Is the 64-bit name of the second general-purpose register to be compared and loaded.

Xt
Is the 64-bit name of the first general-purpose register to be conditionally stored.

<X(t+1)>
Is the 64-bit name of the second general-purpose register to be conditionally stored.

Architectures supported

Supported in the Armv8.1 architecture and later.

Usage

Compare and Swap Pair of words or doublewords in memory reads a pair of 32-bit words or 64-bit doublewords from memory, and compares them against the values held in the first pair of registers. If the comparison is equal, the values in the second pair of registers are written to memory. If the writes are performed, the reads and writes occur atomically such that no other modification of the memory location can take place between the reads and writes.

- `CASPA` and `CASPAL` load from memory with acquire semantics.
- `CASPL` and `CASPAL` store to memory with release semantics.
- `CAS` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

The architecture permits that the data read clears any exclusive monitors associated with that location, even if the compare subsequently fails.

If the instruction generates a synchronous Data Abort, the registers which are compared and loaded, that is *Ws* and `<W(s+1)>`, or *Xs* and `<X(s+1)>`, are restored to the values held in the registers before the instruction was executed.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.6 LDADDA, LDADDAL, LDADD, LDADDL, LDADDAL, LDADD, LDADDL

Atomic add on word or doubleword in memory.

Syntax

```
LDADDA Ws, Wt, [Xn|SP] ; 32-bit, acquire general registers

LDADDAL Ws, Wt, [Xn|SP] ; 32-bit, acquire and release general registers

LDADD Ws, Wt, [Xn|SP] ; 32-bit, no memory ordering general registers

LDADDL Ws, Wt, [Xn|SP] ; 32-bit, release general registers

LDADDA Xs, Xt, [Xn|SP] ; 64-bit, acquire general registers

LDADDAL Xs, Xt, [Xn|SP] ; 64-bit, acquire and release general registers

LDADD Xs, Xt, [Xn|SP] ; 64-bit, no memory ordering general registers

LDADDL Xs, Xt, [Xn|SP] ; 64-bit, release general registers
```

Where:

Ws
Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location.

Wt
Is the 32-bit name of the general-purpose register to be loaded.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Xs
Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location.

Xt
Is the 64-bit name of the general-purpose register to be loaded.

Architectures supported

Supported in the Armv8.1 architecture and later.

Usage

Atomic add on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, adds the value held in a register to it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of `WZR` or `XZR`, `LDADDA` and `LDADDAL` load from memory with acquire semantics.
- `LDADDL` and `LDADDAL` store to memory with release semantics.
- `LDADD` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.7 LDADDAB, LDADDALB, LDADDB, LDADDLB

Atomic add on byte in memory.

Syntax

```
LDADDAB Ws, Wt, [Xn|SP] ; Acquire general registers

LDADDALB Ws, Wt, [Xn|SP] ; Acquire and release general registers

LDADDB Ws, Wt, [Xn|SP] ; No memory ordering general registers

LDADDLB Ws, Wt, [Xn|SP] ; Release general registers
```

Where:

Ws
Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location.

Wt
Is the 32-bit name of the general-purpose register to be loaded.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Architectures supported

Supported in the Armv8.1 architecture and later.

Usage

Atomic add on byte in memory atomically loads an 8-bit byte from memory, adds the value held in a register to it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not `WZR`, `LDADDAB` and `LDADDALB` load from memory with acquire semantics.
- `LDADDLB` and `LDADDALB` store to memory with release semantics.
- `LDADDB` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.8 LDADDAH, LDADDALH, LDADDH, LDADDLH

Atomic add on halfword in memory.

Syntax

```
LDADDAH Ws, Wt, [Xn|SP] ; Acquire general registers

LDADDALH Ws, Wt, [Xn|SP] ; Acquire and release general registers

LDADDH Ws, Wt, [Xn|SP] ; No memory ordering general registers

LDADDLH Ws, Wt, [Xn|SP] ; Release general registers
```

Where:

Ws
Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location.

Wt
Is the 32-bit name of the general-purpose register to be loaded.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Architectures supported

Supported in the Armv8.1 architecture and later.

Usage

Atomic add on halfword in memory atomically loads a 16-bit halfword from memory, adds the value held in a register to it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not `WZR`, `LDADDAH` and `LDADDALH` load from memory with acquire semantics.
- `LDADDLH` and `LDADDALH` store to memory with release semantics.
- `LDADDH` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.9 LDAPR

Load-Acquire RCpc Register.

Syntax

```
LDAPR Wt, [Xn|SP {,#0}] ; 32-bit
LDAPR Xt, [Xn|SP {,#0}] ; 64-bit
```

Where:

Wt
Is the 32-bit name of the general-purpose register to be loaded.

Xt
Is the 64-bit name of the general-purpose register to be loaded.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Architectures supported

This instruction is supported in the Armv8.3-A architecture and later. It is optionally supported in the Armv8.2-A architecture with the RCpc extension.

Usage

Load-Acquire RCpc Register derives an address from a base register value, loads a 32-bit word or 64-bit doubleword from the derived address in memory, and writes it to a register.

The instruction has memory ordering semantics as described in *Load-Acquire, Store-Release* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*, except that:

- There is no ordering requirement, separate from the requirements of a Load-Acquirepc or a Store-Release, created by having a Store-Release followed by a Load-Acquirepc instruction.
- The reading of a value written by a Store-Release by a Load-Acquirepc instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

For information about memory accesses, see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.10 LDAPRB

Load-Acquire RCpc Register Byte.

Syntax

```
LDAPRB Wt, [Xn|SP {,#0}]
```

Where:

Wt
Is the 32-bit name of the general-purpose register to be loaded.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Architectures supported

This instruction is supported in the Armv8.3-A architecture and later. It is optionally supported in the Armv8.2-A architecture with the RCpc extension.

Usage

Load-Acquire RCpc Register Byte derives an address from a base register value, loads a byte from the derived address in memory, zero-extends it and writes it to a register.

The instruction has memory ordering semantics as described in *Load-Acquire, Store-Release* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*, except that:

- There is no ordering requirement, separate from the requirements of a Load-Acquirepc or a Store-Release, created by having a Store-Release followed by a Load-Acquirepc instruction.
- The reading of a value written by a Store-Release by a Load-Acquirepc instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

For information about memory accesses, see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.11 LDAPRH

Load-Acquire RCpc Register Halfword.

Syntax

```
LDAPRH Wt, [Xn|SP {,#0}]
```

Where:

Wt
Is the 32-bit name of the general-purpose register to be loaded.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Architectures supported

This instruction is supported in the Armv8.3-A architecture and later. It is optionally supported in Armv8.2-A architecture with the RCpc extension.

Usage

Load-Acquire RCpc Register Halfword derives an address from a base register value, loads a halfword from the derived address in memory, zero-extends it and writes it to a register.

The instruction has memory ordering semantics as described in *Load-Acquire, Store-Release* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*, except that:

- There is no ordering requirement, separate from the requirements of a Load-Acquirepc or a Store-Release, created by having a Store-Release followed by a Load-Acquirepc instruction.
- The reading of a value written by a Store-Release by a Load-Acquirepc instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

For information about memory accesses, see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.12 LDAR

Load-Acquire Register.

Syntax

```
LDAR Wt, [Xn|SP{,#0}] ; 32-bit
LDAR Xt, [Xn|SP{,#0}] ; 64-bit
```

Where:

Wt
Is the 32-bit name of the general-purpose register to be transferred.

Xt
Is the 64-bit name of the general-purpose register to be transferred.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Usage

Load-Acquire Register derives an address from a base register value, loads a 32-bit word or 64-bit doubleword from memory, and writes it to a register. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*. For information about memory accesses, see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.13 LDARB

Load-Acquire Register Byte.

Syntax

`LDARB Wt, [Xn|SP{,#0}]`

Where:

Wt
Is the 32-bit name of the general-purpose register to be transferred.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Usage

Load-Acquire Register Byte derives an address from a base register value, loads a byte from memory, zero-extends it and writes it to a register. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*. For information about memory accesses, see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.14 LDARH

Load-Acquire Register Halfword.

Syntax

`LDARH Wt, [Xn|SP{,#0}]`

Where:

Wt
Is the 32-bit name of the general-purpose register to be transferred.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Usage

Load-Acquire Register Halfword derives an address from a base register value, loads a halfword from memory, zero-extends it, and writes it to a register. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*. For information about memory accesses, see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.15 LDAXP

Load-Acquire Exclusive Pair of Registers.

Syntax

```
LDAXP Wt1, Wt2, [Xn|SP{,#0}] ; 32-bit
LDAXP Xt1, Xt2, [Xn|SP{,#0}] ; 64-bit
```

Where:

Wt1
Is the 32-bit name of the first general-purpose register to be transferred.

Wt2
Is the 32-bit name of the second general-purpose register to be transferred.

Xt1
Is the 64-bit name of the first general-purpose register to be transferred.

Xt2
Is the 64-bit name of the second general-purpose register to be transferred.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Usage

Load-Acquire Exclusive Pair of Registers derives an address from a base register value, loads two 32-bit words or two 64-bit doublewords from memory, and writes them to two registers. A 32-bit pair requires the address to be doubleword aligned and is single-copy atomic at doubleword granularity. A 64-bit pair requires the address to be quadword aligned and is single-copy atomic for each doubleword at doubleword granularity. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See *Synchronization and semaphores* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*. For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Note

For 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*, and particularly *LDAXP*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.16 LDAXR

Load-Acquire Exclusive Register.

Syntax

```
LDAXR Wt, [Xn|SP{,#0}] ; 32-bit
LDAXR Xt, [Xn|SP{,#0}] ; 64-bit
```

Where:

Wt
Is the 32-bit name of the general-purpose register to be transferred.

Xt
Is the 64-bit name of the general-purpose register to be transferred.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Usage

Load-Acquire Exclusive Register derives an address from a base register value, loads a 32-bit word or 64-bit doubleword from memory, and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See *Synchronization and semaphores* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*. For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.17 LDAXRB

Load-Acquire Exclusive Register Byte.

Syntax

```
LDAXRB Wt, [Xn|SP{,#0}]
```

Where:

Wt
Is the 32-bit name of the general-purpose register to be transferred.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Usage

Load-Acquire Exclusive Register Byte derives an address from a base register value, loads a byte from memory, zero-extends it and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See *Synchronization and semaphores* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*. For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.18 LDAXRH

Load-Acquire Exclusive Register Halfword.

Syntax

```
LDAXRH Wt, [Xn|SP{,#0}]
```

Where:

Wt
Is the 32-bit name of the general-purpose register to be transferred.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Usage

Load-Acquire Exclusive Register Halfword derives an address from a base register value, loads a halfword from memory, zero-extends it and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See *Synchronization and semaphores* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*. For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.19 LDCLRA, LDCLRAL, LDCLR, LDCLRL, LDCLRAL, LDCLR, LDCLRL

Atomic bit clear on word or doubleword in memory.

Syntax

```
LDCLRA Ws, Wt, [Xn|SP] ; 32-bit, acquire general registers

LDCLRAL Ws, Wt, [Xn|SP] ; 32-bit, acquire and release general registers

LDCLR Ws, Wt, [Xn|SP] ; 32-bit, no memory ordering general registers

LDCLRL Ws, Wt, [Xn|SP] ; 32-bit, release general registers

LDCLRA Xs, Xt, [Xn|SP] ; 64-bit, acquire general registers

LDCLRAL Xs, Xt, [Xn|SP] ; 64-bit, acquire and release general registers

LDCLR Xs, Xt, [Xn|SP] ; 64-bit, no memory ordering general registers

LDCLRL Xs, Xt, [Xn|SP] ; 64-bit, release general registers
```

Where:

Ws
Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location.

Wt
Is the 32-bit name of the general-purpose register to be loaded.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Xs
Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location.

Xt
Is the 64-bit name of the general-purpose register to be loaded.

Architectures supported

Supported in the Armv8.1 architecture and later.

Usage

Atomic bit clear on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of `WZR` or `XZR`, `LDCLRA` and `LDCLRAL` load from memory with acquire semantics.
- `LDCLRL` and `LDCLRAL` store to memory with release semantics.
- `LDCLR` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.20 LDCLRAB, LDCLRALB, LDCLRB, LDCLRLB

Atomic bit clear on byte in memory.

Syntax

```
LDCLRAB Ws, Wt, [Xn|SP] ; Acquire general registers

LDCLRALB Ws, Wt, [Xn|SP] ; Acquire and release general registers

LDCLRB Ws, Wt, [Xn|SP] ; No memory ordering general registers

LDCLRLB Ws, Wt, [Xn|SP] ; Release general registers
```

Where:

Ws
Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location.

Wt
Is the 32-bit name of the general-purpose register to be loaded.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Architectures supported

Supported in the Armv8.1 architecture and later.

Usage

Atomic bit clear on byte in memory atomically loads an 8-bit byte from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not `WZR`, `LDCLRAB` and `LDCLRALB` load from memory with acquire semantics.
- `LDCLRLB` and `LDCLRALB` store to memory with release semantics.
- `LDCLRB` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.21 LDCLRAH, LDCLRALH, LDCLRH, LDCLRLH

Atomic bit clear on halfword in memory.

Syntax

```
LDCLRAH Ws, Wt, [Xn|SP] ; Acquire general registers

LDCLRALH Ws, Wt, [Xn|SP] ; Acquire and release general registers

LDCLRH Ws, Wt, [Xn|SP] ; No memory ordering general registers

LDCLRLH Ws, Wt, [Xn|SP] ; Release general registers
```

Where:

Ws
Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location.

Wt
Is the 32-bit name of the general-purpose register to be loaded.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Architectures supported

Supported in the Armv8.1 architecture and later.

Usage

Atomic bit clear on halfword in memory atomically loads a 16-bit halfword from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not `WZR`, `LDCLRAH` and `LDCLRALH` load from memory with acquire semantics.
- `LDCLRLH` and `LDCLRALH` store to memory with release semantics.
- `LDCLRH` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.22 LDEORA, LDEORAL, LDEOR, LDEORL, LDEORAL, LDEOR, LDEORL

Atomic exclusive OR on word or doubleword in memory.

Syntax

```
LDEORA Ws, Wt, [Xn|SP] ; 32-bit, acquire general registers

LDEORAL Ws, Wt, [Xn|SP] ; 32-bit, acquire and release general registers

LDEOR Ws, Wt, [Xn|SP] ; 32-bit, no memory ordering general registers

LDEORL Ws, Wt, [Xn|SP] ; 32-bit, release general registers

LDEORA Xs, Xt, [Xn|SP] ; 64-bit, acquire general registers

LDEORAL Xs, Xt, [Xn|SP] ; 64-bit, acquire and release general registers

LDEOR Xs, Xt, [Xn|SP] ; 64-bit, no memory ordering general registers

LDEORL Xs, Xt, [Xn|SP] ; 64-bit, release general registers
```

Where:

Ws
Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location.

Wt
Is the 32-bit name of the general-purpose register to be loaded.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Xs
Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location.

Xt
Is the 64-bit name of the general-purpose register to be loaded.

Architectures supported

Supported in the Armv8.1 architecture and later.

Usage

Atomic exclusive OR on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, performs an exclusive OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of `WZR` or `XZR`, `LDEORA` and `LDEORAL` load from memory with acquire semantics.
- `LDEORL` and `LDEORAL` store to memory with release semantics.
- `LDEOR` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.23 LDEORAB, LDEORALB, LDEORB, LDEORLB

Atomic exclusive OR on byte in memory.

Syntax

```
LDEORAB Ws, Wt, [Xn|SP] ; Acquire general registers

LDEORALB Ws, Wt, [Xn|SP] ; Acquire and release general registers

LDEORB Ws, Wt, [Xn|SP] ; No memory ordering general registers

LDEORLB Ws, Wt, [Xn|SP] ; Release general registers
```

Where:

Ws
Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location.

Wt
Is the 32-bit name of the general-purpose register to be loaded.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Architectures supported

Supported in the Armv8.1 architecture and later.

Usage

Atomic exclusive OR on byte in memory atomically loads an 8-bit byte from memory, performs an exclusive OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not `WZR`, `LDEORAB` and `LDEORALB` load from memory with acquire semantics.
- `LDEORLB` and `LDEORALB` store to memory with release semantics.
- `LDEORB` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.24 LDEORAH, LDEORALH, LDEORH, LDEORLH

Atomic exclusive OR on halfword in memory.

Syntax

```
LDEORAH Ws, Wt, [Xn|SP] ; Acquire general registers

LDEORALH Ws, Wt, [Xn|SP] ; Acquire and release general registers

LDEORH Ws, Wt, [Xn|SP] ; No memory ordering general registers

LDEORLH Ws, Wt, [Xn|SP] ; Release general registers
```

Where:

Ws
Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location.

Wt
Is the 32-bit name of the general-purpose register to be loaded.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Architectures supported

Supported in the Armv8.1 architecture and later.

Usage

Atomic exclusive OR on halfword in memory atomically loads a 16-bit halfword from memory, performs an exclusive OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not `WZR`, `LDEORAH` and `LDEORALH` load from memory with acquire semantics.
- `LDEORLH` and `LDEORALH` store to memory with release semantics.
- `LDEORH` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.25 LDLAR

Load LOAcquire Register.

Syntax

```
LDLAR Wt, [Xn|SP{,#0}] ; 32-bit

LDLAR Xt, [Xn|SP{,#0}] ; 64-bit
```

Where:

Wt
Is the 32-bit name of the general-purpose register to be transferred.

Xt
Is the 64-bit name of the general-purpose register to be transferred.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Architectures supported

Supported in the Armv8.1 architecture and later.

Usage

Load LOAcquire Register loads a 32-bit word or 64-bit doubleword from memory, and writes it to a register. The instruction also has memory ordering semantics as described in *Load LOAcquire, Store LORelease* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*. For information about memory accesses, see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.26 LDLARB

Load LOAcquire Register Byte.

Syntax

`LDLARB Wt, [Xn|SP{,#0}]`

Where:

Wt
Is the 32-bit name of the general-purpose register to be transferred.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Architectures supported

Supported in the Armv8.1 architecture and later.

Usage

Load LOAcquire Register Byte loads a byte from memory, zero-extends it and writes it to a register. The instruction also has memory ordering semantics as described in *Load LOAcquire, Store LORelease* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*. For information about memory accesses, see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.27 LDLARH

Load LOAcquire Register Halfword.

Syntax

`LDLARH Wt, [Xn|SP{,#0}]`

Where:

Wt
Is the 32-bit name of the general-purpose register to be transferred.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Architectures supported

Supported in the Armv8.1 architecture and later.

Usage

Load LOAcquire Register Halfword loads a halfword from memory, zero-extends it, and writes it to a register. The instruction also has memory ordering semantics as described in *Load LOAcquire, Store LORelease* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*. For information about memory accesses, see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.28 LDNP

Load Pair of Registers, with non-temporal hint.

Syntax

```
LDNP Wt1, Wt2, [Xn|SP{, #imm}] ; 32-bit, Signed offset

LDNP Xt1, Xt2, [Xn|SP{, #imm}] ; 64-bit, Signed offset
```

Where:

Wt1
Is the 32-bit name of the first general-purpose register to be transferred.

Wt2
Is the 32-bit name of the second general-purpose register to be transferred.

imm
Depends on the instruction variant:

32-bit general registers
Is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0.

64-bit general registers
Is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0.

Xt1
Is the 64-bit name of the first general-purpose register to be transferred.

Xt2
Is the 64-bit name of the second general-purpose register to be transferred.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Usage

Load Pair of Registers, with non-temporal hint, calculates an address from a base register value and an immediate offset, loads two 32-bit words or two 64-bit doublewords from memory, and writes them to two registers.

For information about memory accesses, see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*. For information about Non-temporal pair instructions, see *Load/Store Non-temporal pair* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Note

For 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*, and particularly *LDNP*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.29 LDP

Load Pair of Registers.

Syntax

```
LDP Wt1, Wt2, [Xn|SP], #imm ; 32-bit, Post-index

LDP Xt1, Xt2, [Xn|SP], #imm ; 64-bit, Post-index

LDP Wt1, Wt2, [Xn|SP, #imm]! ; 32-bit, Pre-index

LDP Xt1, Xt2, [Xn|SP, #imm]! ; 64-bit, Pre-index

LDP Wt1, Wt2, [Xn|SP{, #imm}] ; 32-bit, Signed offset

LDP Xt1, Xt2, [Xn|SP{, #imm}] ; 64-bit, Signed offset
```

Where:

Wt1
Is the 32-bit name of the first general-purpose register to be transferred.

Wt2
Is the 32-bit name of the second general-purpose register to be transferred.

imm
Depends on the instruction variant:

32-bit general registers
Is the signed immediate byte offset, a multiple of 4 in the range -256 to 252.

64-bit general registers
Is the signed immediate byte offset, a multiple of 8 in the range -512 to 504.

Xt1
Is the 64-bit name of the first general-purpose register to be transferred.

Xt2
Is the 64-bit name of the second general-purpose register to be transferred.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Usage

Load Pair of Registers calculates an address from a base register value and an immediate offset, loads two 32-bit words or two 64-bit doublewords from memory, and writes them to two registers. For information about memory accesses, see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Note

For 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*, and particularly *LDP*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.30 LDPSW

Load Pair of Registers Signed Word.

Syntax

```
LDPSW Xt1, Xt2, [Xn|SP], #imm ; Post-index general registers

LDPSW Xt1, Xt2, [Xn|SP, #imm]! ; Pre-index general registers

LDPSW Xt1, Xt2, [Xn|SP{, #imm}] ; Signed offset general registers
```

Where:

imm

Depends on the instruction variant:

Post-index and Pre-index general registers

Is the signed immediate byte offset, a multiple of 4 in the range -256 to 252.

Signed offset general registers

Is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0.

Xt1

Is the 64-bit name of the first general-purpose register to be transferred.

Xt2

Is the 64-bit name of the second general-purpose register to be transferred.

Xn|SP

Is the 64-bit name of the general-purpose base register or stack pointer.

Usage

Load Pair of Registers Signed Word calculates an address from a base register value and an immediate offset, loads two 32-bit words from memory, sign-extends them, and writes them to two registers. For information about memory accesses, see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Note

For 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*, and particularly *LDPSW*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.31 LDR (immediate)

Load Register (immediate).

Syntax

```
LDR Wt, [Xn|SP], #simm ; 32-bit, Post-index

LDR Xt, [Xn|SP], #simm ; 64-bit, Post-index

LDR Wt, [Xn|SP, #simm]! ; 32-bit, Pre-index

LDR Xt, [Xn|SP, #simm]! ; 64-bit, Pre-index

LDR Wt, [Xn|SP{, #pimm}] ; 32-bit

LDR Xt, [Xn|SP{, #pimm}] ; 64-bit
```

Where:

Wt
Is the 32-bit name of the general-purpose register to be transferred.

simm
Is the signed immediate byte offset, in the range -256 to 255.

Xt
Is the 64-bit name of the general-purpose register to be transferred.

pimm
Depends on the instruction variant:

32-bit general registers
Is the optional positive immediate byte offset, a multiple of 4 in the range 0 to 16380, defaulting to 0.

64-bit general registers
Is the optional positive immediate byte offset, a multiple of 8 in the range 0 to 32760, defaulting to 0.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Usage

Load Register (immediate) loads a word or doubleword from memory and writes it to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about memory accesses, see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*. The Unsigned offset variant scales the immediate offset value by the size of the value accessed before adding it to the base register value.

Note

For 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*, and particularly *LDR (immediate)*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.32 LDR (literal)

Load Register (literal).

Syntax

```
LDR Wt, label ; 32-bit
LDR Xt, label ; 64-bit
```

Where:

Wt
Is the 32-bit name of the general-purpose register to be loaded.

Xt
Is the 64-bit name of the general-purpose register to be loaded.

label
Is the program label from which the data is to be loaded. Its offset from the address of this instruction, in the range ±1MB.

Usage

Load Register (literal) calculates an address from the PC value and an immediate offset, loads a word from memory, and writes it to a register. For information about memory accesses, see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.33 LDR pseudo-instruction

Load a register with either a 32-bit or 64-bit immediate value or an address.

Note

This description is for the `LDR` pseudo-instruction only, and not for the `LDR` instruction.

Syntax

`LDR` *`Wd, =expr`*

`LDR` *`Xd, =expr`*

`LDR` *`Wd, =label_expr`*

`LDR` *`Xd, =label_expr`*

where:

`Wd`
Is the register to load with a 32-bit value.

`Xd`
Is the register to load with a 64-bit value.

`expr`
Evaluates to a numeric value.

`label_expr`
Is a PC-relative or external expression of an address in the form of a label plus or minus a numeric value.

Usage

When using the `LDR` pseudo-instruction, the assembler places the value of *`expr`* or *`label_expr`* in a literal pool and generates a PC-relative `LDR` instruction that reads the constant from the literal pool.

Note

- An address loaded in this way is fixed at link time, so the code is not position-independent.
- The address holding the constant remains valid regardless of where the linker places the ELF section containing the `LDR` instruction.

If *`label_expr`* is an external expression, or is not contained in the current section, the assembler places a linker relocation directive in the object file. The linker generates the address at link time.

If *`label_expr`* is a local label, the assembler places a linker relocation directive in the object file and generates a symbol for that local label. The address is generated at link time.

The offset from the PC to the value in the literal pool must be less than ±1MB . You are responsible for ensuring that there is a literal pool within range.

Examples

```
       LDR     w1,=0xfff    ; loads 0xfff into W1
                            ; =>  LDR w1,[pc,offset_to_litpool]
                            ;     ...
                            ;     litpool DCD 4095

       LDR     x2,=place    ; loads the address of
                            ; place into X2
                            ; =>  LDR x2,[pc,offset_to_litpool]
                            ;     ...
                            ;     litpool DCQ place
```

Related concepts

12.3 Numeric constants on page 12-300.

12.5 Register-relative and PC-relative expressions on page 12-302.

12.10 Numeric local labels on page 12-307.

6.7 Load immediate values using LDR Rd, =const on page 6-110.

Related information

Armv8-A Architecture Reference Manual.

17.34 LDR (register)

Load Register (register).

Syntax

```
LDR Wt, [Xn|SP, (Wm|Xm){, extend {amount}}] ; 32-bit
LDR Xt, [Xn|SP, (Wm|Xm){, extend {amount}}] ; 64-bit
```

Where:

Wt
Is the 32-bit name of the general-purpose register to be transferred.

amount
Is the index shift amount, optional only when *extend* is not LSL. Where it is permitted to be optional, it defaults to #0. It is:

32-bit general registers
Can be one of `#0` or `#2`.

64-bit general registers
Can be one of `#0` or `#3`.

Xt
Is the 64-bit name of the general-purpose register to be transferred.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Wm
When "option<0>" is set to 0, is the 32-bit name of the general-purpose index register.

Xm
When "option<0>" is set to 1, is the 64-bit name of the general-purpose index register.

extend
Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when *amount* is omitted, and can be one of the values shown in Usage.

Usage

Load Register (register) calculates an address from a base register value and an offset register value, loads a word from memory, and writes it to a register. The offset register value can optionally be shifted and extended. For information about memory accesses, see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.35 LDRAA, LDRAB, LDRAB

Load Register, with pointer authentication.

Syntax

```
LDRAA Xt, [Xn|SP{, #simm}] ; LDRAA

LDRAA Xt, [Xn|SP{, #simm}]! ; LDRAA

LDRAB Xt, [Xn|SP{, #simm}] ; LDRAB

LDRAB Xt, [Xn|SP{, #simm}]! ; LDRAB
```

Where:

Xt
Is the 64-bit name of the general-purpose register to be transferred.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

simm
Is the optional signed immediate byte offset, in the range -512 to 511, defaulting to 0.

Architectures supported

Supported in the Armv8.2-A architecture and later.

Usage

Load Register, with pointer authentication. This instruction authenticates an address from a base register using a modifier of zero and the specified key, adds an immediate offset to the authenticated address, and loads a 64-bit doubleword from memory at this resulting address into a register.

Key A is used for `LDRAA`, and key B is used for `LDRAB`.

If the authentication passes, the PE behaves the same as for an `LDR` instruction. If the authentication fails, a Translation fault is generated.

The authenticated address is not written back to the base register, unless the pre-indexed variant of the instruction is used. In this case, the address that is written back to the base register does not include the pointer authentication code.

For information about memory accesses, see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.36 LDRB (immediate)

Load Register Byte (immediate).

Syntax

```
LDRB Wt, [Xn|SP], #simm ; Post-index general registers

LDRB Wt, [Xn|SP, #simm]! ; Pre-index general registers

LDRB Wt, [Xn|SP{, #pimm}] ; Unsigned offset general registers
```

Where:

simm
Is the signed immediate byte offset, in the range -256 to 255.

pimm
Is the optional positive immediate byte offset, in the range 0 to 4095, defaulting to 0.

Wt
Is the 32-bit name of the general-purpose register to be transferred.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Usage

Load Register Byte (immediate) loads a byte from memory, zero-extends it, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about memory accesses, see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Note

For 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*, and particularly *LDRB (immediate)*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.37 LDRB (register)

Load Register Byte (register).

Syntax

```
LDRB Wt, [Xn|SP, (Wm|Xm), extend {amount}] ; Extended register general registers

LDRB Wt, [Xn|SP, Xm{, LSL amount}] ; Shifted register general registers
```

Where:

Wt
Is the 32-bit name of the general-purpose register to be transferred.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Wm
When "option<0>" is set to 0, is the 32-bit name of the general-purpose index register.

Xm
When "option<0>" is set to 1, is the 64-bit name of the general-purpose index register.

extend
Is the index extend specifier, and can be one of the values shown in Usage.

amount
Is the index shift amount, it must be.

Usage

Load Register Byte (register) calculates an address from a base register value and an offset register value, loads a byte from memory, zero-extends it, and writes it to a register. For information about memory accesses, see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.38 LDRH (immediate)

Load Register Halfword (immediate).

Syntax

```
LDRH Wt, [Xn|SP], #simm ; Post-index general registers

LDRH Wt, [Xn|SP, #simm]! ; Pre-index general registers

LDRH Wt, [Xn|SP{, #pimm}] ; Unsigned offset general registers
```

Where:

simm
Is the signed immediate byte offset, in the range -256 to 255.

pimm
Is the optional positive immediate byte offset, a multiple of 2 in the range 0 to 8190, defaulting to 0.

Wt
Is the 32-bit name of the general-purpose register to be transferred.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Usage

Load Register Halfword (immediate) loads a halfword from memory, zero-extends it, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about memory accesses, see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Note

For 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*, and particularly *LDRH (immediate)*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.39 LDRH (register)

Load Register Halfword (register).

Syntax

```
LDRH Wt, [Xn|SP, (Wm|Xm){, extend {amount}}]
```

Where:

Wt
Is the 32-bit name of the general-purpose register to be transferred.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Wm
When "option<0>" is set to 0, is the 32-bit name of the general-purpose index register.

Xm
When "option<0>" is set to 1, is the 64-bit name of the general-purpose index register.

extend
Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when *amount* is omitted, and can be one of `UXTW`, `LSL`, `SXTW` or `SXTX`.

amount
Is the index shift amount, optional only when *extend* is not LSL. Where it is permitted to be optional, it defaults to #0. It is, and can be either `#0` or `#1`.

Usage

Load Register Halfword (register) calculates an address from a base register value and an offset register value, loads a halfword from memory, zero-extends it, and writes it to a register. For information about memory accesses, see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Note

For 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*, and particularly *LDRH (register)*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.40 LDRSB (immediate)

Load Register Signed Byte (immediate).

Syntax

```
LDRSB Wt, [Xn|SP], #simm ; 32-bit, Post-index

LDRSB Xt, [Xn|SP], #simm ; 64-bit, Post-index

LDRSB Wt, [Xn|SP, #simm]! ; 32-bit, Pre-index

LDRSB Xt, [Xn|SP, #simm]! ; 64-bit, Pre-index

LDRSB Wt, [Xn|SP{, #pimm}] ; 32-bit

LDRSB Xt, [Xn|SP{, #pimm}] ; 64-bit
```

Where:

Wt
Is the 32-bit name of the general-purpose register to be transferred.

simm
Is the signed immediate byte offset, in the range -256 to 255.

Xt
Is the 64-bit name of the general-purpose register to be transferred.

pimm
Is the optional positive immediate byte offset, in the range 0 to 4095, defaulting to 0.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Usage

Load Register Signed Byte (immediate) loads a byte from memory, sign-extends it to either 32 bits or 64 bits, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about memory accesses, see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Note

For 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*, and particularly *LDRSB (immediate)*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.41 LDRSB (register)

Load Register Signed Byte (register).

Syntax

```
LDRSB Wt, [Xn|SP, (Wm|Xm), extend {amount}] ; 32-bit with extended register offset general registers

LDRSB Wt, [Xn|SP, Xm{, LSL amount}] ; 32-bit with shifted register offset general registers

LDRSB Xt, [Xn|SP, (Wm|Xm), extend {amount}] ; 64-bit with extended register offset general registers

LDRSB Xt, [Xn|SP, Xm{, LSL amount}] ; 64-bit with shifted register offset general registers
```

Where:

Wt
Is the 32-bit name of the general-purpose register to be transferred.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Wm
When "option<0>" is set to 0, is the 32-bit name of the general-purpose index register.

Xm
When "option<0>" is set to 1, is the 64-bit name of the general-purpose index register.

extend
Is the index extend specifier:

Can be one of `UXTW`, `SXTW` or `SXTX`.

amount
Is the index shift amount, it must be.

Xt
Is the 64-bit name of the general-purpose register to be transferred.

Usage

Load Register Signed Byte (register) calculates an address from a base register value and an offset register value, loads a byte from memory, sign-extends it, and writes it to a register. For information about memory accesses, see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.42 LDRSH (immediate)

Load Register Signed Halfword (immediate).

Syntax

```
LDRSH Wt, [Xn|SP], #simm ; 32-bit, Post-index

LDRSH Xt, [Xn|SP], #simm ; 64-bit, Post-index

LDRSH Wt, [Xn|SP, #simm]! ; 32-bit, Pre-index

LDRSH Xt, [Xn|SP, #simm]! ; 64-bit, Pre-index

LDRSH Wt, [Xn|SP{, #pimm}] ; 32-bit

LDRSH Xt, [Xn|SP{, #pimm}] ; 64-bit
```

Where:

Wt
Is the 32-bit name of the general-purpose register to be transferred.

simm
Is the signed immediate byte offset, in the range -256 to 255.

Xt
Is the 64-bit name of the general-purpose register to be transferred.

pimm
Is the optional positive immediate byte offset, a multiple of 2 in the range 0 to 8190, defaulting to 0.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Usage

Load Register Signed Halfword (immediate) loads a halfword from memory, sign-extends it to 32 bits or 64 bits, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about memory accesses, see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Note

For 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*, and particularly *LDRSH (immediate)*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.43 LDRSH (register)

Load Register Signed Halfword (register).

Syntax

```
LDRSH Wt, [Xn|SP, (Wm|Xm){, extend {amount}}] ; 32-bit
LDRSH Xt, [Xn|SP, (Wm|Xm){, extend {amount}}] ; 64-bit
```

Where:

Wt
Is the 32-bit name of the general-purpose register to be transferred.

Xt
Is the 64-bit name of the general-purpose register to be transferred.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Wm
When "option<0>" is set to 0, is the 32-bit name of the general-purpose index register.

Xm
When "option<0>" is set to 1, is the 64-bit name of the general-purpose index register.

extend
Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when *amount* is omitted, and can be one of the values shown in Usage.

amount
Is the index shift amount, optional only when *extend* is not LSL. Where it is permitted to be optional, it defaults to #0. It is, and can be either `#0` or `#1`.

Usage

Load Register Signed Halfword (register) calculates an address from a base register value and an offset register value, loads a halfword from memory, sign-extends it, and writes it to a register. For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.44 LDRSW (immediate)

Load Register Signed Word (immediate).

Syntax

```
LDRSW Xt, [Xn|SP], #simm ; Post-index general registers

LDRSW Xt, [Xn|SP, #simm]! ; Pre-index general registers

LDRSW Xt, [Xn|SP{, #pimm}] ; Unsigned offset general registers
```

Where:

simm
Is the signed immediate byte offset, in the range -256 to 255.

pimm
Is the optional positive immediate byte offset, a multiple of 4 in the range 0 to 16380, defaulting to 0.

Xt
Is the 64-bit name of the general-purpose register to be transferred.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Usage

Load Register Signed Word (immediate) loads a word from memory, sign-extends it to 64 bits, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about memory accesses, see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Note

For 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*, and particularly *LDRSW (immediate)*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.45 LDRSW (literal)

Load Register Signed Word (literal).

Syntax

`LDRSW Xt, label`

Where:

Xt
Is the 64-bit name of the general-purpose register to be loaded.

label
Is the program label from which the data is to be loaded. Its offset from the address of this instruction, in the range ±1MB.

Usage

Load Register Signed Word (literal) calculates an address from the PC value and an immediate offset, loads a word from memory, and writes it to a register. For information about memory accesses, see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.46 LDRSW (register)

Load Register Signed Word (register).

Syntax

`LDRSW Xt, [Xn|SP, (Wm|Xm){, extend {amount}}]`

Where:

Xt
Is the 64-bit name of the general-purpose register to be transferred.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Wm
When "option<0>" is set to 0, is the 32-bit name of the general-purpose index register.

Xm
When "option<0>" is set to 1, is the 64-bit name of the general-purpose index register.

extend
Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when *amount* is omitted, and can be one of `UXTW`, `LSL`, `SXTW` or `SXTX`.

amount
Is the index shift amount, optional only when *extend* is not LSL. Where it is permitted to be optional, it defaults to #0. It is, and can be either `#0` or `#2`.

Usage

Load Register Signed Word (register) calculates an address from a base register value and an offset register value, loads a word from memory, sign-extends it to form a 64-bit value, and writes it to a register. The offset register value can be shifted left by 0 or 2 bits. For information about memory accesses, see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.47 LDSETA, LDSETAL, LDSET, LDSETL, LDSETAL, LDSET, LDSETL

Atomic bit set on word or doubleword in memory.

Syntax

```
LDSETA Ws, Wt, [Xn|SP] ; 32-bit, acquire general registers

LDSETAL Ws, Wt, [Xn|SP] ; 32-bit, acquire and release general registers

LDSET Ws, Wt, [Xn|SP] ; 32-bit, no memory ordering general registers

LDSETL Ws, Wt, [Xn|SP] ; 32-bit, release general registers

LDSETA Xs, Xt, [Xn|SP] ; 64-bit, acquire general registers

LDSETAL Xs, Xt, [Xn|SP] ; 64-bit, acquire and release general registers

LDSET Xs, Xt, [Xn|SP] ; 64-bit, no memory ordering general registers

LDSETL Xs, Xt, [Xn|SP] ; 64-bit, release general registers
```

Where:

Ws
Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location.

Wt
Is the 32-bit name of the general-purpose register to be loaded.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Xs
Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location.

Xt
Is the 64-bit name of the general-purpose register to be loaded.

Architectures supported

Supported in the Armv8.1 architecture and later.

Usage

Atomic bit set on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of `WZR` or `XZR`, `LDSETA` and `LDSETAL` load from memory with acquire semantics.
- `LDSETL` and `LDSETAL` store to memory with release semantics.
- `LDSET` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.48 LDSETAB, LDSETALB, LDSETB, LDSETLB

Atomic bit set on byte in memory.

Syntax

```
LDSETAB Ws, Wt, [Xn|SP] ; Acquire general registers

LDSETALB Ws, Wt, [Xn|SP] ; Acquire and release general registers

LDSETB Ws, Wt, [Xn|SP] ; No memory ordering general registers

LDSETLB Ws, Wt, [Xn|SP] ; Release general registers
```

Where:

Ws
Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location.

Wt
Is the 32-bit name of the general-purpose register to be loaded.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Architectures supported

Supported in the Armv8.1 architecture and later.

Usage

Atomic bit set on byte in memory atomically loads an 8-bit byte from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not `WZR`, `LDSETAB` and `LDSETALB` load from memory with acquire semantics.
- `LDSETLB` and `LDSETALB` store to memory with release semantics.
- `LDSETB` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.49 LDSETAH, LDSETALH, LDSETH, LDSETLH

Atomic bit set on halfword in memory.

Syntax

```
LDSETAH Ws, Wt, [Xn|SP] ; Acquire general registers

LDSETALH Ws, Wt, [Xn|SP] ; Acquire and release general registers

LDSETH Ws, Wt, [Xn|SP] ; No memory ordering general registers

LDSETLH Ws, Wt, [Xn|SP] ; Release general registers
```

Where:

Ws
Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location.

Wt
Is the 32-bit name of the general-purpose register to be loaded.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Architectures supported

Supported in the Armv8.1 architecture and later.

Usage

Atomic bit set on halfword in memory atomically loads a 16-bit halfword from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not `WZR`, `LDSETAH` and `LDSETALH` load from memory with acquire semantics.
- `LDSETLH` and `LDSETALH` store to memory with release semantics.
- `LDSETH` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.50 LDSMAXA, LDSMAXAL, LDSMAX, LDSMAXL, LDSMAXAL, LDSMAX, LDSMAXL

Atomic signed maximum on word or doubleword in memory.

Syntax

```
LDSMAXA Ws, Wt, [Xn|SP] ; 32-bit, acquire general registers

LDSMAXAL Ws, Wt, [Xn|SP] ; 32-bit, acquire and release general registers

LDSMAX Ws, Wt, [Xn|SP] ; 32-bit, no memory ordering general registers

LDSMAXL Ws, Wt, [Xn|SP] ; 32-bit, release general registers

LDSMAXA Xs, Xt, [Xn|SP] ; 64-bit, acquire general registers

LDSMAXAL Xs, Xt, [Xn|SP] ; 64-bit, acquire and release general registers

LDSMAX Xs, Xt, [Xn|SP] ; 64-bit, no memory ordering general registers

LDSMAXL Xs, Xt, [Xn|SP] ; 64-bit, release general registers
```

Where:

Ws
Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location.

Wt
Is the 32-bit name of the general-purpose register to be loaded.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Xs
Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location.

Xt
Is the 64-bit name of the general-purpose register to be loaded.

Architectures supported

Supported in the Armv8.1 architecture and later.

Usage

Atomic signed maximum on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of `WZR` or `XZR`, `LDSMAXA` and `LDSMAXAL` load from memory with acquire semantics.
- `LDSMAXL` and `LDSMAXAL` store to memory with release semantics.
- `LDSMAX` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.51 LDSMAXAB, LDSMAXALB, LDSMAXB, LDSMAXLB

Atomic signed maximum on byte in memory.

Syntax

```
LDSMAXAB Ws, Wt, [Xn|SP] ; Acquire general registers

LDSMAXALB Ws, Wt, [Xn|SP] ; Acquire and release general registers

LDSMAXB Ws, Wt, [Xn|SP] ; No memory ordering general registers

LDSMAXLB Ws, Wt, [Xn|SP] ; Release general registers
```

Where:

Ws
Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location.

Wt
Is the 32-bit name of the general-purpose register to be loaded.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Architectures supported

Supported in the Armv8.1 architecture and later.

Usage

Atomic signed maximum on byte in memory atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not `WZR`, `LDSMAXAB` and `LDSMAXALB` load from memory with acquire semantics.
- `LDSMAXLB` and `LDSMAXALB` store to memory with release semantics.
- `LDSMAXB` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.52 LDSMAXAH, LDSMAXALH, LDSMAXH, LDSMAXLH

Atomic signed maximum on halfword in memory.

Syntax

```
LDSMAXAH Ws, Wt, [Xn|SP] ; Acquire general registers

LDSMAXALH Ws, Wt, [Xn|SP] ; Acquire and release general registers

LDSMAXH Ws, Wt, [Xn|SP] ; No memory ordering general registers

LDSMAXLH Ws, Wt, [Xn|SP] ; Release general registers
```

Where:

Ws
Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location.

Wt
Is the 32-bit name of the general-purpose register to be loaded.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Architectures supported

Supported in the Armv8.1 architecture and later.

Usage

Atomic signed maximum on halfword in memory atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not `WZR`, `LDSMAXAH` and `LDSMAXALH` load from memory with acquire semantics.
- `LDSMAXLH` and `LDSMAXALH` store to memory with release semantics.
- `LDSMAXH` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.53 LDSMINA, LDSMINAL, LDSMIN, LDSMINL, LDSMINAL, LDSMIN, LDSMINL

Atomic signed minimum on word or doubleword in memory.

Syntax

```
LDSMINA Ws, Wt, [Xn|SP] ; 32-bit, acquire general registers

LDSMINAL Ws, Wt, [Xn|SP] ; 32-bit, acquire and release general registers

LDSMIN Ws, Wt, [Xn|SP] ; 32-bit, no memory ordering general registers

LDSMINL Ws, Wt, [Xn|SP] ; 32-bit, release general registers

LDSMINA Xs, Xt, [Xn|SP] ; 64-bit, acquire general registers

LDSMINAL Xs, Xt, [Xn|SP] ; 64-bit, acquire and release general registers

LDSMIN Xs, Xt, [Xn|SP] ; 64-bit, no memory ordering general registers

LDSMINL Xs, Xt, [Xn|SP] ; 64-bit, release general registers
```

Where:

Ws
Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location.

Wt
Is the 32-bit name of the general-purpose register to be loaded.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Xs
Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location.

Xt
Is the 64-bit name of the general-purpose register to be loaded.

Architectures supported

Supported in the Armv8.1 architecture and later.

Usage

Atomic signed minimum on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of `WZR` or `XZR`, `LDSMINA` and `LDSMINAL` load from memory with acquire semantics.
- `LDSMINL` and `LDSMINAL` store to memory with release semantics.
- `LDSMIN` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.54 LDSMINAB, LDSMINALB, LDSMINB, LDSMINLB

Atomic signed minimum on byte in memory.

Syntax

```
LDSMINAB Ws, Wt, [Xn|SP] ; Acquire general registers

LDSMINALB Ws, Wt, [Xn|SP] ; Acquire and release general registers

LDSMINB Ws, Wt, [Xn|SP] ; No memory ordering general registers

LDSMINLB Ws, Wt, [Xn|SP] ; Release general registers
```

Where:

Ws
Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location.

Wt
Is the 32-bit name of the general-purpose register to be loaded.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Architectures supported

Supported in the Armv8.1 architecture and later.

Usage

Atomic signed minimum on byte in memory atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not `WZR`, `LDSMINAB` and `LDSMINALB` load from memory with acquire semantics.
- `LDSMINLB` and `LDSMINALB` store to memory with release semantics.
- `LDSMINB` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.55 LDSMINAH, LDSMINALH, LDSMINH, LDSMINLH

Atomic signed minimum on halfword in memory.

Syntax

```
LDSMINAH Ws, Wt, [Xn|SP] ; Acquire general registers

LDSMINALH Ws, Wt, [Xn|SP] ; Acquire and release general registers

LDSMINH Ws, Wt, [Xn|SP] ; No memory ordering general registers

LDSMINLH Ws, Wt, [Xn|SP] ; Release general registers
```

Where:

Ws
Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location.

Wt
Is the 32-bit name of the general-purpose register to be loaded.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Architectures supported

Supported in the Armv8.1 architecture and later.

Usage

Atomic signed minimum on halfword in memory atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not `WZR`, `LDSMINAH` and `LDSMINALH` load from memory with acquire semantics.
- `LDSMINLH` and `LDSMINALH` store to memory with release semantics.
- `LDSMINH` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.56 LDTR

Load Register (unprivileged).

Syntax

```
LDTR Wt, [Xn|SP{, #simm}] ; 32-bit
LDTR Xt, [Xn|SP{, #simm}] ; 64-bit
```

Where:

Wt
Is the 32-bit name of the general-purpose register to be transferred.

Xt
Is the 64-bit name of the general-purpose register to be transferred.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

simm
Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0.

Usage

Load Register (unprivileged) loads a word or doubleword from memory, and writes it to a register. The address that is used for the load is calculated from a base register and an immediate offset.

The memory is restricted as if execution is at EL0 when:

- Executing at EL1.
- Executing at EL2, in Armv8.1, with HCR_EL2.{E2H, TGE} set to {1, 1}.

Otherwise, the access permission is for the Exception level at which the instruction is executed. For information about memory accesses, see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.57 LDTRB

Load Register Byte (unprivileged).

Syntax

```
LDTRB Wt, [Xn|SP{, #simm}]
```

Where:

Wt
Is the 32-bit name of the general-purpose register to be transferred.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

simm
Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0.

Usage

Load Register Byte (unprivileged) loads a byte from memory, zero-extends it, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset.

The memory is restricted as if execution is at EL0 when:

- Executing at EL1.
- Executing at EL2, in Armv8.1, with HCR_EL2.{E2H, TGE} set to {1, 1}.

Otherwise, the access permission is for the Exception level at which the instruction is executed. For information about memory accesses, see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.58 LDTRH

Load Register Halfword (unprivileged).

Syntax

```
LDTRH Wt, [Xn|SP{, #simm}]
```

Where:

Wt
Is the 32-bit name of the general-purpose register to be transferred.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

simm
Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0.

Usage

Load Register Halfword (unprivileged) loads a halfword from memory, zero-extends it, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset.

The memory is restricted as if execution is at EL0 when:

- Executing at EL1.
- Executing at EL2, in Armv8.1, with HCR_EL2.{E2H, TGE} set to {1, 1}.

Otherwise, the access permission is for the Exception level at which the instruction is executed. For information about memory accesses, see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.59 LDTRSB

Load Register Signed Byte (unprivileged).

Syntax

```
LDTRSB Wt, [Xn|SP{, #simm}] ; 32-bit

LDTRSB Xt, [Xn|SP{, #simm}] ; 64-bit
```

Where:

Wt
Is the 32-bit name of the general-purpose register to be transferred.

Xt
Is the 64-bit name of the general-purpose register to be transferred.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

simm
Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0.

Usage

Load Register Signed Byte (unprivileged) loads a byte from memory, sign-extends it to 32 bits or 64 bits, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset.

The memory is restricted as if execution is at EL0 when:

- Executing at EL1.
- Executing at EL2, in Armv8.1, with HCR_EL2.{E2H, TGE} set to {1, 1}.

Otherwise, the access permission is for the Exception level at which the instruction is executed. For information about memory accesses, see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.60 LDTRSH

Load Register Signed Halfword (unprivileged).

Syntax

```
LDTRSH Wt, [Xn|SP{, #simm}] ; 32-bit

LDTRSH Xt, [Xn|SP{, #simm}] ; 64-bit
```

Where:

Wt
Is the 32-bit name of the general-purpose register to be transferred.

Xt
Is the 64-bit name of the general-purpose register to be transferred.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

simm
Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0.

Usage

Load Register Signed Halfword (unprivileged) loads a halfword from memory, sign-extends it to 32 bits or 64 bits, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset.

The memory is restricted as if execution is at EL0 when:

- Executing at EL1.
- Executing at EL2, in Armv8.1, with HCR_EL2.{E2H, TGE} set to {1, 1}.

Otherwise, the access permission is for the Exception level at which the instruction is executed. For information about memory accesses, see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.61 LDTRSW

Load Register Signed Word (unprivileged).

Syntax

```
LDTRSW Xt, [Xn|SP{, #simm}]
```

Where:

Xt
Is the 64-bit name of the general-purpose register to be transferred.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

simm
Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0.

Usage

Load Register Signed Word (unprivileged) loads a word from memory, sign-extends it to 64 bits, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset.

The memory is restricted as if execution is at EL0 when:

- Executing at EL1.
- Executing at EL2, in Armv8.1, with HCR_EL2.{E2H, TGE} set to {1, 1}.

Otherwise, the access permission is for the Exception level at which the instruction is executed. For information about memory accesses, see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.62 LDUMAXA, LDUMAXAL, LDUMAX, LDUMAXL, LDUMAXAL, LDUMAX, LDUMAXL

Atomic unsigned maximum on word or doubleword in memory.

Syntax

```
LDUMAXA Ws, Wt, [Xn|SP] ; 32-bit, acquire general registers

LDUMAXAL Ws, Wt, [Xn|SP] ; 32-bit, acquire and release general registers

LDUMAX Ws, Wt, [Xn|SP] ; 32-bit, no memory ordering general registers

LDUMAXL Ws, Wt, [Xn|SP] ; 32-bit, release general registers

LDUMAXA Xs, Xt, [Xn|SP] ; 64-bit, acquire general registers

LDUMAXAL Xs, Xt, [Xn|SP] ; 64-bit, acquire and release general registers

LDUMAX Xs, Xt, [Xn|SP] ; 64-bit, no memory ordering general registers

LDUMAXL Xs, Xt, [Xn|SP] ; 64-bit, release general registers
```

Where:

Ws
Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location.

Wt
Is the 32-bit name of the general-purpose register to be loaded.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Xs
Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location.

Xt
Is the 64-bit name of the general-purpose register to be loaded.

Architectures supported

Supported in the Armv8.1 architecture and later.

Usage

Atomic unsigned maximum on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of `WZR` or `XZR`, `LDUMAXA` and `LDUMAXAL` load from memory with acquire semantics.
- `LDUMAXL` and `LDUMAXAL` store to memory with release semantics.
- `LDUMAX` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.63 LDUMAXAB, LDUMAXALB, LDUMAXB, LDUMAXLB

Atomic unsigned maximum on byte in memory.

Syntax

```
LDUMAXAB Ws, Wt, [Xn|SP] ; Acquire general registers

LDUMAXALB Ws, Wt, [Xn|SP] ; Acquire and release general registers

LDUMAXB Ws, Wt, [Xn|SP] ; No memory ordering general registers

LDUMAXLB Ws, Wt, [Xn|SP] ; Release general registers
```

Where:

Ws
Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location.

Wt
Is the 32-bit name of the general-purpose register to be loaded.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Architectures supported

Supported in the Armv8.1 architecture and later.

Usage

Atomic unsigned maximum on byte in memory atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not `WZR`, `LDUMAXAB` and `LDUMAXALB` load from memory with acquire semantics.
- `LDUMAXLB` and `LDUMAXALB` store to memory with release semantics.
- `LDUMAXB` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.64 LDUMAXAH, LDUMAXALH, LDUMAXH, LDUMAXLH

Atomic unsigned maximum on halfword in memory.

Syntax

```
LDUMAXAH Ws, Wt, [Xn|SP] ; Acquire general registers

LDUMAXALH Ws, Wt, [Xn|SP] ; Acquire and release general registers

LDUMAXH Ws, Wt, [Xn|SP] ; No memory ordering general registers

LDUMAXLH Ws, Wt, [Xn|SP] ; Release general registers
```

Where:

Ws
Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location.

Wt
Is the 32-bit name of the general-purpose register to be loaded.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Architectures supported

Supported in the Armv8.1 architecture and later.

Usage

Atomic unsigned maximum on halfword in memory atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not `WZR`, `LDUMAXAH` and `LDUMAXALH` load from memory with acquire semantics.
- `LDUMAXLH` and `LDUMAXALH` store to memory with release semantics.
- `LDUMAXH` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.65 LDUMINA, LDUMINAL, LDUMIN, LDUMINL, LDUMINAL, LDUMIN, LDUMINL

Atomic unsigned minimum on word or doubleword in memory.

Syntax

```
LDUMINA Ws, Wt, [Xn|SP] ; 32-bit, acquire general registers

LDUMINAL Ws, Wt, [Xn|SP] ; 32-bit, acquire and release general registers

LDUMIN Ws, Wt, [Xn|SP] ; 32-bit, no memory ordering general registers

LDUMINL Ws, Wt, [Xn|SP] ; 32-bit, release general registers

LDUMINA Xs, Xt, [Xn|SP] ; 64-bit, acquire general registers

LDUMINAL Xs, Xt, [Xn|SP] ; 64-bit, acquire and release general registers

LDUMIN Xs, Xt, [Xn|SP] ; 64-bit, no memory ordering general registers

LDUMINL Xs, Xt, [Xn|SP] ; 64-bit, release general registers
```

Where:

Ws
Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location.

Wt
Is the 32-bit name of the general-purpose register to be loaded.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Xs
Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location.

Xt
Is the 64-bit name of the general-purpose register to be loaded.

Architectures supported

Supported in the Armv8.1 architecture and later.

Usage

Atomic unsigned minimum on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of `WZR` or `XZR`, `LDUMINA` and `LDUMINAL` load from memory with acquire semantics.
- `LDUMINL` and `LDUMINAL` store to memory with release semantics.
- `LDUMIN` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.66 LDUMINAB, LDUMINALB, LDUMINB, LDUMINLB

Atomic unsigned minimum on byte in memory.

Syntax

```
LDUMINAB Ws, Wt, [Xn|SP] ; Acquire general registers

LDUMINALB Ws, Wt, [Xn|SP] ; Acquire and release general registers

LDUMINB Ws, Wt, [Xn|SP] ; No memory ordering general registers

LDUMINLB Ws, Wt, [Xn|SP] ; Release general registers
```

Where:

Ws
Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location.

Wt
Is the 32-bit name of the general-purpose register to be loaded.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Architectures supported

Supported in the Armv8.1 architecture and later.

Usage

Atomic unsigned minimum on byte in memory atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not `WZR`, `LDUMINAB` and `LDUMINALB` load from memory with acquire semantics.
- `LDUMINLB` and `LDUMINALB` store to memory with release semantics.
- `LDUMINB` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.67 LDUMINAH, LDUMINALH, LDUMINH, LDUMINLH

Atomic unsigned minimum on halfword in memory.

Syntax

```
LDUMINAH Ws, Wt, [Xn|SP] ; Acquire general registers

LDUMINALH Ws, Wt, [Xn|SP] ; Acquire and release general registers

LDUMINH Ws, Wt, [Xn|SP] ; No memory ordering general registers

LDUMINLH Ws, Wt, [Xn|SP] ; Release general registers
```

Where:

Ws
Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location.

Wt
Is the 32-bit name of the general-purpose register to be loaded.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Architectures supported

Supported in the Armv8.1 architecture and later.

Usage

Atomic unsigned minimum on halfword in memory atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not `WZR`, `LDUMINAH` and `LDUMINALH` load from memory with acquire semantics.
- `LDUMINLH` and `LDUMINALH` store to memory with release semantics.
- `LDUMINH` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.68 LDUR

Load Register (unscaled).

Syntax

```
LDUR Wt, [Xn|SP{, #simm}] ; 32-bit
LDUR Xt, [Xn|SP{, #simm}] ; 64-bit
```

Where:

Wt
Is the 32-bit name of the general-purpose register to be transferred.

Xt
Is the 64-bit name of the general-purpose register to be transferred.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

simm
Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0.

Usage

Load Register (unscaled) calculates an address from a base register and an immediate offset, loads a 32-bit word or 64-bit doubleword from memory, zero-extends it, and writes it to a register. For information about memory accesses, see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.69 LDURB

Load Register Byte (unscaled).

Syntax

```
LDURB Wt, [Xn|SP{, #simm}]
```

Where:

Wt
Is the 32-bit name of the general-purpose register to be transferred.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

simm
Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0.

Usage

Load Register Byte (unscaled) calculates an address from a base register and an immediate offset, loads a byte from memory, zero-extends it, and writes it to a register. For information about memory accesses, see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.70 LDURH

Load Register Halfword (unscaled).

Syntax

```
LDURH Wt, [Xn|SP{, #simm}]
```

Where:

Wt
Is the 32-bit name of the general-purpose register to be transferred.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

simm
Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0.

Usage

Load Register Halfword (unscaled) calculates an address from a base register and an immediate offset, loads a halfword from memory, zero-extends it, and writes it to a register. For information about memory accesses, see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.71 LDURSB

Load Register Signed Byte (unscaled).

Syntax

```
LDURSB Wt, [Xn|SP{, #simm}] ; 32-bit
LDURSB Xt, [Xn|SP{, #simm}] ; 64-bit
```

Where:

Wt
Is the 32-bit name of the general-purpose register to be transferred.

Xt
Is the 64-bit name of the general-purpose register to be transferred.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

simm
Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0.

Usage

Load Register Signed Byte (unscaled) calculates an address from a base register and an immediate offset, loads a signed byte from memory, sign-extends it, and writes it to a register. For information about memory accesses, see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.72 LDURSH

Load Register Signed Halfword (unscaled).

Syntax

```
LDURSH Wt, [Xn|SP{, #simm}] ; 32-bit
LDURSH Xt, [Xn|SP{, #simm}] ; 64-bit
```

Where:

Wt
Is the 32-bit name of the general-purpose register to be transferred.

Xt
Is the 64-bit name of the general-purpose register to be transferred.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

simm
Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0.

Usage

Load Register Signed Halfword (unscaled) calculates an address from a base register and an immediate offset, loads a signed halfword from memory, sign-extends it, and writes it to a register. For information about memory accesses, see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.73 LDURSW

Load Register Signed Word (unscaled).

Syntax

```
LDURSW Xt, [Xn|SP{, #simm}]
```

Where:

Xt
Is the 64-bit name of the general-purpose register to be transferred.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

simm
Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0.

Usage

Load Register Signed Word (unscaled) calculates an address from a base register and an immediate offset, loads a signed word from memory, sign-extends it, and writes it to a register. For information about memory accesses, see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.74 LDXP

Load Exclusive Pair of Registers.

Syntax

```
LDXP Wt1, Wt2, [Xn|SP{,#0}] ; 32-bit

LDXP Xt1, Xt2, [Xn|SP{,#0}] ; 64-bit
```

Where:

Wt1
Is the 32-bit name of the first general-purpose register to be transferred.

Wt2
Is the 32-bit name of the second general-purpose register to be transferred.

Xt1
Is the 64-bit name of the first general-purpose register to be transferred.

Xt2
Is the 64-bit name of the second general-purpose register to be transferred.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Usage

Load Exclusive Pair of Registers derives an address from a base register value, loads two 32-bit words or two 64-bit doublewords from memory, and writes them to two registers. A 32-bit pair requires the address to be doubleword aligned and is single-copy atomic at doubleword granularity. A 64-bit pair requires the address to be quadword aligned and is single-copy atomic for each doubleword at doubleword granularity. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See *Synchronization and semaphores* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*. For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Note

For 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*, and particularly *LDXP*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.75 LDXR

Load Exclusive Register.

Syntax

```
LDXR Wt, [Xn|SP{,#0}] ; 32-bit

LDXR Xt, [Xn|SP{,#0}] ; 64-bit
```

Where:

Wt
Is the 32-bit name of the general-purpose register to be transferred.

Xt
Is the 64-bit name of the general-purpose register to be transferred.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Usage

Load Exclusive Register derives an address from a base register value, loads a 32-bit word or a 64-bit doubleword from memory, and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See *Synchronization and semaphores* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*. For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.76 LDXRB

Load Exclusive Register Byte.

Syntax

```
LDXRB Wt, [Xn|SP{,#0}]
```

Where:

Wt
Is the 32-bit name of the general-purpose register to be transferred.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Usage

Load Exclusive Register Byte derives an address from a base register value, loads a byte from memory, zero-extends it and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See *Synchronization and semaphores* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*. For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.77 LDXRH

Load Exclusive Register Halfword.

Syntax

```
LDXRH Wt, [Xn|SP{,#0}]
```

Where:

Wt
Is the 32-bit name of the general-purpose register to be transferred.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Usage

Load Exclusive Register Halfword derives an address from a base register value, loads a halfword from memory, zero-extends it and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See *Synchronization and semaphores* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*. For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.78 PRFM (immediate)

Prefetch Memory (immediate).

Syntax

`PRFM (prfop|#imm5), [Xn|SP{, #pimm}]`

Where:

prfop

Is the prefetch operation, defined as `type<target><policy>`.

`type` is one of:

PLD
: Prefetch for load.

PLI
: Preload instructions.

PST
: Prefetch for store.

`<target>` is one of:

L1
: Level 1 cache.

L2
: Level 2 cache.

L3
: Level 3 cache.

`<policy>` is one of:

KEEP
: Retained or temporal prefetch, allocated in the cache normally.

STRM
: Streaming or non-temporal prefetch, for data that is used only once.

imm5

Is the prefetch operation encoding as an immediate, in the range 0 to 31.

This syntax is only for encodings that are not accessible using *prfop*.

Xn|SP

Is the 64-bit name of the general-purpose base register or stack pointer.

pimm

Is the optional positive immediate byte offset, a multiple of 8 in the range 0 to 32760, defaulting to 0.

Usage

Prefetch Memory (immediate) signals the memory system that data memory accesses from a specified address are likely to occur in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as preloading the cache line containing the specified address into one or more caches.

The effect of an PRFM instruction is IMPLEMENTATION DEFINED. For more information, see *Prefetch memory* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

For information about memory accesses, see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.79 PRFM (literal)

Prefetch Memory (literal).

Syntax

`PRFM (prfop|#imm5), label`

Where:

prfop

Is the prefetch operation, defined as *type*<target><policy>.

type is one of:

PLD
: Prefetch for load.

PLI
: Preload instructions.

PST
: Prefetch for store.

<target> is one of:

L1
: Level 1 cache.

L2
: Level 2 cache.

L3
: Level 3 cache.

<policy> is one of:

KEEP
: Retained or temporal prefetch, allocated in the cache normally.

STRM
: Streaming or non-temporal prefetch, for data that is used only once.

imm5

Is the prefetch operation encoding as an immediate, in the range 0 to 31.

This syntax is only for encodings that are not accessible using *prfop*.

label

Is the program label from which the data is to be loaded. Its offset from the address of this instruction, in the range ±1MB.

Usage

Prefetch Memory (literal) signals the memory system that data memory accesses from a specified address are likely to occur in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as preloading the cache line containing the specified address into one or more caches.

The effect of an PRFM instruction is IMPLEMENTATION DEFINED. For more information, see *Prefetch memory* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

For information about memory accesses, see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.80 PRFM (register)

Prefetch Memory (register).

Syntax

`PRFM (prfop|#imm5), [Xn|SP, (Wm|Xm){, extend {amount}}]`

Where:

prfop

Is the prefetch operation, defined as *type*<target><policy>.

type is one of:

PLD
: Prefetch for load.

PLI
: Preload instructions.

PST
: Prefetch for store.

<target> is one of:

L1
: Level 1 cache.

L2
: Level 2 cache.

L3
: Level 3 cache.

<policy> is one of:

KEEP
: Retained or temporal prefetch, allocated in the cache normally.

STRM
: Streaming or non-temporal prefetch, for data that is used only once.

imm5

Is the prefetch operation encoding as an immediate, in the range 0 to 31.

This syntax is only for encodings that are not accessible using *prfop*.

Xn|SP

Is the 64-bit name of the general-purpose base register or stack pointer.

Wm

When "option<0>" is set to 0, is the 32-bit name of the general-purpose index register.

Xm

When "option<0>" is set to 1, is the 64-bit name of the general-purpose index register.

extend

Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when *amount* is omitted, and can be one of UXTW, LSL, SXTW or SXTX.

amount

Is the index shift amount, optional only when *extend* is not LSL. Where it is permitted to be optional, it defaults to #0. It is, and can be either #0 or #3.

Usage

Prefetch Memory (register) signals the memory system that data memory accesses from a specified address are likely to occur in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as preloading the cache line containing the specified address into one or more caches.

The effect of an PRFM instruction is IMPLEMENTATION DEFINED. For more information, see *Prefetch memory* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

For information about memory accesses, see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.81 PRFUM (unscaled offset)

Prefetch Memory (unscaled offset).

Syntax

`PRFUM (prfop|#imm5), [Xn|SP{, #simm}]`

Where:

prfop

Is the prefetch operation, defined as *type*<target><policy>.

type is one of:

PLD
: Prefetch for load.

PLI
: Preload instructions.

PST
: Prefetch for store.

<target> is one of:

L1
: Level 1 cache.

L2
: Level 2 cache.

L3
: Level 3 cache.

<policy> is one of:

KEEP
: Retained or temporal prefetch, allocated in the cache normally.

STRM
: Streaming or non-temporal prefetch, for data that is used only once.

imm5

Is the prefetch operation encoding as an immediate, in the range 0 to 31.

This syntax is only for encodings that are not accessible using *prfop*.

Xn|SP

Is the 64-bit name of the general-purpose base register or stack pointer.

simm

Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0.

Usage

Prefetch Memory (unscaled offset) signals the memory system that data memory accesses from a specified address are likely to occur in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as preloading the cache line containing the specified address into one or more caches.

The effect of an PRFUM instruction is IMPLEMENTATION DEFINED. For more information, see *Prefetch memory* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

For information about memory accesses, see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.82 STADD, STADDL, STADDL

Atomic add on word or doubleword in memory, without return.

Syntax

```
STADD Ws, [Xn|SP] ; 32-bit, no memory ordering general registers

STADDL Ws, [Xn|SP] ; 32-bit, release general registers

STADD Xs, [Xn|SP] ; 64-bit, no memory ordering general registers

STADDL Xs, [Xn|SP] ; 64-bit, release general registers
```

Where:

Ws
Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Xs
Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location.

Architectures supported

Supported in the Armv8.1 architecture and later.

Usage

Atomic add on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, adds the value held in a register to it, and stores the result back to memory.

- `STADD` has no memory ordering semantics.
- `STADDL` stores to memory with release semantics, as described in *Load-Acquire, Store-Release* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.83 STADDB, STADDLB

Atomic add on byte in memory, without return.

Syntax

```
STADDB Ws, [Xn|SP] ; No memory ordering general registers

STADDLB Ws, [Xn|SP] ; Release general registers
```

Where:

Ws
Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Architectures supported

Supported in the Armv8.1 architecture and later.

Usage

Atomic add on byte in memory, without return, atomically loads an 8-bit byte from memory, adds the value held in a register to it, and stores the result back to memory.

- `STADDB` has no memory ordering semantics.
- `STADDLB` stores to memory with release semantics, as described in *Load-Acquire, Store-Release* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.84 STADDH, STADDLH

Atomic add on halfword in memory, without return.

Syntax

```
STADDH Ws, [Xn|SP] ; No memory ordering general registers

STADDLH Ws, [Xn|SP] ; Release general registers
```

Where:

Ws
Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Architectures supported

Supported in the Armv8.1 architecture and later.

Usage

Atomic add on halfword in memory, without return, atomically loads a 16-bit halfword from memory, adds the value held in a register to it, and stores the result back to memory.

- `STADDH` has no memory ordering semantics.
- `STADDLH` stores to memory with release semantics, as described in *Load-Acquire, Store-Release* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.85 STCLR, STCLRL, STCLRL

Atomic bit clear on word or doubleword in memory, without return.

Syntax

```
STCLR Ws, [Xn|SP] ; 32-bit, no memory ordering general registers

STCLRL Ws, [Xn|SP] ; 32-bit, release general registers

STCLR Xs, [Xn|SP] ; 64-bit, no memory ordering general registers

STCLRL Xs, [Xn|SP] ; 64-bit, release general registers
```

Where:

Ws
Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Xs
Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location.

Architectures supported

Supported in the Armv8.1 architecture and later.

Usage

Atomic bit clear on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory.

- `STCLR` has no memory ordering semantics.
- `STCLRL` stores to memory with release semantics, as described in *Load-Acquire, Store-Release* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.86 STCLRB, STCLRLB

Atomic bit clear on byte in memory, without return.

Syntax

```
STCLRB Ws, [Xn|SP] ; No memory ordering general registers

STCLRLB Ws, [Xn|SP] ; Release general registers
```

Where:

Ws
Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Architectures supported

Supported in the Armv8.1 architecture and later.

Usage

Atomic bit clear on byte in memory, without return, atomically loads an 8-bit byte from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory.

- `STCLRB` has no memory ordering semantics.
- `STCLRLB` stores to memory with release semantics, as described in *Load-Acquire, Store-Release* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.87 STCLRH, STCLRLH

Atomic bit clear on halfword in memory, without return.

Syntax

```
STCLRH Ws, [Xn|SP] ; No memory ordering general registers

STCLRLH Ws, [Xn|SP] ; Release general registers
```

Where:

Ws
Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Architectures supported

Supported in the Armv8.1 architecture and later.

Usage

Atomic bit clear on halfword in memory, without return, atomically loads a 16-bit halfword from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory.

- `STCLRH` has no memory ordering semantics.
- `STCLRLH` stores to memory with release semantics, as described in *Load-Acquire, Store-Release* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.88 STEOR, STEORL, STEORL

Atomic exclusive OR on word or doubleword in memory, without return.

Syntax

```
STEOR Ws, [Xn|SP] ; 32-bit, no memory ordering general registers

STEORL Ws, [Xn|SP] ; 32-bit, release general registers

STEOR Xs, [Xn|SP] ; 64-bit, no memory ordering general registers

STEORL Xs, [Xn|SP] ; 64-bit, release general registers
```

Where:

Ws
Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Xs
Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location.

Architectures supported

Supported in the Armv8.1 architecture and later.

Usage

Atomic exclusive OR on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, performs an exclusive OR with the value held in a register on it, and stores the result back to memory.

- `STEOR` has no memory ordering semantics.
- `STEORL` stores to memory with release semantics, as described in *Load-Acquire, Store-Release* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.89 STEORB, STEORLB

Atomic exclusive OR on byte in memory, without return.

Syntax

```
STEORB Ws, [Xn|SP] ; No memory ordering general registers

STEORLB Ws, [Xn|SP] ; Release general registers
```

Where:

Ws
Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Architectures supported

Supported in the Armv8.1 architecture and later.

Usage

Atomic exclusive OR on byte in memory, without return, atomically loads an 8-bit byte from memory, performs an exclusive OR with the value held in a register on it, and stores the result back to memory.

- `STEORB` has no memory ordering semantics.
- `STEORLB` stores to memory with release semantics, as described in *Load-Acquire, Store-Release* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.90 STEORH, STEORLH

Atomic exclusive OR on halfword in memory, without return.

Syntax

```
STEORH Ws, [Xn|SP] ; No memory ordering general registers

STEORLH Ws, [Xn|SP] ; Release general registers
```

Where:

Ws
Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Architectures supported

Supported in the Armv8.1 architecture and later.

Usage

Atomic exclusive OR on halfword in memory, without return, atomically loads a 16-bit halfword from memory, performs an exclusive OR with the value held in a register on it, and stores the result back to memory.

- `STEORH` has no memory ordering semantics.
- `STEORLH` stores to memory with release semantics, as described in *Load-Acquire, Store-Release* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.91 STLLR

Store LORelease Register.

Syntax

```
STLLR Wt, [Xn|SP{,#0}] ; 32-bit
STLLR Xt, [Xn|SP{,#0}] ; 64-bit
```

Where:

Wt
Is the 32-bit name of the general-purpose register to be transferred.

Xt
Is the 64-bit name of the general-purpose register to be transferred.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Architectures supported

Supported in the Armv8.1 architecture and later.

Usage

Store LORelease Register stores a 32-bit word or a 64-bit doubleword to a memory location, from a register. The instruction also has memory ordering semantics as described in *Load LOAcquire, Store LORelease* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*. For information about memory accesses, see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.92 STLLRB

Store LORelease Register Byte.

Syntax

`STLLRB Wt, [Xn|SP{,#0}]`

Where:

Wt
Is the 32-bit name of the general-purpose register to be transferred.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Architectures supported

Supported in the Armv8.1 architecture and later.

Usage

Store LORelease Register Byte stores a byte from a 32-bit register to a memory location. The instruction also has memory ordering semantics as described in *Load LOAcquire, Store LORelease* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*. For information about memory accesses, see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.93 STLLRH

Store LORelease Register Halfword.

Syntax

```
STLLRH Wt, [Xn|SP{,#0}]
```

Where:

Wt
Is the 32-bit name of the general-purpose register to be transferred.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Architectures supported

Supported in the Armv8.1 architecture and later.

Usage

Store LORelease Register Halfword stores a halfword from a 32-bit register to a memory location. The instruction also has memory ordering semantics as described in *Load LOAcquire, Store LORelease* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*. For information about memory accesses, see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.94 STLR

Store-Release Register.

Syntax

```
STLR Wt, [Xn|SP{,#0}] ; 32-bit
STLR Xt, [Xn|SP{,#0}] ; 64-bit
```

Where:

Wt
Is the 32-bit name of the general-purpose register to be transferred.

Xt
Is the 64-bit name of the general-purpose register to be transferred.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Usage

Store-Release Register stores a 32-bit word or a 64-bit doubleword to a memory location, from a register. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*. For information about memory accesses, see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.95 STLRB

Store-Release Register Byte.

Syntax

```
STLRB Wt, [Xn|SP{,#0}]
```

Where:

Wt
Is the 32-bit name of the general-purpose register to be transferred.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Usage

Store-Release Register Byte stores a byte from a 32-bit register to a memory location. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*. For information about memory accesses, see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.96 STLRH

Store-Release Register Halfword.

Syntax

```
STLRH Wt, [Xn|SP{,#0}]
```

Where:

Wt
Is the 32-bit name of the general-purpose register to be transferred.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Usage

Store-Release Register Halfword stores a halfword from a 32-bit register to a memory location. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*. For information about memory accesses, see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.97 STLXP

Store-Release Exclusive Pair of registers.

Syntax

```
STLXP Ws, Wt1, Wt2, [Xn|SP{,#0}] ; 32-bit

STLXP Ws, Xt1, Xt2, [Xn|SP{,#0}] ; 64-bit
```

Where:

Wt1
Is the 32-bit name of the first general-purpose register to be transferred.

Wt2
Is the 32-bit name of the second general-purpose register to be transferred.

Xt1
Is the 64-bit name of the first general-purpose register to be transferred.

Xt2
Is the 64-bit name of the second general-purpose register to be transferred.

Ws
Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written. The value returned is.

0
If the operation updates memory.

1
If the operation fails to update memory.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- *Ws* is not updated.

Accessing an address that is not aligned to the size of the data being accessed causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- The exception is generated if the Exclusive Monitors for the current PE include all of the addresses associated with the virtual address region of size bytes starting at address. The immediately following memory write must be to the same addresses.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

Whether the detection of memory aborts happens before or after the check on the local Exclusive Monitor depends on the implementation. As a result a failure of the local monitor can occur on some implementations even if the memory access would give a memory abort.

Usage

Store-Release Exclusive Pair of registers stores two 32-bit words or two 64-bit doublewords to a memory location if the PE has exclusive access to the memory address, from two registers, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See *Synchronization and semaphores* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*. A 32-bit pair requires the address to be doubleword aligned and is single-copy atomic at doubleword granularity. A 64-bit pair requires the address to be quadword aligned and, if the Store-Exclusive succeeds, it causes a single-copy atomic update of the 128-bit memory location being updated. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*. For information about

memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Note

For 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*, and particularly *STLXP*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.98 STLXR

Store-Release Exclusive Register.

Syntax

```
STLXR Ws, Wt, [Xn|SP{,#0}] ; 32-bit

STLXR Ws, Xt, [Xn|SP{,#0}] ; 64-bit
```

Where:

Wt
Is the 32-bit name of the general-purpose register to be transferred.

Xt
Is the 64-bit name of the general-purpose register to be transferred.

Ws
Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written. The value returned is.

0
If the operation updates memory.

1
If the operation fails to update memory.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- *Ws* is not updated.

Accessing an address that is not aligned to the size of the data being accessed causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- The exception is generated if the Exclusive Monitors for the current PE include all of the addresses associated with the virtual address region of size bytes starting at address. The immediately following memory write must be to the same addresses.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

Whether the detection of memory aborts happens before or after the check on the local Exclusive Monitor depends on the implementation. As a result a failure of the local monitor can occur on some implementations even if the memory access would give a memory abort.

Usage

Store-Release Exclusive Register stores a 32-bit word or a 64-bit doubleword to memory if the PE has exclusive access to the memory address, from two registers, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See *Synchronization and semaphores* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*. The memory access is atomic. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*. For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Note

For 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*, and particularly *STLXR*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.99 STLXRB

Store-Release Exclusive Register Byte.

Syntax

```
STLXRB Ws, Wt, [Xn|SP{,#0}]
```

Where:

Ws
Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written. The value returned is.

0
If the operation updates memory.

1
If the operation fails to update memory.

Wt
Is the 32-bit name of the general-purpose register to be transferred.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Aborts

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- *Ws* is not updated.

Whether the detection of memory aborts happens before or after the check on the local Exclusive Monitor depends on the implementation. As a result a failure of the local monitor can occur on some implementations even if the memory access would give a memory abort.

Usage

Store-Release Exclusive Register Byte stores a byte from a 32-bit register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See *Synchronization and semaphores* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*. The memory access is atomic. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*. For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Note

For 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*, and particularly *STLXRB*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.100 STLXRH

Store-Release Exclusive Register Halfword.

Syntax

```
STLXRH Ws, Wt, [Xn|SP{,#0}]
```

Where:

Ws
Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written. The value returned is.

0
If the operation updates memory.

1
If the operation fails to update memory.

Wt
Is the 32-bit name of the general-purpose register to be transferred.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- `Ws` is not updated.

A non halfword-aligned memory address causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- The exception is generated if the Exclusive Monitors for the current PE include all of the addresses associated with the virtual address region of size bytes starting at address. The immediately following memory write must be to the same addresses.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

Whether the detection of memory aborts happens before or after the check on the local Exclusive Monitor depends on the implementation. As a result a failure of the local monitor can occur on some implementations even if the memory access would give a memory abort.

Usage

Store-Release Exclusive Register Halfword stores a halfword from a 32-bit register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See *Synchronization and semaphores* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*. The memory access is atomic. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*. For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Note

For 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*, and particularly *STLXRH*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.101 STNP

Store Pair of Registers, with non-temporal hint.

Syntax

```
STNP Wt1, Wt2, [Xn|SP{, #imm}] ; 32-bit, Signed offset

STNP Xt1, Xt2, [Xn|SP{, #imm}] ; 64-bit, Signed offset
```

Where:

Wt1
Is the 32-bit name of the first general-purpose register to be transferred.

Wt2
Is the 32-bit name of the second general-purpose register to be transferred.

imm
Depends on the instruction variant:

32-bit general registers
Is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0.

64-bit general registers
Is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0.

Xt1
Is the 64-bit name of the first general-purpose register to be transferred.

Xt2
Is the 64-bit name of the second general-purpose register to be transferred.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Usage

Store Pair of Registers, with non-temporal hint, calculates an address from a base register value and an immediate offset, and stores two 32-bit words or two 64-bit doublewords to the calculated address, from two registers. For information about memory accesses, see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*. For information about Non-temporal pair instructions, see *Load/Store Non-temporal pair* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.102 STP

Store Pair of Registers.

Syntax

```
STP Wt1, Wt2, [Xn|SP], #imm ; 32-bit, Post-index

STP Xt1, Xt2, [Xn|SP], #imm ; 64-bit, Post-index

STP Wt1, Wt2, [Xn|SP, #imm]! ; 32-bit, Pre-index

STP Xt1, Xt2, [Xn|SP, #imm]! ; 64-bit, Pre-index

STP Wt1, Wt2, [Xn|SP{, #imm}] ; 32-bit, Signed offset

STP Xt1, Xt2, [Xn|SP{, #imm}] ; 64-bit, Signed offset
```

Where:

Wt1
Is the 32-bit name of the first general-purpose register to be transferred.

Wt2
Is the 32-bit name of the second general-purpose register to be transferred.

imm
Depends on the instruction variant:

32-bit general registers
Is the signed immediate byte offset, a multiple of 4 in the range -256 to 252.

64-bit general registers
Is the signed immediate byte offset, a multiple of 8 in the range -512 to 504.

Xt1
Is the 64-bit name of the first general-purpose register to be transferred.

Xt2
Is the 64-bit name of the second general-purpose register to be transferred.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Usage

Store Pair of Registers calculates an address from a base register value and an immediate offset, and stores two 32-bit words or two 64-bit doublewords to the calculated address, from two registers. For information about memory accesses, see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Note

For 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*, and particularly *STP*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.103 STR (immediate)

Store Register (immediate).

Syntax

```
STR Wt, [Xn|SP], #simm ; 32-bit, Post-index

STR Xt, [Xn|SP], #simm ; 64-bit, Post-index

STR Wt, [Xn|SP, #simm]! ; 32-bit, Pre-index

STR Xt, [Xn|SP, #simm]! ; 64-bit, Pre-index

STR Wt, [Xn|SP{, #pimm}] ; 32-bit

STR Xt, [Xn|SP{, #pimm}] ; 64-bit
```

Where:

Wt
Is the 32-bit name of the general-purpose register to be transferred.

simm
Is the signed immediate byte offset, in the range -256 to 255.

Xt
Is the 64-bit name of the general-purpose register to be transferred.

pimm
Depends on the instruction variant:

32-bit general registers
Is the optional positive immediate byte offset, a multiple of 4 in the range 0 to 16380, defaulting to 0.

64-bit general registers
Is the optional positive immediate byte offset, a multiple of 8 in the range 0 to 32760, defaulting to 0.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Usage

Store Register (immediate) stores a word or a doubleword from a register to memory. The address that is used for the store is calculated from a base register and an immediate offset. For information about memory accesses, see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.104 STR (register)

Store Register (register).

Syntax

```
STR Wt, [Xn|SP, (Wm|Xm){, extend {amount}}] ; 32-bit

STR Xt, [Xn|SP, (Wm|Xm){, extend {amount}}] ; 64-bit
```

Where:

Wt
Is the 32-bit name of the general-purpose register to be transferred.

amount
Is the index shift amount, optional only when *extend* is not LSL. Where it is permitted to be optional, it defaults to #0. It is:

32-bit general registers
Can be one of `#0` or `#2`.

64-bit general registers
Can be one of `#0` or `#3`.

Xt
Is the 64-bit name of the general-purpose register to be transferred.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Wm
When "option<0>" is set to 0, is the 32-bit name of the general-purpose index register.

Xm
When "option<0>" is set to 1, is the 64-bit name of the general-purpose index register.

extend
Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when *amount* is omitted, and can be one of the values shown in Usage.

Usage

Store Register (register) calculates an address from a base register value and an offset register value, and stores a 32-bit word or a 64-bit doubleword to the calculated address, from a register. For information about memory accesses, see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

The instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an offset register value. The offset can be optionally shifted and extended.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.105 STRB (immediate)

Store Register Byte (immediate).

Syntax

```
STRB Wt, [Xn|SP], #simm ; Post-index general registers

STRB Wt, [Xn|SP, #simm]! ; Pre-index general registers

STRB Wt, [Xn|SP{, #pimm}] ; Unsigned offset general registers
```

Where:

simm
Is the signed immediate byte offset, in the range -256 to 255.

pimm
Is the optional positive immediate byte offset, in the range 0 to 4095, defaulting to 0.

Wt
Is the 32-bit name of the general-purpose register to be transferred.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Usage

Store Register Byte (immediate) stores the least significant byte of a 32-bit register to memory. The address that is used for the store is calculated from a base register and an immediate offset. For information about memory accesses, see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Note

For 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*, and particularly *STRB (immediate)*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.106 STRB (register)

Store Register Byte (register).

Syntax

```
STRB Wt, [Xn|SP, (Wm|Xm), extend {amount}] ; Extended register general registers

STRB Wt, [Xn|SP, Xm{, LSL amount}] ; Shifted register general registers
```

Where:

Wt
Is the 32-bit name of the general-purpose register to be transferred.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Wm
When "option<0>" is set to 0, is the 32-bit name of the general-purpose index register.

Xm
When "option<0>" is set to 1, is the 64-bit name of the general-purpose index register.

extend
Is the index extend specifier, and can be one of the values shown in Usage.

amount
Is the index shift amount, it must be.

Usage

Store Register Byte (register) calculates an address from a base register value and an offset register value, and stores a byte from a 32-bit register to the calculated address. For information about memory accesses, see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

The instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an offset register value. The offset can be optionally shifted and extended.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.107 STRH (immediate)

Store Register Halfword (immediate).

Syntax

```
STRH Wt, [Xn|SP], #simm ; Post-index general registers

STRH Wt, [Xn|SP, #simm]! ; Pre-index general registers

STRH Wt, [Xn|SP{, #pimm}] ; Unsigned offset general registers
```

Where:

simm
Is the signed immediate byte offset, in the range -256 to 255.

pimm
Is the optional positive immediate byte offset, a multiple of 2 in the range 0 to 8190, defaulting to 0.

Wt
Is the 32-bit name of the general-purpose register to be transferred.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Usage

Store Register Halfword (immediate) stores the least significant halfword of a 32-bit register to memory. The address that is used for the store is calculated from a base register and an immediate offset. For information about memory accesses, see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Note

For 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*, and particularly *STRH (immediate)*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.108 STRH (register)

Store Register Halfword (register).

Syntax

```
STRH Wt, [Xn|SP, (Wm|Xm){, extend {amount}}]
```

Where:

Wt
Is the 32-bit name of the general-purpose register to be transferred.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Wm
When "option<0>" is set to 0, is the 32-bit name of the general-purpose index register.

Xm
When "option<0>" is set to 1, is the 64-bit name of the general-purpose index register.

extend
Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when *amount* is omitted, and can be one of `UXTW`, `LSL`, `SXTW` or `SXTX`.

amount
Is the index shift amount, optional only when *extend* is not LSL. Where it is permitted to be optional, it defaults to #0. It is, and can be either `#0` or `#1`.

Usage

Store Register Halfword (register) calculates an address from a base register value and an offset register value, and stores a halfword from a 32-bit register to the calculated address. For information about memory accesses, see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

The instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an offset register value. The offset can be optionally shifted and extended.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.109 STSET, STSETL, STSETL

Atomic bit set on word or doubleword in memory, without return.

Syntax

```
STSET Ws, [Xn|SP] ; 32-bit, no memory ordering general registers

STSETL Ws, [Xn|SP] ; 32-bit, release general registers

STSET Xs, [Xn|SP] ; 64-bit, no memory ordering general registers

STSETL Xs, [Xn|SP] ; 64-bit, release general registers
```

Where:

Ws
Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Xs
Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location.

Architectures supported

Supported in the Armv8.1 architecture and later.

Usage

Atomic bit set on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory.

- `STSET` has no memory ordering semantics.
- `STSETL` stores to memory with release semantics, as described in *Load-Acquire, Store-Release* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.110 STSETB, STSETLB

Atomic bit set on byte in memory, without return.

Syntax

```
STSETB Ws, [Xn|SP] ; No memory ordering general registers

STSETLB Ws, [Xn|SP] ; Release general registers
```

Where:

Ws
Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Architectures supported

Supported in the Armv8.1 architecture and later.

Usage

Atomic bit set on byte in memory, without return, atomically loads an 8-bit byte from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory.

- `STSETB` has no memory ordering semantics.
- `STSETLB` stores to memory with release semantics, as described in *Load-Acquire, Store-Release* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.111 STSETH, STSETLH

Atomic bit set on halfword in memory, without return.

Syntax

```
STSETH Ws, [Xn|SP] ; No memory ordering general registers

STSETLH Ws, [Xn|SP] ; Release general registers
```

Where:

Ws
Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Architectures supported

Supported in the Armv8.1 architecture and later.

Usage

Atomic bit set on halfword in memory, without return, atomically loads a 16-bit halfword from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory.

- `STSETH` has no memory ordering semantics.
- `STSETLH` stores to memory with release semantics, as described in *Load-Acquire, Store-Release* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.112 STSMAX, STSMAXL, STSMAXL

Atomic signed maximum on word or doubleword in memory, without return.

Syntax

```
STSMAX Ws, [Xn|SP] ; 32-bit, no memory ordering general registers

STSMAXL Ws, [Xn|SP] ; 32-bit, release general registers

STSMAX Xs, [Xn|SP] ; 64-bit, no memory ordering general registers

STSMAXL Xs, [Xn|SP] ; 64-bit, release general registers
```

Where:

Ws
Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Xs
Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location.

Architectures supported

Supported in the Armv8.1 architecture and later.

Usage

Atomic signed maximum on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers.

- `STSMAX` has no memory ordering semantics.
- `STSMAXL` stores to memory with release semantics, as described in *Load-Acquire, Store-Release* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.113 STSMAXB, STSMAXLB

Atomic signed maximum on byte in memory, without return.

Syntax

```
STSMAXB Ws, [Xn|SP] ; No memory ordering general registers

STSMAXLB Ws, [Xn|SP] ; Release general registers
```

Where:

Ws
Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Architectures supported

Supported in the Armv8.1 architecture and later.

Usage

Atomic signed maximum on byte in memory, without return, atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers.

- `STSMAXB` has no memory ordering semantics.
- `STSMAXLB` stores to memory with release semantics, as described in *Load-Acquire, Store-Release* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.114 STSMAXH, STSMAXLH

Atomic signed maximum on halfword in memory, without return.

Syntax

```
STSMAXH Ws, [Xn|SP] ; No memory ordering general registers

STSMAXLH Ws, [Xn|SP] ; Release general registers
```

Where:

Ws
Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Architectures supported

Supported in the Armv8.1 architecture and later.

Usage

Atomic signed maximum on halfword in memory, without return, atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers.

- `STSMAXH` has no memory ordering semantics.
- `STSMAXLH` stores to memory with release semantics, as described in *Load-Acquire, Store-Release* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.115 STSMIN, STSMINL, STSMINL

Atomic signed minimum on word or doubleword in memory, without return.

Syntax

```
STSMIN Ws, [Xn|SP] ; 32-bit, no memory ordering general registers

STSMINL Ws, [Xn|SP] ; 32-bit, release general registers

STSMIN Xs, [Xn|SP] ; 64-bit, no memory ordering general registers

STSMINL Xs, [Xn|SP] ; 64-bit, release general registers
```

Where:

Ws
Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Xs
Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location.

Architectures supported

Supported in the Armv8.1 architecture and later.

Usage

Atomic signed minimum on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers.

- `STSMIN` has no memory ordering semantics.
- `STSMINL` stores to memory with release semantics, as described in *Load-Acquire, Store-Release* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.116 STSMINB, STSMINLB

Atomic signed minimum on byte in memory, without return.

Syntax

```
STSMINB Ws, [Xn|SP] ; No memory ordering general registers

STSMINLB Ws, [Xn|SP] ; Release general registers
```

Where:

Ws
Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Architectures supported

Supported in the Armv8.1 architecture and later.

Usage

Atomic signed minimum on byte in memory, without return, atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers.

- `STSMINB` has no memory ordering semantics.
- `STSMINLB` stores to memory with release semantics, as described in *Load-Acquire, Store-Release* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.117 STSMINH, STSMINLH

Atomic signed minimum on halfword in memory, without return.

Syntax

```
STSMINH Ws, [Xn|SP] ; No memory ordering general registers

STSMINLH Ws, [Xn|SP] ; Release general registers
```

Where:

Ws
Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Architectures supported

Supported in the Armv8.1 architecture and later.

Usage

Atomic signed minimum on halfword in memory, without return, atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers.

- `STSMINH` has no memory ordering semantics.
- `STSMINLH` stores to memory with release semantics, as described in *Load-Acquire, Store-Release* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.118 STTR

Store Register (unprivileged).

Syntax

```
STTR Wt, [Xn|SP{, #simm}] ; 32-bit
STTR Xt, [Xn|SP{, #simm}] ; 64-bit
```

Where:

Wt
Is the 32-bit name of the general-purpose register to be transferred.

Xt
Is the 64-bit name of the general-purpose register to be transferred.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

simm
Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0.

Usage

Store Register (unprivileged) stores a word or doubleword from a register to memory. The address that is used for the store is calculated from a base register and an immediate offset.

The memory is restricted as if execution is at EL0 when:

- Executing at EL1.
- Executing at EL2, in Armv8.1, with HCR_EL2.{E2H, TGE} set to {1, 1}.

Otherwise, the access permission is for the Exception level at which the instruction is executed. For information about memory accesses, see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.119 STTRB

Store Register Byte (unprivileged).

Syntax

```
STTRB Wt, [Xn|SP{, #simm}]
```

Where:

Wt
Is the 32-bit name of the general-purpose register to be transferred.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

simm
Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0.

Usage

Store Register Byte (unprivileged) stores a byte from a 32-bit register to memory. The address that is used for the store is calculated from a base register and an immediate offset.

The memory is restricted as if execution is at EL0 when:

- Executing at EL1.
- Executing at EL2, in Armv8.1, with HCR_EL2.{E2H, TGE} set to {1, 1}.

Otherwise, the access permission is for the Exception level at which the instruction is executed. For information about memory accesses, see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.120 STTRH

Store Register Halfword (unprivileged).

Syntax

```
STTRH Wt, [Xn|SP{, #simm}]
```

Where:

Wt
Is the 32-bit name of the general-purpose register to be transferred.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

simm
Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0.

Usage

Store Register Halfword (unprivileged) stores a halfword from a 32-bit register to memory. The address that is used for the store is calculated from a base register and an immediate offset.

The memory is restricted as if execution is at EL0 when:

- Executing at EL1.
- Executing at EL2, in Armv8.1, with HCR_EL2.{E2H, TGE} set to {1, 1}.

Otherwise, the access permission is for the Exception level at which the instruction is executed. For information about memory accesses, see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.121 STUMAX, STUMAXL, STUMAXL

Atomic unsigned maximum on word or doubleword in memory, without return.

Syntax

```
STUMAX Ws, [Xn|SP] ; 32-bit, no memory ordering general registers

STUMAXL Ws, [Xn|SP] ; 32-bit, release general registers

STUMAX Xs, [Xn|SP] ; 64-bit, no memory ordering general registers

STUMAXL Xs, [Xn|SP] ; 64-bit, release general registers
```

Where:

Ws
Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Xs
Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location.

Architectures supported

Supported in the Armv8.1 architecture and later.

Usage

Atomic unsigned maximum on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers.

- `STUMAX` has no memory ordering semantics.
- `STUMAXL` stores to memory with release semantics, as described in *Load-Acquire, Store-Release* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.122 STUMAXB, STUMAXLB

Atomic unsigned maximum on byte in memory, without return.

Syntax

```
STUMAXB Ws, [Xn|SP] ; No memory ordering general registers

STUMAXLB Ws, [Xn|SP] ; Release general registers
```

Where:

Ws
Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Architectures supported

Supported in the Armv8.1 architecture and later.

Usage

Atomic unsigned maximum on byte in memory, without return, atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers.

- `STUMAXB` has no memory ordering semantics.
- `STUMAXLB` stores to memory with release semantics, as described in *Load-Acquire, Store-Release* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.123 STUMAXH, STUMAXLH

Atomic unsigned maximum on halfword in memory, without return.

Syntax

```
STUMAXH Ws, [Xn|SP] ; No memory ordering general registers

STUMAXLH Ws, [Xn|SP] ; Release general registers
```

Where:

Ws
Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Architectures supported

Supported in the Armv8.1 architecture and later.

Usage

Atomic unsigned maximum on halfword in memory, without return, atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers.

- `STUMAXH` has no memory ordering semantics.
- `STUMAXLH` stores to memory with release semantics, as described in *Load-Acquire, Store-Release* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.124 STUMIN, STUMINL, STUMINL

Atomic unsigned minimum on word or doubleword in memory, without return.

Syntax

```
STUMIN Ws, [Xn|SP] ; 32-bit, no memory ordering general registers

STUMINL Ws, [Xn|SP] ; 32-bit, release general registers

STUMIN Xs, [Xn|SP] ; 64-bit, no memory ordering general registers

STUMINL Xs, [Xn|SP] ; 64-bit, release general registers
```

Where:

Ws
Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Xs
Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location.

Architectures supported

Supported in the Armv8.1 architecture and later.

Usage

Atomic unsigned minimum on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers.

- `STUMIN` has no memory ordering semantics.
- `STUMINL` stores to memory with release semantics, as described in *Load-Acquire, Store-Release* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.125 STUMINB, STUMINLB

Atomic unsigned minimum on byte in memory, without return.

Syntax

```
STUMINB Ws, [Xn|SP] ; No memory ordering general registers

STUMINLB Ws, [Xn|SP] ; Release general registers
```

Where:

Ws
Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Architectures supported

Supported in the Armv8.1 architecture and later.

Usage

Atomic unsigned minimum on byte in memory, without return, atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers.

- `STUMINB` has no memory ordering semantics.
- `STUMINLB` stores to memory with release semantics, as described in *Load-Acquire, Store-Release* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.126 STUMINH, STUMINLH

Atomic unsigned minimum on halfword in memory, without return.

Syntax

```
STUMINH Ws, [Xn|SP] ; No memory ordering general registers

STUMINLH Ws, [Xn|SP] ; Release general registers
```

Where:

Ws
Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Architectures supported

Supported in the Armv8.1 architecture and later.

Usage

Atomic unsigned minimum on halfword in memory, without return, atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers.

- `STUMINH` has no memory ordering semantics.
- `STUMINLH` stores to memory with release semantics, as described in *Load-Acquire, Store-Release* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.127 STUR

Store Register (unscaled).

Syntax

```
STUR Wt, [Xn|SP{, #simm}] ; 32-bit
STUR Xt, [Xn|SP{, #simm}] ; 64-bit
```

Where:

Wt
Is the 32-bit name of the general-purpose register to be transferred.

Xt
Is the 64-bit name of the general-purpose register to be transferred.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

simm
Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0.

Usage

Store Register (unscaled) calculates an address from a base register value and an immediate offset, and stores a 32-bit word or a 64-bit doubleword to the calculated address, from a register. For information about memory accesses, see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.128 STURB

Store Register Byte (unscaled).

Syntax

```
STURB Wt, [Xn|SP{, #simm}]
```

Where:

Wt
Is the 32-bit name of the general-purpose register to be transferred.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

simm
Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0.

Usage

Store Register Byte (unscaled) calculates an address from a base register value and an immediate offset, and stores a byte to the calculated address, from a 32-bit register. For information about memory accesses, see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.129 STURH

Store Register Halfword (unscaled).

Syntax

```
STURH Wt, [Xn|SP{, #simm}]
```

Where:

Wt
Is the 32-bit name of the general-purpose register to be transferred.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

simm
Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0.

Usage

Store Register Halfword (unscaled) calculates an address from a base register value and an immediate offset, and stores a halfword to the calculated address, from a 32-bit register. For information about memory accesses, see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.130 STXP

Store Exclusive Pair of registers.

Syntax

```
STXP Ws, Wt1, Wt2, [Xn|SP{,#0}] ; 32-bit

STXP Ws, Xt1, Xt2, [Xn|SP{,#0}] ; 64-bit
```

Where:

Wt1
Is the 32-bit name of the first general-purpose register to be transferred.

Wt2
Is the 32-bit name of the second general-purpose register to be transferred.

Xt1
Is the 64-bit name of the first general-purpose register to be transferred.

Xt2
Is the 64-bit name of the second general-purpose register to be transferred.

Ws
Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written. The value returned is.

0
If the operation updates memory.

1
If the operation fails to update memory.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- *Ws* is not updated.

Accessing an address that is not aligned to the size of the data being accessed causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- The exception is generated if the Exclusive Monitors for the current PE include all of the addresses associated with the virtual address region of size bytes starting at address. The immediately following memory write must be to the same addresses.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

Whether the detection of memory aborts happens before or after the check on the local Exclusive Monitor depends on the implementation. As a result a failure of the local monitor can occur on some implementations even if the memory access would give a memory abort.

Usage

Store Exclusive Pair of registers stores two 32-bit words or two 64-bit doublewords from two registers to a memory location if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See *Synchronization and semaphores* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*. A 32-bit pair requires the address to be doubleword aligned and is single-copy atomic at doubleword granularity. A 64-bit pair requires the address to be quadword aligned and, if the Store-Exclusive succeeds, it causes a single-copy atomic update of the 128-bit memory location being updated. For information about memory accesses

see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Note

For 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*, and particularly *STXP*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.131 STXR

Store Exclusive Register.

Syntax

```
STXR Ws, Wt, [Xn|SP{,#0}] ; 32-bit
STXR Ws, Xt, [Xn|SP{,#0}] ; 64-bit
```

Where:

Wt
Is the 32-bit name of the general-purpose register to be transferred.

Xt
Is the 64-bit name of the general-purpose register to be transferred.

Ws
Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written. The value returned is.

0
If the operation updates memory.

1
If the operation fails to update memory.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- *Ws* is not updated.

Accessing an address that is not aligned to the size of the data being accessed causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- The exception is generated if the Exclusive Monitors for the current PE include all of the addresses associated with the virtual address region of size bytes starting at address. The immediately following memory write must be to the same addresses.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

Whether the detection of memory aborts happens before or after the check on the local Exclusive Monitor depends on the implementation. As a result a failure of the local monitor can occur on some implementations even if the memory access would give a memory abort.

Usage

Store Exclusive Register stores a 32-bit word or a 64-bit doubleword from a register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See *Synchronization and semaphores* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*. For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Note

For 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*, and particularly *STXR*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.132 STXRB

Store Exclusive Register Byte.

Syntax

`STXRB Ws, Wt, [Xn|SP{,#0}]`

Where:

Ws
Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written. The value returned is.

0
If the operation updates memory.

1
If the operation fails to update memory.

Wt
Is the 32-bit name of the general-purpose register to be transferred.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Aborts

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- *Ws* is not updated.

Whether the detection of memory aborts happens before or after the check on the local Exclusive Monitor depends on the implementation. As a result a failure of the local monitor can occur on some implementations even if the memory access would give a memory abort.

Usage

Store Exclusive Register Byte stores a byte from a register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See *Synchronization and semaphores* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*. The memory access is atomic.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Note

For 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*, and particularly *STXRB*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.133 STXRH

Store Exclusive Register Halfword.

Syntax

```
STXRH Ws, Wt, [Xn|SP{,#0}]
```

Where:

Ws
Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written. The value returned is.

0
If the operation updates memory.

1
If the operation fails to update memory.

Wt
Is the 32-bit name of the general-purpose register to be transferred.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- *Ws* is not updated.

A non halfword-aligned memory address causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- The exception is generated if the Exclusive Monitors for the current PE include all of the addresses associated with the virtual address region of size bytes starting at address. The immediately following memory write must be to the same addresses.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

Whether the detection of memory aborts happens before or after the check on the local Exclusive Monitor depends on the implementation. As a result a failure of the local monitor can occur on some implementations even if the memory access would give a memory abort.

Usage

Store Exclusive Register Halfword stores a halfword from a register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See *Synchronization and semaphores* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*. The memory access is atomic.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.134 SWPA, SWPAL, SWP, SWPL, SWPAL, SWP, SWPL

Swap word or doubleword in memory.

Syntax

```
SWPA Ws, Wt, [Xn|SP] ; 32-bit, acquire general registers

SWPAL Ws, Wt, [Xn|SP] ; 32-bit, acquire and release general registers

SWP Ws, Wt, [Xn|SP] ; 32-bit, no memory ordering general registers

SWPL Ws, Wt, [Xn|SP] ; 32-bit, release general registers

SWPA Xs, Xt, [Xn|SP] ; 64-bit, acquire general registers

SWPAL Xs, Xt, [Xn|SP] ; 64-bit, acquire and release general registers

SWP Xs, Xt, [Xn|SP] ; 64-bit, no memory ordering general registers

SWPL Xs, Xt, [Xn|SP] ; 64-bit, release general registers
```

Where:

Ws
Is the 32-bit name of the general-purpose register to be stored.

Wt
Is the 32-bit name of the general-purpose register to be loaded.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Xs
Is the 64-bit name of the general-purpose register to be stored.

Xt
Is the 64-bit name of the general-purpose register to be loaded.

Architectures supported

Supported in the Armv8.1 architecture and later.

Usage

Swap word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from a memory location, and stores the value held in a register back to the same memory location. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of `WZR` or `XZR`, `SWPA` and `SWPAL` load from memory with acquire semantics.
- `SWPL` and `SWPAL` store to memory with release semantics.
- `SWP` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.135 SWPAB, SWPALB, SWPB, SWPLB

Swap byte in memory.

Syntax

```
SWPAB Ws, Wt, [Xn|SP] ; Acquire general registers

SWPALB Ws, Wt, [Xn|SP] ; Acquire and release general registers

SWPB Ws, Wt, [Xn|SP] ; No memory ordering general registers

SWPLB Ws, Wt, [Xn|SP] ; Release general registers
```

Where:

Ws
Is the 32-bit name of the general-purpose register to be stored.

Wt
Is the 32-bit name of the general-purpose register to be loaded.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Architectures supported

Supported in the Armv8.1 architecture and later.

Usage

Swap byte in memory atomically loads an 8-bit byte from a memory location, and stores the value held in a register back to the same memory location. The value initially loaded from memory is returned in the destination register.

- If the destination register is not `WZR`, `SWPAB` and `SWPALB` load from memory with acquire semantics.
- `SWPLB` and `SWPALB` store to memory with release semantics.
- `SWPB` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

17.136 SWPAH, SWPALH, SWPH, SWPLH

Swap halfword in memory.

Syntax

```
SWPAH Ws, Wt, [Xn|SP] ; Acquire general registers

SWPALH Ws, Wt, [Xn|SP] ; Acquire and release general registers

SWPH Ws, Wt, [Xn|SP] ; No memory ordering general registers

SWPLH Ws, Wt, [Xn|SP] ; Release general registers
```

Where:

Ws
Is the 32-bit name of the general-purpose register to be stored.

Wt
Is the 32-bit name of the general-purpose register to be loaded.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Architectures supported

Supported in the Armv8.1 architecture and later.

Usage

Swap halfword in memory atomically loads a 16-bit halfword from a memory location, and stores the value held in a register back to the same memory location. The value initially loaded from memory is returned in the destination register.

- If the destination register is not `WZR`, `SWPAH` and `SWPALH` load from memory with acquire semantics.
- `SWPLH` and `SWPALH` store to memory with release semantics.
- `SWPH` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

For information about memory accesses see *Load/Store addressing modes* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

Chapter 18
A64 Floating-point Instructions

Describes the A64 floating-point instructions.

It contains the following sections:

18.1 A64 floating-point instructions in alphabetical order

A summary of the A64 floating-point instructions that are supported.

Table 18-1 Summary of A64 floating-point instructions

Mnemonic	Brief description	See
FABS (scalar)	Floating-point Absolute value (scalar)	*18.2 FABS (scalar)* on page 18-1146
FADD (scalar)	Floating-point Add (scalar)	*18.3 FADD (scalar)* on page 18-1147
FCCMP	Floating-point Conditional quiet Compare (scalar)	*18.4 FCCMP* on page 18-1148
FCCMPE	Floating-point Conditional signaling Compare (scalar)	*18.5 FCCMPE* on page 18-1149
FCMP	Floating-point quiet Compare (scalar)	*18.6 FCMP* on page 18-1151
FCMPE	Floating-point signaling Compare (scalar)	*18.7 FCMPE* on page 18-1153
FCSEL	Floating-point Conditional Select (scalar)	*18.8 FCSEL* on page 18-1155
FCVT	Floating-point Convert precision (scalar)	*18.9 FCVT* on page 18-1156
FCVTAS (scalar)	Floating-point Convert to Signed integer, rounding to nearest with ties to Away (scalar)	*18.10 FCVTAS (scalar)* on page 18-1157
FCVTAU (scalar)	Floating-point Convert to Unsigned integer, rounding to nearest with ties to Away (scalar)	*18.11 FCVTAU (scalar)* on page 18-1158
FCVTMS (scalar)	Floating-point Convert to Signed integer, rounding toward Minus infinity (scalar)	*18.12 FCVTMS (scalar)* on page 18-1159
FCVTMU (scalar)	Floating-point Convert to Unsigned integer, rounding toward Minus infinity (scalar)	*18.13 FCVTMU (scalar)* on page 18-1160
FCVTNS (scalar)	Floating-point Convert to Signed integer, rounding to nearest with ties to even (scalar)	*18.14 FCVTNS (scalar)* on page 18-1161
FCVTNU (scalar)	Floating-point Convert to Unsigned integer, rounding to nearest with ties to even (scalar)	*18.15 FCVTNU (scalar)* on page 18-1162
FCVTPS (scalar)	Floating-point Convert to Signed integer, rounding toward Plus infinity (scalar)	*18.16 FCVTPS (scalar)* on page 18-1163
FCVTPU (scalar)	Floating-point Convert to Unsigned integer, rounding toward Plus infinity (scalar)	*18.17 FCVTPU (scalar)* on page 18-1164
FCVTZS (scalar, fixed-point)	Floating-point Convert to Signed fixed-point, rounding toward Zero (scalar)	*18.18 FCVTZS (scalar, fixed-point)* on page 18-1165
FCVTZS (scalar, integer)	Floating-point Convert to Signed integer, rounding toward Zero (scalar)	*18.19 FCVTZS (scalar, integer)* on page 18-1166
FCVTZU (scalar, fixed-point)	Floating-point Convert to Unsigned fixed-point, rounding toward Zero (scalar)	*18.20 FCVTZU (scalar, fixed-point)* on page 18-1167
FCVTZU (scalar, integer)	Floating-point Convert to Unsigned integer, rounding toward Zero (scalar)	*18.21 FCVTZU (scalar, integer)* on page 18-1168
FDIV (scalar)	Floating-point Divide (scalar)	*18.22 FDIV (scalar)* on page 18-1169
FJCVTZS	Floating-point Javascript Convert to Signed fixed-point, rounding toward Zero	*18.23 FJCVTZS* on page 18-1170

Table 18-1 Summary of A64 floating-point instructions (continued)

Mnemonic	Brief description	See
FMADD	Floating-point fused Multiply-Add (scalar)	*18.24 FMADD* on page 18-1171
FMAX (scalar)	Floating-point Maximum (scalar)	*18.25 FMAX (scalar)* on page 18-1172
FMAXNM (scalar)	Floating-point Maximum Number (scalar)	*18.26 FMAXNM (scalar)* on page 18-1173
FMIN (scalar)	Floating-point Minimum (scalar)	*18.27 FMIN (scalar)* on page 18-1174
FMINNM (scalar)	Floating-point Minimum Number (scalar)	*18.28 FMINNM (scalar)* on page 18-1175
FMOV (register)	Floating-point Move register without conversion	*18.29 FMOV (register)* on page 18-1176
FMOV (general)	Floating-point Move to or from general-purpose register without conversion	*18.30 FMOV (general)* on page 18-1177
FMOV (scalar, immediate)	Floating-point move immediate (scalar)	*18.31 FMOV (scalar, immediate)* on page 18-1178
FMSUB	Floating-point Fused Multiply-Subtract (scalar)	*18.32 FMSUB* on page 18-1179
FMUL (scalar)	Floating-point Multiply (scalar)	*18.33 FMUL (scalar)* on page 18-1180
FNEG (scalar)	Floating-point Negate (scalar)	*18.34 FNEG (scalar)* on page 18-1181
FNMADD	Floating-point Negated fused Multiply-Add (scalar)	*18.35 FNMADD* on page 18-1182
FNMSUB	Floating-point Negated fused Multiply-Subtract (scalar)	*18.36 FNMSUB* on page 18-1183
FNMUL (scalar)	Floating-point Multiply-Negate (scalar)	*18.37 FNMUL (scalar)* on page 18-1184
FRINTA (scalar)	Floating-point Round to Integral, to nearest with ties to Away (scalar)	*18.38 FRINTA (scalar)* on page 18-1185
FRINTI (scalar)	Floating-point Round to Integral, using current rounding mode (scalar)	*18.39 FRINTI (scalar)* on page 18-1186
FRINTM (scalar)	Floating-point Round to Integral, toward Minus infinity (scalar)	*18.40 FRINTM (scalar)* on page 18-1187
FRINTN (scalar)	Floating-point Round to Integral, to nearest with ties to even (scalar)	*18.41 FRINTN (scalar)* on page 18-1188
FRINTP (scalar)	Floating-point Round to Integral, toward Plus infinity (scalar)	*18.42 FRINTP (scalar)* on page 18-1189
FRINTX (scalar)	Floating-point Round to Integral exact, using current rounding mode (scalar)	*18.43 FRINTX (scalar)* on page 18-1190
FRINTZ (scalar)	Floating-point Round to Integral, toward Zero (scalar)	*18.44 FRINTZ (scalar)* on page 18-1191
FSQRT (scalar)	Floating-point Square Root (scalar)	*18.45 FSQRT (scalar)* on page 18-1192
FSUB (scalar)	Floating-point Subtract (scalar)	*18.46 FSUB (scalar)* on page 18-1193
LDNP (SIMD and FP)	Load Pair of SIMD and FP registers, with Non-temporal hint	*18.47 LDNP (SIMD and FP)* on page 18-1194
LDP (SIMD and FP)	Load Pair of SIMD and FP registers	*18.48 LDP (SIMD and FP)* on page 18-1195
LDR (immediate, SIMD and FP)	Load SIMD and FP Register (immediate offset)	*18.49 LDR (immediate, SIMD and FP)* on page 18-1197

Table 18-1 Summary of A64 floating-point instructions (continued)

Mnemonic	Brief description	See
`LDR` (literal, SIMD and FP)	Load SIMD and FP Register (PC-relative literal)	*18.50 LDR (literal, SIMD and FP)* on page 18-1199
`LDR` (register, SIMD and FP)	Load SIMD and FP Register (register offset)	*18.51 LDR (register, SIMD and FP)* on page 18-1200
`LDUR` (SIMD and FP)	Load SIMD and FP Register (unscaled offset)	*18.52 LDUR (SIMD and FP)* on page 18-1201
`SCVTF` (scalar, fixed-point)	Signed fixed-point Convert to Floating-point (scalar)	*18.53 SCVTF (scalar, fixed-point)* on page 18-1202
`SCVTF` (scalar, integer)	Signed integer Convert to Floating-point (scalar)	*18.54 SCVTF (scalar, integer)* on page 18-1203
`STNP` (SIMD and FP)	Store Pair of SIMD and FP registers, with Non-temporal hint	*18.55 STNP (SIMD and FP)* on page 18-1204
`STP` (SIMD and FP)	Store Pair of SIMD and FP registers	*18.56 STP (SIMD and FP)* on page 18-1205
`STR` (immediate, SIMD and FP)	Store SIMD and FP register (immediate offset)	*18.57 STR (immediate, SIMD and FP)* on page 18-1206
`STR` (register, SIMD and FP)	Store SIMD and FP register (register offset)	*18.58 STR (register, SIMD and FP)* on page 18-1208
`STUR` (SIMD and FP)	Store SIMD and FP register (unscaled offset)	*18.59 STUR (SIMD and FP)* on page 18-1209
`UCVTF` (scalar, fixed-point)	Unsigned fixed-point Convert to Floating-point (scalar)	*18.60 UCVTF (scalar, fixed-point)* on page 18-1210
`UCVTF` (scalar, integer)	Unsigned integer Convert to Floating-point (scalar)	*18.61 UCVTF (scalar, integer)* on page 18-1211

18.2 FABS (scalar)

Floating-point Absolute value (scalar).

Syntax

```
FABS Hd, Hn ; Half-precision

FABS Sd, Sn ; Single-precision

FABS Dd, Dn ; Double-precision
```

Where:

Hd
Is the 16-bit name of the SIMD and FP destination register.

Hn
Is the 16-bit name of the SIMD and FP source register.

Sd
Is the 32-bit name of the SIMD and FP destination register.

Sn
Is the 32-bit name of the SIMD and FP source register.

Dd
Is the 64-bit name of the SIMD and FP destination register.

Dn
Is the 64-bit name of the SIMD and FP source register.

Operation

Floating-point Absolute value (scalar). This instruction calculates the absolute value in the SIMD and FP source register and writes the result to the SIMD and FP destination register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
Vd = abs(Vn).
```

Related references

18.1 A64 floating-point instructions in alphabetical order on page 18-1143.

18.3 FADD (scalar)

Floating-point Add (scalar).

Syntax

```
FADD Hd, Hn, Hm ; Half-precision

FADD Sd, Sn, Sm ; Single-precision

FADD Dd, Dn, Dm ; Double-precision
```

Where:

Hd
Is the 16-bit name of the SIMD and FP destination register.

Hn
Is the 16-bit name of the first SIMD and FP source register.

Hm
Is the 16-bit name of the second SIMD and FP source register.

Sd
Is the 32-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.

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.

Operation

Floating-point Add (scalar). This instruction adds the floating-point values of the two source SIMD and FP registers, and writes the result to the destination SIMD and FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
Vd = Vn + Vm.
```

Related references

18.1 A64 floating-point instructions in alphabetical order on page 18-1143.

18.4 FCCMP

Floating-point Conditional quiet Compare (scalar).

Syntax

```
FCCMP Hn, Hm, #nzcv, cond ; Half-precision

FCCMP Sn, Sm, #nzcv, cond ; Single-precision

FCCMP Dn, Dm, #nzcv, cond ; Double-precision
```

Where:

Hn
Is the 16-bit name of the first SIMD and FP source register.

Hm
Is the 16-bit name of the second SIMD and FP source 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.

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.

nzcv
Is the flag bit specifier, an immediate in the range 0 to 15, giving the alternative state for the 4-bit NZCV condition flags.

cond
Is one of the standard conditions.

NaNs

The IEEE 754 standard specifies that the result of a comparison is precisely one of <, ==, > or unordered. If either or both of the operands are NaNs, they are unordered, and all three of (Operand1 < Operand2), (Operand1 == Operand2) and (Operand1 > Operand2) are false. This case results in the *FPSCR* flags being set to N=0, Z=0, C=1, and V=1.

Operation

Floating-point Conditional quiet Compare (scalar). This instruction compares the two SIMD and FP source register values and writes the result to the PSTATE.{N, Z, C, V} flags. If the condition does not pass then the PSTATE.{N, Z, C, V} flags are set to the flag bit specifier.

It raises an Invalid Operation exception only if either operand is a signaling NaN.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
flags = if cond then compareQuiet(Vn,Vm) else #nzcv.
```

Related references

7.12 Condition code suffixes and related flags on page 7-151.
18.1 A64 floating-point instructions in alphabetical order on page 18-1143.

18.5 FCCMPE

Floating-point Conditional signaling Compare (scalar).

Syntax

```
FCCMPE Hn, Hm, #nzcv, cond ; Half-precision

FCCMPE Sn, Sm, #nzcv, cond ; Single-precision

FCCMPE Dn, Dm, #nzcv, cond ; Double-precision
```

Where:

Hn
Is the 16-bit name of the first SIMD and FP source register.

Hm
Is the 16-bit name of the second SIMD and FP source 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.

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.

nzcv
Is the flag bit specifier, an immediate in the range 0 to 15, giving the alternative state for the 4-bit NZCV condition flags.

cond
Is one of the standard conditions.

NaNs

The IEEE 754 standard specifies that the result of a comparison is precisely one of <, ==, > or unordered. If either or both of the operands are NaNs, they are unordered, and all three of (Operand1 < Operand2), (Operand1 == Operand2) and (Operand1 > Operand2) are false. This case results in the *FPSCR* flags being set to N=0, Z=0, C=1, and V=1.

`FCCMPE` raises an Invalid Operation exception if either operand is any type of NaN, and is suitable for testing for <, <=, >, >=, and other predicates that raise an exception when the operands are unordered.

Operation

Floating-point Conditional signaling Compare (scalar). This instruction compares the two SIMD and FP source register values and writes the result to the PSTATE.{N, Z, C, V} flags. If the condition does not pass then the PSTATE.{N, Z, C, V} flags are set to the flag bit specifier.

If either operand is any type of NaN, or if either operand is a signaling NaN, the instruction raises an Invalid Operation exception.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
flags = if cond then compareSignaling(Vn,Vm) else #nzcv.
```

Related references

7.12 Condition code suffixes and related flags on page 7-151.

18.1 A64 floating-point instructions in alphabetical order on page 18-1143.

18.6 FCMP

Floating-point quiet Compare (scalar).

Syntax

```
FCMP Hn, Hm ; Half-precision

FCMP Hn, #0.0 ; Half-precision, zero

FCMP Sn, Sm ; Single-precision

FCMP Sn, #0.0 ; Single-precision, zero

FCMP Dn, Dm ; Double-precision

FCMP Dn, #0.0 ; Double-precision, zero
```

Where:

Hn

Depends on the instruction variant:

Half-precision
Is the 16-bit name of the first SIMD and FP source register

Half-precision, zero
Is the 16-bit name of the SIMD and FP source register

Hm

Is the 16-bit name of the second SIMD and FP source register.

Sn

Depends on the instruction variant:

Single-precision
Is the 32-bit name of the first SIMD and FP source register.

Single-precision, zero
Is the 32-bit name of the SIMD and FP source register.

Sm

Is the 32-bit name of the second SIMD and FP source register.

Dn

Depends on the instruction variant:

Double-precision
Is the 64-bit name of the first SIMD and FP source register.

Double-precision, zero
Is the 64-bit name of the SIMD and FP source register.

Dm

Is the 64-bit name of the second SIMD and FP source register.

NaNs

The IEEE 754 standard specifies that the result of a comparison is precisely one of <, ==, > or unordered. If either or both of the operands are NaNs, they are unordered, and all three of (Operand1 < Operand2), (Operand1 == Operand2) and (Operand1 > Operand2) are false. This case results in the *FPSCR* flags being set to N=0, Z=0, C=1, and V=1.

Usage

Floating-point quiet Compare (scalar). This instruction compares the two SIMD and FP source register values, or the first SIMD and FP source register value and zero. It writes the result to the PSTATE.{N, Z, C, V} flags.

It raises an Invalid Operation exception only if either operand is a signaling NaN.

This instruction can generate a floating-point exception. Depending on the settings in FPCR in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*, the exception results in either a flag being set in FPSR in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

18.1 A64 floating-point instructions in alphabetical order on page 18-1143.

18.7 FCMPE

Floating-point signaling Compare (scalar).

Syntax

```
FCMPE Hn, Hm ; Half-precision

FCMPE Hn, #0.0 ; Half-precision, zero

FCMPE Sn, Sm ; Single-precision

FCMPE Sn, #0.0 ; Single-precision, zero

FCMPE Dn, Dm ; Double-precision

FCMPE Dn, #0.0 ; Double-precision, zero
```

Where:

Hn

Depends on the instruction variant:

Half-precision
Is the 16-bit name of the first SIMD and FP source register

Half-precision, zero
Is the 16-bit name of the SIMD and FP source register

Hm

Is the 16-bit name of the second SIMD and FP source register.

Sn

Depends on the instruction variant:

Single-precision
Is the 32-bit name of the first SIMD and FP source register.

Single-precision, zero
Is the 32-bit name of the SIMD and FP source register.

Sm

Is the 32-bit name of the second SIMD and FP source register.

Dn

Depends on the instruction variant:

Double-precision
Is the 64-bit name of the first SIMD and FP source register.

Double-precision, zero
Is the 64-bit name of the SIMD and FP source register.

Dm

Is the 64-bit name of the second SIMD and FP source register.

NaNs

The IEEE 754 standard specifies that the result of a comparison is precisely one of <, ==, > or unordered. If either or both of the operands are NaNs, they are unordered, and all three of (Operand1 < Operand2), (Operand1 == Operand2) and (Operand1 > Operand2) are false. This case results in the *FPSCR* flags being set to N=0, Z=0, C=1, and V=1.

`FCMPE` raises an Invalid Operation exception if either operand is any type of NaN, and is suitable for testing for <, <=, >, >=, and other predicates that raise an exception when the operands are unordered.

Usage

Floating-point signaling Compare (scalar). This instruction compares the two SIMD and FP source register values, or the first SIMD and FP source register value and zero. It writes the result to the PSTATE.{N, Z, C, V} flags.

If either operand is any type of NaN, or if either operand is a signaling NaN, the instruction raises an Invalid Operation exception.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

18.1 A64 floating-point instructions in alphabetical order on page 18-1143.

18.8 FCSEL

Floating-point Conditional Select (scalar).

Syntax

```
FCSEL Hd, Hn, Hm, cond ; Half-precision

FCSEL Sd, Sn, Sm, cond ; Single-precision

FCSEL Dd, Dn, Dm, cond ; Double-precision
```

Where:

Hd
Is the 16-bit name of the SIMD and FP destination register.

Hn
Is the 16-bit name of the first SIMD and FP source register.

Hm
Is the 16-bit name of the second SIMD and FP source register.

Sd
Is the 32-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.

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.

cond
Is one of the standard conditions.

Operation

Floating-point Conditional Select (scalar). This instruction allows the SIMD and FP destination register to take the value from either one or the other of two SIMD and FP source registers. If the condition passes, the first SIMD and FP source register value is taken, otherwise the second SIMD and FP source register value is taken.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
Vd = if cond then Vn else Vm.
```

Related references

7.12 Condition code suffixes and related flags on page 7-151.
18.1 A64 floating-point instructions in alphabetical order on page 18-1143.

18.9 FCVT

Floating-point Convert precision (scalar).

Syntax

```
FCVT Sd, Hn ; Half-precision to single-precision
FCVT Dd, Hn ; Half-precision to double-precision
FCVT Hd, Sn ; Single-precision to half-precision
FCVT Dd, Sn ; Single-precision to double-precision
FCVT Hd, Dn ; Double-precision to half-precision
FCVT Sd, Dn ; Double-precision to single-precision
```

Where:

Sd
Is the 32-bit name of the SIMD and FP destination register.

Hn
Is the 16-bit name of the SIMD and FP source register.

Dd
Is the 64-bit name of the SIMD and FP destination register.

Hd
Is the 16-bit name of the SIMD and FP destination register.

Sn
Is the 32-bit name of the SIMD and FP source register.

Dn
Is the 64-bit name of the SIMD and FP source register.

Operation

Floating-point Convert precision (scalar). This instruction converts the floating-point value in the SIMD and FP source register to the precision for the destination register data type using the rounding mode that is determined by the FPCR and writes the result to the SIMD and FP destination register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

`Vd = convertFormat(Vn)`, where *V* is `D`, `H`, or `S`.

Related references

18.1 A64 floating-point instructions in alphabetical order on page 18-1143.

18.10 FCVTAS (scalar)

Floating-point Convert to Signed integer, rounding to nearest with ties to Away (scalar).

Syntax

```
FCVTAS Wd, Hn ; Half-precision to 32-bit
FCVTAS Xd, Hn ; Half-precision to 64-bit
FCVTAS Wd, Sn ; Single-precision to 32-bit
FCVTAS Xd, Sn ; Single-precision to 64-bit
FCVTAS Wd, Dn ; Double-precision to 32-bit
FCVTAS Xd, Dn ; Double-precision to 64-bit
```

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

Hn
Is the 16-bit name of the SIMD and FP source register.

Xd
Is the 64-bit name of the general-purpose destination register.

Sn
Is the 32-bit name of the SIMD and FP source register.

Dn
Is the 64-bit name of the SIMD and FP source register.

Operation

Floating-point Convert to Signed integer, rounding to nearest with ties to Away (scalar). This instruction converts the floating-point value in the SIMD and FP source register to a 32-bit or 64-bit signed integer using the Round to Nearest with Ties to Away rounding mode, and writes the result to the general-purpose destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

`Rd = signed_convertToIntegerExactTiesToAway(Vn)`, where `R` is either `W` or `X`.

Related references

18.1 A64 floating-point instructions in alphabetical order on page 18-1143.

18.11 FCVTAU (scalar)

Floating-point Convert to Unsigned integer, rounding to nearest with ties to Away (scalar).

Syntax

```
FCVTAU Wd, Hn ; Half-precision to 32-bit
FCVTAU Xd, Hn ; Half-precision to 64-bit
FCVTAU Wd, Sn ; Single-precision to 32-bit
FCVTAU Xd, Sn ; Single-precision to 64-bit
FCVTAU Wd, Dn ; Double-precision to 32-bit
FCVTAU Xd, Dn ; Double-precision to 64-bit
```

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

Hn
Is the 16-bit name of the SIMD and FP source register.

Xd
Is the 64-bit name of the general-purpose destination register.

Sn
Is the 32-bit name of the SIMD and FP source register.

Dn
Is the 64-bit name of the SIMD and FP source register.

Operation

Floating-point Convert to Unsigned integer, rounding to nearest with ties to Away (scalar). This instruction converts the floating-point value in the SIMD and FP source register to a 32-bit or 64-bit unsigned integer using the Round to Nearest with Ties to Away rounding mode, and writes the result to the general-purpose destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

`Rd = unsigned_convertToIntegerExactTiesToAway(Vn)`, where `R` is either `W` or `X`.

Related references

18.1 A64 floating-point instructions in alphabetical order on page 18-1143.

18.12 FCVTMS (scalar)

Floating-point Convert to Signed integer, rounding toward Minus infinity (scalar).

Syntax

```
FCVTMS Wd, Hn ; Half-precision to 32-bit

FCVTMS Xd, Hn ; Half-precision to 64-bit

FCVTMS Wd, Sn ; Single-precision to 32-bit

FCVTMS Xd, Sn ; Single-precision to 64-bit

FCVTMS Wd, Dn ; Double-precision to 32-bit

FCVTMS Xd, Dn ; Double-precision to 64-bit
```

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

Hn
Is the 16-bit name of the SIMD and FP source register.

Xd
Is the 64-bit name of the general-purpose destination register.

Sn
Is the 32-bit name of the SIMD and FP source register.

Dn
Is the 64-bit name of the SIMD and FP source register.

Operation

Floating-point Convert to Signed integer, rounding toward Minus infinity (scalar). This instruction converts the floating-point value in the SIMD and FP source register to a 32-bit or 64-bit signed integer using the Round towards Minus Infinity rounding mode, and writes the result to the general-purpose destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

`Rd = signed_convertToIntegerExactTowardNegative(Vn)`, where *R* is either `W` or `X`.

Related references

18.1 A64 floating-point instructions in alphabetical order on page 18-1143.

18.13 FCVTMU (scalar)

Floating-point Convert to Unsigned integer, rounding toward Minus infinity (scalar).

Syntax

```
FCVTMU Wd, Hn ; Half-precision to 32-bit
FCVTMU Xd, Hn ; Half-precision to 64-bit
FCVTMU Wd, Sn ; Single-precision to 32-bit
FCVTMU Xd, Sn ; Single-precision to 64-bit
FCVTMU Wd, Dn ; Double-precision to 32-bit
FCVTMU Xd, Dn ; Double-precision to 64-bit
```

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

Hn
Is the 16-bit name of the SIMD and FP source register.

Xd
Is the 64-bit name of the general-purpose destination register.

Sn
Is the 32-bit name of the SIMD and FP source register.

Dn
Is the 64-bit name of the SIMD and FP source register.

Operation

Floating-point Convert to Unsigned integer, rounding toward Minus infinity (scalar). This instruction converts the floating-point value in the SIMD and FP source register to a 32-bit or 64-bit unsigned integer using the Round towards Minus Infinity rounding mode, and writes the result to the general-purpose destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

`Rd = unsigned_convertToIntegerExactTowardNegative(Vn)`, where *R* is either `W` or `X`.

Related references

18.1 A64 floating-point instructions in alphabetical order on page 18-1143.

18.14 FCVTNS (scalar)

Floating-point Convert to Signed integer, rounding to nearest with ties to even (scalar).

Syntax

```
FCVTNS Wd, Hn ; Half-precision to 32-bit

FCVTNS Xd, Hn ; Half-precision to 64-bit

FCVTNS Wd, Sn ; Single-precision to 32-bit

FCVTNS Xd, Sn ; Single-precision to 64-bit

FCVTNS Wd, Dn ; Double-precision to 32-bit

FCVTNS Xd, Dn ; Double-precision to 64-bit
```

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

Hn
Is the 16-bit name of the SIMD and FP source register.

Xd
Is the 64-bit name of the general-purpose destination register.

Sn
Is the 32-bit name of the SIMD and FP source register.

Dn
Is the 64-bit name of the SIMD and FP source register.

Operation

Floating-point Convert to Signed integer, rounding to nearest with ties to even (scalar). This instruction converts the floating-point value in the SIMD and FP source register to a 32-bit or 64-bit signed integer using the Round to Nearest rounding mode, and writes the result to the general-purpose destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

`Rd = signed_convertToIntegerExactTiesToEven(Vn)`, where `R` is either `W` or `X`.

Related references

18.1 A64 floating-point instructions in alphabetical order on page 18-1143.

18.15 FCVTNU (scalar)

Floating-point Convert to Unsigned integer, rounding to nearest with ties to even (scalar).

Syntax

```
FCVTNU Wd, Hn ; Half-precision to 32-bit
FCVTNU Xd, Hn ; Half-precision to 64-bit
FCVTNU Wd, Sn ; Single-precision to 32-bit
FCVTNU Xd, Sn ; Single-precision to 64-bit
FCVTNU Wd, Dn ; Double-precision to 32-bit
FCVTNU Xd, Dn ; Double-precision to 64-bit
```

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

Hn
Is the 16-bit name of the SIMD and FP source register.

Xd
Is the 64-bit name of the general-purpose destination register.

Sn
Is the 32-bit name of the SIMD and FP source register.

Dn
Is the 64-bit name of the SIMD and FP source register.

Operation

Floating-point Convert to Unsigned integer, rounding to nearest with ties to even (scalar). This instruction converts the floating-point value in the SIMD and FP source register to a 32-bit or 64-bit unsigned integer using the Round to Nearest rounding mode, and writes the result to the general-purpose destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

`Rd = unsigned_convertToIntegerExactTiesToEven(Vn)`, where `R` is either `W` or `X`.

Related references

18.1 A64 floating-point instructions in alphabetical order on page 18-1143.

18.16 FCVTPS (scalar)

Floating-point Convert to Signed integer, rounding toward Plus infinity (scalar).

Syntax

```
FCVTPS Wd, Hn ; Half-precision to 32-bit
FCVTPS Xd, Hn ; Half-precision to 64-bit
FCVTPS Wd, Sn ; Single-precision to 32-bit
FCVTPS Xd, Sn ; Single-precision to 64-bit
FCVTPS Wd, Dn ; Double-precision to 32-bit
FCVTPS Xd, Dn ; Double-precision to 64-bit
```

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

Hn
Is the 16-bit name of the SIMD and FP source register.

Xd
Is the 64-bit name of the general-purpose destination register.

Sn
Is the 32-bit name of the SIMD and FP source register.

Dn
Is the 64-bit name of the SIMD and FP source register.

Operation

Floating-point Convert to Signed integer, rounding toward Plus infinity (scalar). This instruction converts the floating-point value in the SIMD and FP source register to a 32-bit or 64-bit signed integer using the Round towards Plus Infinity rounding mode, and writes the result to the general-purpose destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

`Rd = signed_convertToIntegerExactTowardPositive(Vn)`, where *R* is either `W` or `X`.

Related references

18.1 A64 floating-point instructions in alphabetical order on page 18-1143.

18.17 FCVTPU (scalar)

Floating-point Convert to Unsigned integer, rounding toward Plus infinity (scalar).

Syntax

```
FCVTPU Wd, Hn ; Half-precision to 32-bit
FCVTPU Xd, Hn ; Half-precision to 64-bit
FCVTPU Wd, Sn ; Single-precision to 32-bit
FCVTPU Xd, Sn ; Single-precision to 64-bit
FCVTPU Wd, Dn ; Double-precision to 32-bit
FCVTPU Xd, Dn ; Double-precision to 64-bit
```

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

Hn
Is the 16-bit name of the SIMD and FP source register.

Xd
Is the 64-bit name of the general-purpose destination register.

Sn
Is the 32-bit name of the SIMD and FP source register.

Dn
Is the 64-bit name of the SIMD and FP source register.

Operation

Floating-point Convert to Unsigned integer, rounding toward Plus infinity (scalar). This instruction converts the floating-point value in the SIMD and FP source register to a 32-bit or 64-bit unsigned integer using the Round towards Plus Infinity rounding mode, and writes the result to the general-purpose destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

`Rd = unsigned_convertToIntegerExactTowardPositive(Vn)`, where `R` is either `W` or `X`.

Related references

18.1 A64 floating-point instructions in alphabetical order on page 18-1143.

18.18 FCVTZS (scalar, fixed-point)

Floating-point Convert to Signed fixed-point, rounding toward Zero (scalar).

Syntax

```
FCVTZS Wd, Hn, #fbits ; Half-precision to 32-bit
FCVTZS Xd, Hn, #fbits ; Half-precision to 64-bit
FCVTZS Wd, Sn, #fbits ; Single-precision to 32-bit
FCVTZS Xd, Sn, #fbits ; Single-precision to 64-bit
FCVTZS Wd, Dn, #fbits ; Double-precision to 32-bit
FCVTZS Xd, Dn, #fbits ; Double-precision to 64-bit
```

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

Hn
Is the 16-bit name of the SIMD and FP source register.

fbits
Depends on the instruction variant:

32-bit
Is the number of bits after the binary point in the fixed-point destination, in the range 1 to 32.

64-bit
Is the number of bits after the binary point in the fixed-point destination, in the range 1 to 64.

Xd
Is the 64-bit name of the general-purpose destination register.

Sn
Is the 32-bit name of the SIMD and FP source register.

Dn
Is the 64-bit name of the SIMD and FP source register.

Operation

Floating-point Convert to Signed fixed-point, rounding toward Zero (scalar). This instruction converts the floating-point value in the SIMD and FP source register to a 32-bit or 64-bit fixed-point signed integer using the Round towards Zero rounding mode, and writes the result to the general-purpose destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

`Rd = signed_convertToIntegerExactTowardZero(Vn*(2^fbits))`, where `R` is either `W` or `X`.

Related references

18.1 A64 floating-point instructions in alphabetical order on page 18-1143.

18.19 FCVTZS (scalar, integer)

Floating-point Convert to Signed integer, rounding toward Zero (scalar).

Syntax

```
FCVTZS Wd, Hn ; Half-precision to 32-bit
FCVTZS Xd, Hn ; Half-precision to 64-bit
FCVTZS Wd, Sn ; Single-precision to 32-bit
FCVTZS Xd, Sn ; Single-precision to 64-bit
FCVTZS Wd, Dn ; Double-precision to 32-bit
FCVTZS Xd, Dn ; Double-precision to 64-bit
```

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

Hn
Is the 16-bit name of the SIMD and FP source register.

Xd
Is the 64-bit name of the general-purpose destination register.

Sn
Is the 32-bit name of the SIMD and FP source register.

Dn
Is the 64-bit name of the SIMD and FP source register.

Operation

Floating-point Convert to Signed integer, rounding toward Zero (scalar). This instruction converts the floating-point value in the SIMD and FP source register to a 32-bit or 64-bit signed integer using the Round towards Zero rounding mode, and writes the result to the general-purpose destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

`Rd = signed_convertToIntegerExactTowardZero(Vn)`, where `R` is either `W` or `X`.

Related references

18.1 A64 floating-point instructions in alphabetical order on page 18-1143.

18.20 FCVTZU (scalar, fixed-point)

Floating-point Convert to Unsigned fixed-point, rounding toward Zero (scalar).

Syntax

```
FCVTZU Wd, Hn, #fbits ; Half-precision to 32-bit
FCVTZU Xd, Hn, #fbits ; Half-precision to 64-bit
FCVTZU Wd, Sn, #fbits ; Single-precision to 32-bit
FCVTZU Xd, Sn, #fbits ; Single-precision to 64-bit
FCVTZU Wd, Dn, #fbits ; Double-precision to 32-bit
FCVTZU Xd, Dn, #fbits ; Double-precision to 64-bit
```

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

Hn
Is the 16-bit name of the SIMD and FP source register.

fbits
Depends on the instruction variant:

32-bit
Is the number of bits after the binary point in the fixed-point destination, in the range 1 to 32.

64-bit
Is the number of bits after the binary point in the fixed-point destination, in the range 1 to 64.

Xd
Is the 64-bit name of the general-purpose destination register.

Sn
Is the 32-bit name of the SIMD and FP source register.

Dn
Is the 64-bit name of the SIMD and FP source register.

Operation

Floating-point Convert to Unsigned fixed-point, rounding toward Zero (scalar). This instruction converts the floating-point value in the SIMD and FP source register to a 32-bit or 64-bit fixed-point unsigned integer using the Round towards Zero rounding mode, and writes the result to the general-purpose destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

`Rd = unsigned_convertToIntegerExactTowardZero(Vn*(2^fbits))`, where `R` is either `W` or `X`.

Related references

18.1 A64 floating-point instructions in alphabetical order on page 18-1143.

18.21 FCVTZU (scalar, integer)

Floating-point Convert to Unsigned integer, rounding toward Zero (scalar).

Syntax

```
FCVTZU Wd, Hn ; Half-precision to 32-bit
FCVTZU Xd, Hn ; Half-precision to 64-bit
FCVTZU Wd, Sn ; Single-precision to 32-bit
FCVTZU Xd, Sn ; Single-precision to 64-bit
FCVTZU Wd, Dn ; Double-precision to 32-bit
FCVTZU Xd, Dn ; Double-precision to 64-bit
```

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

Hn
Is the 16-bit name of the SIMD and FP source register.

Xd
Is the 64-bit name of the general-purpose destination register.

Sn
Is the 32-bit name of the SIMD and FP source register.

Dn
Is the 64-bit name of the SIMD and FP source register.

Operation

Floating-point Convert to Unsigned integer, rounding toward Zero (scalar). This instruction converts the floating-point value in the SIMD and FP source register to a 32-bit or 64-bit unsigned integer using the Round towards Zero rounding mode, and writes the result to the general-purpose destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

`Rd = unsigned_convertToIntegerExactTowardZero(Vn)`, where `R` is either `W` or `X`.

Related references

18.1 A64 floating-point instructions in alphabetical order on page 18-1143.

18.22 FDIV (scalar)

Floating-point Divide (scalar).

Syntax

```
FDIV Hd, Hn, Hm ; Half-precision

FDIV Sd, Sn, Sm ; Single-precision

FDIV Dd, Dn, Dm ; Double-precision
```

Where:

Hd
Is the 16-bit name of the SIMD and FP destination register.

Hn
Is the 16-bit name of the first SIMD and FP source register.

Hm
Is the 16-bit name of the second SIMD and FP source register.

Sd
Is the 32-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.

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.

Operation

Floating-point Divide (scalar). This instruction divides the floating-point value of the first source SIMD and FP register by the floating-point value of the second source SIMD and FP register, and writes the result to the destination SIMD and FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
Vd = Vn / Vm.
```

Related references

18.1 A64 floating-point instructions in alphabetical order on page 18-1143.

18.23 FJCVTZS

Floating-point Javascript Convert to Signed fixed-point, rounding toward Zero.

Syntax

```
FJCVTZS Wd, Dn
```

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

Dn
Is the 64-bit name of the SIMD and FP source register.

Architectures supported

Supported in the Armv8.3-A architecture and later.

Usage

Floating-point 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.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

18.1 A64 floating-point instructions in alphabetical order on page 18-1143.

18.24 FMADD

Floating-point fused Multiply-Add (scalar).

Syntax

```
FMADD Hd, Hn, Hm, Ha ; Half-precision

FMADD Sd, Sn, Sm, Sa ; Single-precision

FMADD Dd, Dn, Dm, Da ; Double-precision
```

Where:

Hd
Is the 16-bit name of the SIMD and FP destination register.

Hn
Is the 16-bit name of the first SIMD and FP source register holding the multiplicand.

Hm
Is the 16-bit name of the second SIMD and FP source register holding the multiplier.

Ha
Is the 16-bit name of the third SIMD and FP source register holding the addend.

Sd
Is the 32-bit name of the SIMD and FP destination register.

Sn
Is the 32-bit name of the first SIMD and FP source register holding the multiplicand.

Sm
Is the 32-bit name of the second SIMD and FP source register holding the multiplier.

Sa
Is the 32-bit name of the third SIMD and FP source register holding the addend.

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 holding the multiplicand.

Dm
Is the 64-bit name of the second SIMD and FP source register holding the multiplier.

Da
Is the 64-bit name of the third SIMD and FP source register holding the addend.

Operation

Floating-point fused Multiply-Add (scalar). This instruction multiplies the values of the first two SIMD and FP source registers, adds the product to the value of the third SIMD and FP source register, and writes the result to the SIMD and FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
Vd = Va + Vn*Vm.
```

Related references

18.1 A64 floating-point instructions in alphabetical order on page 18-1143.

18.25 FMAX (scalar)

Floating-point Maximum (scalar).

Syntax

```
FMAX Hd, Hn, Hm ; Half-precision

FMAX Sd, Sn, Sm ; Single-precision

FMAX Dd, Dn, Dm ; Double-precision
```

Where:

Hd
Is the 16-bit name of the SIMD and FP destination register.

Hn
Is the 16-bit name of the first SIMD and FP source register.

Hm
Is the 16-bit name of the second SIMD and FP source register.

Sd
Is the 32-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.

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.

Operation

Floating-point Maximum (scalar). This instruction compares the two source SIMD and FP registers, and writes the larger of the two floating-point values to the destination SIMD and FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
Vd = max(Vn, Vm).
```

Related references

18.1 A64 floating-point instructions in alphabetical order on page 18-1143.

18.26 FMAXNM (scalar)

Floating-point Maximum Number (scalar).

Syntax

```
FMAXNM Hd, Hn, Hm ; Half-precision

FMAXNM Sd, Sn, Sm ; Single-precision

FMAXNM Dd, Dn, Dm ; Double-precision
```

Where:

Hd
Is the 16-bit name of the SIMD and FP destination register.

Hn
Is the 16-bit name of the first SIMD and FP source register.

Hm
Is the 16-bit name of the second SIMD and FP source register.

Sd
Is the 32-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.

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.

Operation

Floating-point Maximum Number (scalar). This instruction compares the first and second source SIMD and FP register values, and writes the larger of the two floating-point values to the destination SIMD and FP register.

NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result that is placed in the vector is the numerical value, otherwise the result is identical to *FMAX (scalar)*.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
Vd = maxNum(Vn, Vm).
```

Related references

18.1 A64 floating-point instructions in alphabetical order on page 18-1143.

18.27 FMIN (scalar)

Floating-point Minimum (scalar).

Syntax

```
FMIN Hd, Hn, Hm ; Half-precision

FMIN Sd, Sn, Sm ; Single-precision

FMIN Dd, Dn, Dm ; Double-precision
```

Where:

Hd
Is the 16-bit name of the SIMD and FP destination register.

Hn
Is the 16-bit name of the first SIMD and FP source register.

Hm
Is the 16-bit name of the second SIMD and FP source register.

Sd
Is the 32-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.

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.

Operation

Floating-point Minimum (scalar). This instruction compares the first and second source SIMD and FP register values, and writes the smaller of the two floating-point values to the destination SIMD and FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
Vd = min(Vn, Vm).
```

Related references

18.1 A64 floating-point instructions in alphabetical order on page 18-1143.

18.28 FMINNM (scalar)

Floating-point Minimum Number (scalar).

Syntax

```
FMINNM Hd, Hn, Hm ; Half-precision

FMINNM Sd, Sn, Sm ; Single-precision

FMINNM Dd, Dn, Dm ; Double-precision
```

Where:

Hd
Is the 16-bit name of the SIMD and FP destination register.

Hn
Is the 16-bit name of the first SIMD and FP source register.

Hm
Is the 16-bit name of the second SIMD and FP source register.

Sd
Is the 32-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.

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.

Operation

Floating-point Minimum Number (scalar). This instruction compares the first and second source SIMD and FP register values, and writes the smaller of the two floating-point values to the destination SIMD and FP register.

NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result that is placed in the vector is the numerical value, otherwise the result is identical to *FMIN (scalar)*.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
Vd = minNum(Vn, Vm).
```

Related references

18.1 A64 floating-point instructions in alphabetical order on page 18-1143.

18.29 FMOV (register)

Floating-point Move register without conversion.

Syntax

```
FMOV Hd, Hn ; Half-precision

FMOV Sd, Sn ; Single-precision

FMOV Dd, Dn ; Double-precision
```

Where:

Hd
Is the 16-bit name of the SIMD and FP destination register.

Hn
Is the 16-bit name of the SIMD and FP source register.

Sd
Is the 32-bit name of the SIMD and FP destination register.

Sn
Is the 32-bit name of the SIMD and FP source register.

Dd
Is the 64-bit name of the SIMD and FP destination register.

Dn
Is the 64-bit name of the SIMD and FP source register.

Operation

Floating-point Move register without conversion. This instruction copies the floating-point value in the SIMD and FP source register to the SIMD and FP destination register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
Vd = Vn.
```

Related references

18.1 A64 floating-point instructions in alphabetical order on page 18-1143.

18.30 FMOV (general)

Floating-point Move to or from general-purpose register without conversion.

Syntax

```
FMOV Wd, Hn ; Half-precision to 32-bit

FMOV Xd, Hn ; Half-precision to 64-bit

FMOV Hd, Wn ; 32-bit to half-precision

FMOV Sd, Wn ; 32-bit to single-precision

FMOV Wd, Sn ; Single-precision to 32-bit

FMOV Hd, Xn ; 64-bit to half-precision

FMOV Dd, Xn ; 64-bit to double-precision

FMOV Vd.D[1], Xn ; 64-bit to top half of 128-bit

FMOV Xd, Dn ; Double-precision to 64-bit

FMOV Xd, Vn.D[1] ; Top half of 128-bit to 64-bit
```

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

Hn
Is the 16-bit name of the SIMD and FP source register.

Xd
Is the 64-bit name of the general-purpose destination register.

Hd
Is the 16-bit name of the SIMD and FP destination register.

Wn
Is the 32-bit name of the general-purpose source register.

Sd
Is the 32-bit name of the SIMD and FP destination register.

Sn
Is the 32-bit name of the SIMD and FP source register.

Xn
Is the 64-bit name of the general-purpose source register.

Dd
Is the 64-bit name of the SIMD and FP destination register.

Vd
Is the name of the SIMD and FP destination register.

Dn
Is the 64-bit name of the SIMD and FP source register.

Vn
Is the name of the SIMD and FP source register.

Usage

Floating-point Move to or from general-purpose register without conversion. This instruction transfers the contents of a SIMD and FP register to a general-purpose register, or the contents of a general-purpose register to a SIMD and FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

18.1 A64 floating-point instructions in alphabetical order on page 18-1143.

18.31 FMOV (scalar, immediate)

Floating-point move immediate (scalar).

Syntax

```
FMOV Hd, #imm ; Half-precision

FMOV Sd, #imm ; Single-precision

FMOV Dd, #imm ; Double-precision
```

Where:

Hd
Is the 16-bit name of the SIMD and FP destination register.

Sd
Is the 32-bit name of the SIMD and FP destination register.

Dd
Is the 64-bit name of the SIMD and FP destination register.

imm
Is a signed floating-point constant with 3-bit exponent and normalized 4 bits of precision. For details of the range of constants available and the encoding of `imm`, see *Modified immediate constants in A64 floating-point instructions* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Usage

Floating-point move immediate (scalar). This instruction copies a floating-point immediate constant into the SIMD and FP destination register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

`Vd=#imm`.

Related references

18.1 A64 floating-point instructions in alphabetical order on page 18-1143.

18.32 FMSUB

Floating-point Fused Multiply-Subtract (scalar).

Syntax

```
FMSUB Hd, Hn, Hm, Ha ; Half-precision

FMSUB Sd, Sn, Sm, Sa ; Single-precision

FMSUB Dd, Dn, Dm, Da ; Double-precision
```

Where:

Hd
Is the 16-bit name of the SIMD and FP destination register.

Hn
Is the 16-bit name of the first SIMD and FP source register holding the multiplicand.

Hm
Is the 16-bit name of the second SIMD and FP source register holding the multiplier.

Ha
Is the 16-bit name of the third SIMD and FP source register holding the minuend.

Sd
Is the 32-bit name of the SIMD and FP destination register.

Sn
Is the 32-bit name of the first SIMD and FP source register holding the multiplicand.

Sm
Is the 32-bit name of the second SIMD and FP source register holding the multiplier.

Sa
Is the 32-bit name of the third SIMD and FP source register holding the minuend.

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 holding the multiplicand.

Dm
Is the 64-bit name of the second SIMD and FP source register holding the multiplier.

Da
Is the 64-bit name of the third SIMD and FP source register holding the minuend.

Operation

Floating-point Fused Multiply-Subtract (scalar). This instruction multiplies the values of the first two SIMD and FP source registers, negates the product, adds that to the value of the third SIMD and FP source register, and writes the result to the SIMD and FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
Vd = Va + (-Vn)*Vm.
```

Related references

18.1 A64 floating-point instructions in alphabetical order on page 18-1143.

18.33 FMUL (scalar)

Floating-point Multiply (scalar).

Syntax

```
FMUL Hd, Hn, Hm ; Half-precision

FMUL Sd, Sn, Sm ; Single-precision

FMUL Dd, Dn, Dm ; Double-precision
```

Where:

Hd
Is the 16-bit name of the SIMD and FP destination register.

Hn
Is the 16-bit name of the first SIMD and FP source register.

Hm
Is the 16-bit name of the second SIMD and FP source register.

Sd
Is the 32-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.

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.

Operation

Floating-point Multiply (scalar). This instruction multiplies the floating-point values of the two source SIMD and FP registers, and writes the result to the destination SIMD and FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
Vd = Vn * Vm.
```

Related references

18.1 A64 floating-point instructions in alphabetical order on page 18-1143.

18.34 FNEG (scalar)

Floating-point Negate (scalar).

Syntax

```
FNEG Hd, Hn ; Half-precision

FNEG Sd, Sn ; Single-precision

FNEG Dd, Dn ; Double-precision
```

Where:

Hd
Is the 16-bit name of the SIMD and FP destination register.

Hn
Is the 16-bit name of the SIMD and FP source register.

Sd
Is the 32-bit name of the SIMD and FP destination register.

Sn
Is the 32-bit name of the SIMD and FP source register.

Dd
Is the 64-bit name of the SIMD and FP destination register.

Dn
Is the 64-bit name of the SIMD and FP source register.

Operation

Floating-point Negate (scalar). This instruction negates the value in the SIMD and FP source register and writes the result to the SIMD and FP destination register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
Vd = -Vn.
```

Related references

18.1 A64 floating-point instructions in alphabetical order on page 18-1143.

18.35 FNMADD

Floating-point Negated fused Multiply-Add (scalar).

Syntax

```
FNMADD Hd, Hn, Hm, Ha ; Half-precision

FNMADD Sd, Sn, Sm, Sa ; Single-precision

FNMADD Dd, Dn, Dm, Da ; Double-precision
```

Where:

Hd
Is the 16-bit name of the SIMD and FP destination register.

Hn
Is the 16-bit name of the first SIMD and FP source register holding the multiplicand.

Hm
Is the 16-bit name of the second SIMD and FP source register holding the multiplier.

Ha
Is the 16-bit name of the third SIMD and FP source register holding the addend.

Sd
Is the 32-bit name of the SIMD and FP destination register.

Sn
Is the 32-bit name of the first SIMD and FP source register holding the multiplicand.

Sm
Is the 32-bit name of the second SIMD and FP source register holding the multiplier.

Sa
Is the 32-bit name of the third SIMD and FP source register holding the addend.

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 holding the multiplicand.

Dm
Is the 64-bit name of the second SIMD and FP source register holding the multiplier.

Da
Is the 64-bit name of the third SIMD and FP source register holding the addend.

Operation

Floating-point Negated fused Multiply-Add (scalar). This instruction multiplies the values of the first two SIMD and FP source registers, negates the product, subtracts the value of the third SIMD and FP source register, and writes the result to the destination SIMD and FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
Vd = (-Va) + (-Vn)*Vm.
```

Related references

18.1 A64 floating-point instructions in alphabetical order on page 18-1143.

18.36 FNMSUB

Floating-point Negated fused Multiply-Subtract (scalar).

Syntax

```
FNMSUB Hd, Hn, Hm, Ha ; Half-precision

FNMSUB Sd, Sn, Sm, Sa ; Single-precision

FNMSUB Dd, Dn, Dm, Da ; Double-precision
```

Where:

Hd
Is the 16-bit name of the SIMD and FP destination register.

Hn
Is the 16-bit name of the first SIMD and FP source register holding the multiplicand.

Hm
Is the 16-bit name of the second SIMD and FP source register holding the multiplier.

Ha
Is the 16-bit name of the third SIMD and FP source register holding the minuend.

Sd
Is the 32-bit name of the SIMD and FP destination register.

Sn
Is the 32-bit name of the first SIMD and FP source register holding the multiplicand.

Sm
Is the 32-bit name of the second SIMD and FP source register holding the multiplier.

Sa
Is the 32-bit name of the third SIMD and FP source register holding the minuend.

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 holding the multiplicand.

Dm
Is the 64-bit name of the second SIMD and FP source register holding the multiplier.

Da
Is the 64-bit name of the third SIMD and FP source register holding the minuend.

Operation

Floating-point Negated fused Multiply-Subtract (scalar). This instruction multiplies the values of the first two SIMD and FP source registers, subtracts the value of the third SIMD and FP source register, and writes the result to the destination SIMD and FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
Vd = (-Va) + Vn*Vm.
```

Related references

18.1 A64 floating-point instructions in alphabetical order on page 18-1143.

18.37 FNMUL (scalar)

Floating-point Multiply-Negate (scalar).

Syntax

```
FNMUL Hd, Hn, Hm ; Half-precision

FNMUL Sd, Sn, Sm ; Single-precision

FNMUL Dd, Dn, Dm ; Double-precision
```

Where:

Hd
Is the 16-bit name of the SIMD and FP destination register.

Hn
Is the 16-bit name of the first SIMD and FP source register.

Hm
Is the 16-bit name of the second SIMD and FP source register.

Sd
Is the 32-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.

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.

Operation

Floating-point Multiply-Negate (scalar). This instruction multiplies the floating-point values of the two source SIMD and FP registers, and writes the negation of the result to the destination SIMD and FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
Vd = -(Vn * Vm).
```

Related references

18.1 A64 floating-point instructions in alphabetical order on page 18-1143.

18.38 FRINTA (scalar)

Floating-point Round to Integral, to nearest with ties to Away (scalar).

Syntax

```
FRINTA Hd, Hn ; Half-precision

FRINTA Sd, Sn ; Single-precision

FRINTA Dd, Dn ; Double-precision
```

Where:

Hd
Is the 16-bit name of the SIMD and FP destination register.

Hn
Is the 16-bit name of the SIMD and FP source register.

Sd
Is the 32-bit name of the SIMD and FP destination register.

Sn
Is the 32-bit name of the SIMD and FP source register.

Dd
Is the 64-bit name of the SIMD and FP destination register.

Dn
Is the 64-bit name of the SIMD and FP source register.

Operation

Floating-point Round to Integral, to nearest with ties to Away (scalar). This instruction rounds a floating-point value in the SIMD and FP source register to an integral floating-point value of the same size using the Round to Nearest with Ties to Away rounding mode, and writes the result to the SIMD and FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
Vd = roundToIntegralTiesToAway(Vn).
```

Related references

18.1 A64 floating-point instructions in alphabetical order on page 18-1143.

18.39 FRINTI (scalar)

Floating-point Round to Integral, using current rounding mode (scalar).

Syntax

```
FRINTI Hd, Hn ; Half-precision

FRINTI Sd, Sn ; Single-precision

FRINTI Dd, Dn ; Double-precision
```

Where:

Hd
Is the 16-bit name of the SIMD and FP destination register.

Hn
Is the 16-bit name of the SIMD and FP source register.

Sd
Is the 32-bit name of the SIMD and FP destination register.

Sn
Is the 32-bit name of the SIMD and FP source register.

Dd
Is the 64-bit name of the SIMD and FP destination register.

Dn
Is the 64-bit name of the SIMD and FP source register.

Operation

Floating-point Round to Integral, using current rounding mode (scalar). This instruction rounds a floating-point value in the SIMD and FP source register to an integral floating-point value of the same size using the rounding mode that is determined by the FPCR, and writes the result to the SIMD and FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
Vd = roundToIntegral(Vn).
```

Related references

18.1 A64 floating-point instructions in alphabetical order on page 18-1143.

18.40 FRINTM (scalar)

Floating-point Round to Integral, toward Minus infinity (scalar).

Syntax

```
FRINTM Hd, Hn ; Half-precision

FRINTM Sd, Sn ; Single-precision

FRINTM Dd, Dn ; Double-precision
```

Where:

Hd
Is the 16-bit name of the SIMD and FP destination register.

Hn
Is the 16-bit name of the SIMD and FP source register.

Sd
Is the 32-bit name of the SIMD and FP destination register.

Sn
Is the 32-bit name of the SIMD and FP source register.

Dd
Is the 64-bit name of the SIMD and FP destination register.

Dn
Is the 64-bit name of the SIMD and FP source register.

Operation

Floating-point Round to Integral, toward Minus infinity (scalar). This instruction rounds a floating-point value in the SIMD and FP source register to an integral floating-point value of the same size using the Round towards Minus Infinity rounding mode, and writes the result to the SIMD and FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
Vd = roundToIntegralTowardNegative(Vn).
```

Related references

18.1 A64 floating-point instructions in alphabetical order on page 18-1143.

18.41 FRINTN (scalar)

Floating-point Round to Integral, to nearest with ties to even (scalar).

Syntax

```
FRINTN Hd, Hn ; Half-precision

FRINTN Sd, Sn ; Single-precision

FRINTN Dd, Dn ; Double-precision
```

Where:

Hd
Is the 16-bit name of the SIMD and FP destination register.

Hn
Is the 16-bit name of the SIMD and FP source register.

Sd
Is the 32-bit name of the SIMD and FP destination register.

Sn
Is the 32-bit name of the SIMD and FP source register.

Dd
Is the 64-bit name of the SIMD and FP destination register.

Dn
Is the 64-bit name of the SIMD and FP source register.

Operation

Floating-point Round to Integral, to nearest with ties to even (scalar). This instruction rounds a floating-point value in the SIMD and FP source register to an integral floating-point value of the same size using the Round to Nearest rounding mode, and writes the result to the SIMD and FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
Vd = roundToIntegralTiesToEven(Vn).
```

Related references

18.1 A64 floating-point instructions in alphabetical order on page 18-1143.

18.42 FRINTP (scalar)

Floating-point Round to Integral, toward Plus infinity (scalar).

Syntax

```
FRINTP Hd, Hn ; Half-precision

FRINTP Sd, Sn ; Single-precision

FRINTP Dd, Dn ; Double-precision
```

Where:

Hd
Is the 16-bit name of the SIMD and FP destination register.

Hn
Is the 16-bit name of the SIMD and FP source register.

Sd
Is the 32-bit name of the SIMD and FP destination register.

Sn
Is the 32-bit name of the SIMD and FP source register.

Dd
Is the 64-bit name of the SIMD and FP destination register.

Dn
Is the 64-bit name of the SIMD and FP source register.

Operation

Floating-point Round to Integral, toward Plus infinity (scalar). This instruction rounds a floating-point value in the SIMD and FP source register to an integral floating-point value of the same size using the Round towards Plus Infinity rounding mode, and writes the result to the SIMD and FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
Vd = roundToIntegralTowardPositive(Vn).
```

Related references

18.1 A64 floating-point instructions in alphabetical order on page 18-1143.

18.43 FRINTX (scalar)

Floating-point Round to Integral exact, using current rounding mode (scalar).

Syntax

```
FRINTX Hd, Hn ; Half-precision

FRINTX Sd, Sn ; Single-precision

FRINTX Dd, Dn ; Double-precision
```

Where:

Hd
Is the 16-bit name of the SIMD and FP destination register.

Hn
Is the 16-bit name of the SIMD and FP source register.

Sd
Is the 32-bit name of the SIMD and FP destination register.

Sn
Is the 32-bit name of the SIMD and FP source register.

Dd
Is the 64-bit name of the SIMD and FP destination register.

Dn
Is the 64-bit name of the SIMD and FP source register.

Operation

Floating-point Round to Integral exact, using current rounding mode (scalar). This instruction rounds a floating-point value in the SIMD and FP source register to an integral floating-point value of the same size using the rounding mode that is determined by the FPCR, and writes the result to the SIMD and FP destination register.

An Inexact exception is raised when the result value is not numerically equal to the input value. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
Vd = roundToIntegralExact(Vn).
```

Related references

18.1 A64 floating-point instructions in alphabetical order on page 18-1143.

18.44 FRINTZ (scalar)

Floating-point Round to Integral, toward Zero (scalar).

Syntax

```
FRINTZ Hd, Hn ; Half-precision

FRINTZ Sd, Sn ; Single-precision

FRINTZ Dd, Dn ; Double-precision
```

Where:

Hd
Is the 16-bit name of the SIMD and FP destination register.

Hn
Is the 16-bit name of the SIMD and FP source register.

Sd
Is the 32-bit name of the SIMD and FP destination register.

Sn
Is the 32-bit name of the SIMD and FP source register.

Dd
Is the 64-bit name of the SIMD and FP destination register.

Dn
Is the 64-bit name of the SIMD and FP source register.

Operation

Floating-point Round to Integral, toward Zero (scalar). This instruction rounds a floating-point value in the SIMD and FP source register to an integral floating-point value of the same size using the Round towards Zero rounding mode, and writes the result to the SIMD and FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
Vd = roundToIntegralTowardZero(Vn).
```

Related references

18.1 A64 floating-point instructions in alphabetical order on page 18-1143.

18.45 FSQRT (scalar)

Floating-point Square Root (scalar).

Syntax

```
FSQRT Hd, Hn ; Half-precision

FSQRT Sd, Sn ; Single-precision

FSQRT Dd, Dn ; Double-precision
```

Where:

Hd
Is the 16-bit name of the SIMD and FP destination register.

Hn
Is the 16-bit name of the SIMD and FP source register.

Sd
Is the 32-bit name of the SIMD and FP destination register.

Sn
Is the 32-bit name of the SIMD and FP source register.

Dd
Is the 64-bit name of the SIMD and FP destination register.

Dn
Is the 64-bit name of the SIMD and FP source register.

Operation

Floating-point Square Root (scalar). This instruction calculates the square root of the value in the SIMD and FP source register and writes the result to the SIMD and FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
Vd = sqrt(Vn).
```

Related references

18.1 A64 floating-point instructions in alphabetical order on page 18-1143.

18.46 FSUB (scalar)

Floating-point Subtract (scalar).

Syntax

```
FSUB Hd, Hn, Hm ; Half-precision

FSUB Sd, Sn, Sm ; Single-precision

FSUB Dd, Dn, Dm ; Double-precision
```

Where:

Hd
Is the 16-bit name of the SIMD and FP destination register.

Hn
Is the 16-bit name of the first SIMD and FP source register.

Hm
Is the 16-bit name of the second SIMD and FP source register.

Sd
Is the 32-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.

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.

Operation

Floating-point Subtract (scalar). This instruction subtracts the floating-point value of the second source SIMD and FP register from the floating-point value of the first source SIMD and FP register, and writes the result to the destination SIMD and FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
Vd = Vn - Vm.
```

Related references

18.1 A64 floating-point instructions in alphabetical order on page 18-1143.

18.47 LDNP (SIMD and FP)

Load Pair of SIMD and FP registers, with Non-temporal hint.

Syntax

```
LDNP St1, St2, [Xn|SP{, #imm}] ; 32-bit FP/SIMD registers, Signed offset

LDNP Dt1, Dt2, [Xn|SP{, #imm}] ; 64-bit FP/SIMD registers, Signed offset

LDNP Qt1, Qt2, [Xn|SP{, #imm}] ; 128-bit FP/SIMD registers, Signed offset
```

Where:

St1
Is the 32-bit name of the first SIMD and FP register to be transferred.

St2
Is the 32-bit name of the second SIMD and FP register to be transferred.

imm
Depends on the instruction variant:

32-bit FP/SIMD registers
Is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0.

64-bit FP/SIMD registers
Is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0.

128-bit FP/SIMD registers
Is the optional signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, defaulting to 0.

Dt1
Is the 64-bit name of the first SIMD and FP register to be transferred.

Dt2
Is the 64-bit name of the second SIMD and FP register to be transferred.

Qt1
Is the 128-bit name of the first SIMD and FP register to be transferred.

Qt2
Is the 128-bit name of the second SIMD and FP register to be transferred.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Usage

Load Pair of SIMD and FP registers, with Non-temporal hint. This instruction loads a pair of SIMD and FP registers from memory, issuing a hint to the memory system that the access is non-temporal. The address that is used for the load is calculated from a base register value and an optional immediate offset.

For information about non-temporal pair instructions, see *Load/Store SIMD and Floating-point Non-temporal pair* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Note

For 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*, and particularly LDNP (SIMD and FP).

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

18.48 LDP (SIMD and FP)

Load Pair of SIMD and FP registers.

Syntax

```
LDP St1, St2, [Xn|SP], #imm ; 32-bit FP/SIMD registers, Post-index

LDP Dt1, Dt2, [Xn|SP], #imm ; 64-bit FP/SIMD registers, Post-index

LDP Qt1, Qt2, [Xn|SP], #imm ; 128-bit FP/SIMD registers, Post-index

LDP St1, St2, [Xn|SP, #imm]! ; 32-bit FP/SIMD registers, Pre-index

LDP Dt1, Dt2, [Xn|SP, #imm]! ; 64-bit FP/SIMD registers, Pre-index

LDP Qt1, Qt2, [Xn|SP, #imm]! ; 128-bit FP/SIMD registers, Pre-index

LDP St1, St2, [Xn|SP{, #imm}] ; 32-bit FP/SIMD registers, Signed offset

LDP Dt1, Dt2, [Xn|SP{, #imm}] ; 64-bit FP/SIMD registers, Signed offset

LDP Qt1, Qt2, [Xn|SP{, #imm}] ; 128-bit FP/SIMD registers, Signed offset
```

Where:

St1
Is the 32-bit name of the first SIMD and FP register to be transferred.

St2
Is the 32-bit name of the second SIMD and FP register to be transferred.

imm
Depends on the instruction variant:

32-bit FP/SIMD registers
Is the signed immediate byte offset, a multiple of 4 in the range -256 to 252.

64-bit FP/SIMD registers
Is the signed immediate byte offset, a multiple of 8 in the range -512 to 504.

128-bit FP/SIMD registers
Is the signed immediate byte offset, a multiple of 16 in the range -1024 to 1008.

Dt1
Is the 64-bit name of the first SIMD and FP register to be transferred.

Dt2
Is the 64-bit name of the second SIMD and FP register to be transferred.

Qt1
Is the 128-bit name of the first SIMD and FP register to be transferred.

Qt2
Is the 128-bit name of the second SIMD and FP register to be transferred.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Usage

Load Pair of SIMD and FP registers. This instruction loads a pair of SIMD and FP registers from memory. The address that is used for the load is calculated from a base register value and an optional immediate offset.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Note

For 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*, and particularly LDP (SIMD and FP).

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

18.49 LDR (immediate, SIMD and FP)

Load SIMD and FP Register (immediate offset).

Syntax

```
LDR <Bt>, [Xn|SP], #simm ; 8-bit FP/SIMD registers, Post-index
LDR Ht, [Xn|SP], #simm ; 16-bit FP/SIMD registers, Post-index
LDR St, [Xn|SP], #simm ; 32-bit FP/SIMD registers, Post-index
LDR Dt, [Xn|SP], #simm ; 64-bit FP/SIMD registers, Post-index
LDR Qt, [Xn|SP], #simm ; 128-bit FP/SIMD registers, Post-index
LDR <Bt>, [Xn|SP, #simm]! ; 8-bit FP/SIMD registers, Pre-index
LDR Ht, [Xn|SP, #simm]! ; 16-bit FP/SIMD registers, Pre-index
LDR St, [Xn|SP, #simm]! ; 32-bit FP/SIMD registers, Pre-index
LDR Dt, [Xn|SP, #simm]! ; 64-bit FP/SIMD registers, Pre-index
LDR Qt, [Xn|SP, #simm]! ; 128-bit FP/SIMD registers, Pre-index
LDR <Bt>, [Xn|SP{, #pimm}] ; 8-bit FP/SIMD registers
LDR Ht, [Xn|SP{, #pimm}] ; 16-bit FP/SIMD registers
LDR St, [Xn|SP{, #pimm}] ; 32-bit FP/SIMD registers
LDR Dt, [Xn|SP{, #pimm}] ; 64-bit FP/SIMD registers
LDR Qt, [Xn|SP{, #pimm}] ; 128-bit FP/SIMD registers
```

Where:

<Bt>
Is the 8-bit name of the SIMD and FP register to be transferred.

simm
Is the signed immediate byte offset, in the range -256 to 255.

Ht
Is the 16-bit name of the SIMD and FP register to be transferred.

St
Is the 32-bit name of the SIMD and FP register to be transferred.

Dt
Is the 64-bit name of the SIMD and FP register to be transferred.

Qt
Is the 128-bit name of the SIMD and FP register to be transferred.

pimm
Depends on the instruction variant:

8-bit FP/SIMD registers
Is the optional positive immediate byte offset, in the range 0 to 4095, defaulting to 0.

16-bit FP/SIMD registers
Is the optional positive immediate byte offset, a multiple of 2 in the range 0 to 8190, defaulting to 0.

32-bit FP/SIMD registers
Is the optional positive immediate byte offset, a multiple of 4 in the range 0 to 16380, defaulting to 0.

64-bit FP/SIMD registers
Is the optional positive immediate byte offset, a multiple of 8 in the range 0 to 32760, defaulting to 0.

128-bit FP/SIMD registers

Is the optional positive immediate byte offset, a multiple of 16 in the range 0 to 65520, defaulting to 0.

Xn|SP

Is the 64-bit name of the general-purpose base register or stack pointer.

Usage

Load SIMD and FP Register (immediate offset). This instruction loads an element from memory, and writes the result as a scalar to the SIMD and FP register. The address that is used for the load is calculated from a base register value, a signed immediate offset, and an optional offset that is a multiple of the element size.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

18.50 LDR (literal, SIMD and FP)

Load SIMD and FP Register (PC-relative literal).

Syntax

```
LDR St, label ; 32-bit FP/SIMD registers

LDR Dt, label ; 64-bit FP/SIMD registers

LDR Qt, label ; 128-bit FP/SIMD registers
```

Where:

St
Is the 32-bit name of the SIMD and FP register to be loaded.

Dt
Is the 64-bit name of the SIMD and FP register to be loaded.

Qt
Is the 128-bit name of the SIMD and FP register to be loaded.

label
Is the program label from which the data is to be loaded. Its offset from the address of this instruction, in the range ±1MB.

Usage

Load SIMD and FP Register (PC-relative literal). This instruction loads a SIMD and FP register from memory. The address that is used for the load is calculated from the PC value and an immediate offset.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

18.51 LDR (register, SIMD and FP)

Load SIMD and FP Register (register offset).

Syntax

```
LDR <Bt>, [Xn|SP, (Wm|Xm), extend {amount}] ; 8-bit FP/SIMD registers

LDR <Bt>, [Xn|SP, Xm{, LSL amount}] ; 8-bit FP/SIMD registers

LDR Ht, [Xn|SP, (Wm|Xm){, extend {amount}}] ; 16-bit FP/SIMD registers

LDR St, [Xn|SP, (Wm|Xm){, extend {amount}}] ; 32-bit FP/SIMD registers

LDR Dt, [Xn|SP, (Wm|Xm){, extend {amount}}] ; 64-bit FP/SIMD registers

LDR Qt, [Xn|SP, (Wm|Xm){, extend {amount}}] ; 128-bit FP/SIMD registers
```

Where:

<Bt>
Is the 8-bit name of the SIMD and FP register to be transferred.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Wm
When "option<0>" is set to 0, is the 32-bit name of the general-purpose index register.

Xm
When "option<0>" is set to 1, is the 64-bit name of the general-purpose index register.

extend
Is the index extend specifier:

8-bit FP/SIMD registers
Can be one of `UXTW`, `SXTW` or `SXTX`.

16-bit FP/SIMD registers
Can be one of `UXTW`, `LSL`, `SXTW` or `SXTX`.

amount
Is the index shift amount, it must be.

Ht
Is the 16-bit name of the SIMD and FP register to be transferred.

St
Is the 32-bit name of the SIMD and FP register to be transferred.

Dt
Is the 64-bit name of the SIMD and FP register to be transferred.

Qt
Is the 128-bit name of the SIMD and FP register to be transferred.

Usage

Load SIMD and FP Register (register offset). This instruction loads a SIMD and FP register from memory. The address that is used for the load is calculated from a base register value and an offset register value. The offset can be optionally shifted and extended.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

18.52 LDUR (SIMD and FP)

Load SIMD and FP Register (unscaled offset).

Syntax

```
LDUR <Bt>, [Xn|SP{, #simm}] ; 8-bit FP/SIMD registers

LDUR Ht, [Xn|SP{, #simm}] ; 16-bit FP/SIMD registers

LDUR St, [Xn|SP{, #simm}] ; 32-bit FP/SIMD registers

LDUR Dt, [Xn|SP{, #simm}] ; 64-bit FP/SIMD registers

LDUR Qt, [Xn|SP{, #simm}] ; 128-bit FP/SIMD registers
```

Where:

<Bt>
Is the 8-bit name of the SIMD and FP register to be transferred.

Ht
Is the 16-bit name of the SIMD and FP register to be transferred.

St
Is the 32-bit name of the SIMD and FP register to be transferred.

Dt
Is the 64-bit name of the SIMD and FP register to be transferred.

Qt
Is the 128-bit name of the SIMD and FP register to be transferred.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

simm
Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0.

Usage

Load SIMD and FP Register (unscaled offset). This instruction loads a SIMD and FP register from memory. The address that is used for the load is calculated from a base register value and an optional immediate offset.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

18.53 SCVTF (scalar, fixed-point)

Signed fixed-point Convert to Floating-point (scalar).

Syntax

```
SCVTF Hd, Wn, #fbits ; 32-bit to half-precision
SCVTF Sd, Wn, #fbits ; 32-bit to single-precision
SCVTF Dd, Wn, #fbits ; 32-bit to double-precision
SCVTF Hd, Xn, #fbits ; 64-bit to half-precision
SCVTF Sd, Xn, #fbits ; 64-bit to single-precision
SCVTF Dd, Xn, #fbits ; 64-bit to double-precision
```

Where:

Hd
Is the 16-bit name of the SIMD and FP destination register.

Wn
Is the 32-bit name of the general-purpose source register.

fbits
Depends on the instruction variant:

32-bit
Is the number of bits after the binary point in the fixed-point source, in the range 1 to 32.

64-bit
Is the number of bits after the binary point in the fixed-point source, in the range 1 to 64.

Sd
Is the 32-bit name of the SIMD and FP destination register.

Dd
Is the 64-bit name of the SIMD and FP destination register.

Xn
Is the 64-bit name of the general-purpose source register.

Operation

Signed fixed-point Convert to Floating-point (scalar). This instruction converts the signed value in the 32-bit or 64-bit general-purpose source register to a floating-point value using the rounding mode that is specified by the FPCR, and writes the result to the SIMD and FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

`Vd = signed_convertFromInt(Rn/(2^fbits))`, where *R* is either `W` or `X`.

Related references

18.1 A64 floating-point instructions in alphabetical order on page 18-1143.

18.54 SCVTF (scalar, integer)

Signed integer Convert to Floating-point (scalar).

Syntax

```
SCVTF Hd, Wn ; 32-bit to half-precision
SCVTF Sd, Wn ; 32-bit to single-precision
SCVTF Dd, Wn ; 32-bit to double-precision
SCVTF Hd, Xn ; 64-bit to half-precision
SCVTF Sd, Xn ; 64-bit to single-precision
SCVTF Dd, Xn ; 64-bit to double-precision
```

Where:

Hd
Is the 16-bit name of the SIMD and FP destination register.

Wn
Is the 32-bit name of the general-purpose source register.

Sd
Is the 32-bit name of the SIMD and FP destination register.

Dd
Is the 64-bit name of the SIMD and FP destination register.

Xn
Is the 64-bit name of the general-purpose source register.

Operation

Signed integer Convert to Floating-point (scalar). This instruction converts the signed integer value in the general-purpose source register to a floating-point value using the rounding mode that is specified by the FPCR, and writes the result to the SIMD and FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

`Vd = signed_convertFromInt(Rn)`, where `R` is either `W` or `X`.

Related references

18.1 A64 floating-point instructions in alphabetical order on page 18-1143.

18.55 STNP (SIMD and FP)

Store Pair of SIMD and FP registers, with Non-temporal hint.

Syntax

```
STNP St1, St2, [Xn|SP{, #imm}] ; 32-bit FP/SIMD registers, Signed offset

STNP Dt1, Dt2, [Xn|SP{, #imm}] ; 64-bit FP/SIMD registers, Signed offset

STNP Qt1, Qt2, [Xn|SP{, #imm}] ; 128-bit FP/SIMD registers, Signed offset
```

Where:

St1
Is the 32-bit name of the first SIMD and FP register to be transferred.

St2
Is the 32-bit name of the second SIMD and FP register to be transferred.

imm
Depends on the instruction variant:

32-bit FP/SIMD registers
Is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0.

64-bit FP/SIMD registers
Is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0.

128-bit FP/SIMD registers
Is the optional signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, defaulting to 0.

Dt1
Is the 64-bit name of the first SIMD and FP register to be transferred.

Dt2
Is the 64-bit name of the second SIMD and FP register to be transferred.

Qt1
Is the 128-bit name of the first SIMD and FP register to be transferred.

Qt2
Is the 128-bit name of the second SIMD and FP register to be transferred.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Usage

Store Pair of SIMD and FP registers, with Non-temporal hint. This instruction stores a pair of SIMD and FP registers to memory, issuing a hint to the memory system that the access is non-temporal. The address used for the store is calculated from an address from a base register value and an immediate offset. For information about non-temporal pair instructions, see *Load/Store SIMD and Floating-point Non-temporal pair* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

18.56 STP (SIMD and FP)

Store Pair of SIMD and FP registers.

Syntax

```
STP St1, St2, [Xn|SP], #imm ; 32-bit FP/SIMD registers, Post-index

STP Dt1, Dt2, [Xn|SP], #imm ; 64-bit FP/SIMD registers, Post-index

STP Qt1, Qt2, [Xn|SP], #imm ; 128-bit FP/SIMD registers, Post-index

STP St1, St2, [Xn|SP, #imm]! ; 32-bit FP/SIMD registers, Pre-index

STP Dt1, Dt2, [Xn|SP, #imm]! ; 64-bit FP/SIMD registers, Pre-index

STP Qt1, Qt2, [Xn|SP, #imm]! ; 128-bit FP/SIMD registers, Pre-index

STP St1, St2, [Xn|SP{, #imm}] ; 32-bit FP/SIMD registers, Signed offset

STP Dt1, Dt2, [Xn|SP{, #imm}] ; 64-bit FP/SIMD registers, Signed offset

STP Qt1, Qt2, [Xn|SP{, #imm}] ; 128-bit FP/SIMD registers, Signed offset
```

Where:

St1
Is the 32-bit name of the first SIMD and FP register to be transferred.

St2
Is the 32-bit name of the second SIMD and FP register to be transferred.

imm
Depends on the instruction variant:

32-bit FP/SIMD registers
Is the signed immediate byte offset, a multiple of 4 in the range -256 to 252.

64-bit FP/SIMD registers
Is the signed immediate byte offset, a multiple of 8 in the range -512 to 504.

128-bit FP/SIMD registers
Is the signed immediate byte offset, a multiple of 16 in the range -1024 to 1008.

Dt1
Is the 64-bit name of the first SIMD and FP register to be transferred.

Dt2
Is the 64-bit name of the second SIMD and FP register to be transferred.

Qt1
Is the 128-bit name of the first SIMD and FP register to be transferred.

Qt2
Is the 128-bit name of the second SIMD and FP register to be transferred.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Usage

Store Pair of SIMD and FP registers. This instruction stores a pair of SIMD and FP registers to memory. The address used for the store is calculated from a base register value and an immediate offset.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

18.57 STR (immediate, SIMD and FP)

Store SIMD and FP register (immediate offset).

Syntax

```
STR <Bt>, [Xn|SP], #simm ; 8-bit FP/SIMD registers, Post-index

STR Ht, [Xn|SP], #simm ; 16-bit FP/SIMD registers, Post-index

STR St, [Xn|SP], #simm ; 32-bit FP/SIMD registers, Post-index

STR Dt, [Xn|SP], #simm ; 64-bit FP/SIMD registers, Post-index

STR Qt, [Xn|SP], #simm ; 128-bit FP/SIMD registers, Post-index

STR <Bt>, [Xn|SP, #simm]! ; 8-bit FP/SIMD registers, Pre-index

STR Ht, [Xn|SP, #simm]! ; 16-bit FP/SIMD registers, Pre-index

STR St, [Xn|SP, #simm]! ; 32-bit FP/SIMD registers, Pre-index

STR Dt, [Xn|SP, #simm]! ; 64-bit FP/SIMD registers, Pre-index

STR Qt, [Xn|SP, #simm]! ; 128-bit FP/SIMD registers, Pre-index

STR <Bt>, [Xn|SP{, #pimm}] ; 8-bit FP/SIMD registers

STR Ht, [Xn|SP{, #pimm}] ; 16-bit FP/SIMD registers

STR St, [Xn|SP{, #pimm}] ; 32-bit FP/SIMD registers

STR Dt, [Xn|SP{, #pimm}] ; 64-bit FP/SIMD registers

STR Qt, [Xn|SP{, #pimm}] ; 128-bit FP/SIMD registers
```

Where:

<Bt>
Is the 8-bit name of the SIMD and FP register to be transferred.

simm
Is the signed immediate byte offset, in the range -256 to 255.

Ht
Is the 16-bit name of the SIMD and FP register to be transferred.

St
Is the 32-bit name of the SIMD and FP register to be transferred.

Dt
Is the 64-bit name of the SIMD and FP register to be transferred.

Qt
Is the 128-bit name of the SIMD and FP register to be transferred.

pimm
Depends on the instruction variant:

8-bit FP/SIMD registers
Is the optional positive immediate byte offset, in the range 0 to 4095, defaulting to 0.

16-bit FP/SIMD registers
Is the optional positive immediate byte offset, a multiple of 2 in the range 0 to 8190, defaulting to 0.

32-bit FP/SIMD registers
Is the optional positive immediate byte offset, a multiple of 4 in the range 0 to 16380, defaulting to 0.

64-bit FP/SIMD registers
Is the optional positive immediate byte offset, a multiple of 8 in the range 0 to 32760, defaulting to 0.

128-bit FP/SIMD registers

Is the optional positive immediate byte offset, a multiple of 16 in the range 0 to 65520, defaulting to 0.

Xn\|SP

Is the 64-bit name of the general-purpose base register or stack pointer.

Usage

Store SIMD and FP register (immediate offset). This instruction stores a single SIMD and FP register to memory. The address that is used for the store is calculated from a base register value and an immediate offset.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

18.58 STR (register, SIMD and FP)

Store SIMD and FP register (register offset).

Syntax

```
STR <Bt>, [Xn|SP, (Wm|Xm), extend {amount}] ; 8-bit FP/SIMD registers

STR <Bt>, [Xn|SP, Xm{, LSL amount}] ; 8-bit FP/SIMD registers

STR Ht, [Xn|SP, (Wm|Xm){, extend {amount}}] ; 16-bit FP/SIMD registers

STR St, [Xn|SP, (Wm|Xm){, extend {amount}}] ; 32-bit FP/SIMD registers

STR Dt, [Xn|SP, (Wm|Xm){, extend {amount}}] ; 64-bit FP/SIMD registers

STR Qt, [Xn|SP, (Wm|Xm){, extend {amount}}] ; 128-bit FP/SIMD registers
```

Where:

<Bt>
Is the 8-bit name of the SIMD and FP register to be transferred.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Wm
When "option<0>" is set to 0, is the 32-bit name of the general-purpose index register.

Xm
When "option<0>" is set to 1, is the 64-bit name of the general-purpose index register.

extend
Is the index extend specifier:

8-bit FP/SIMD registers
Can be one of `UXTW`, `SXTW` or `SXTX`.

16-bit FP/SIMD registers
Can be one of `UXTW`, `LSL`, `SXTW` or `SXTX`.

amount
Is the index shift amount, it must be.

Ht
Is the 16-bit name of the SIMD and FP register to be transferred.

St
Is the 32-bit name of the SIMD and FP register to be transferred.

Dt
Is the 64-bit name of the SIMD and FP register to be transferred.

Qt
Is the 128-bit name of the SIMD and FP register to be transferred.

Usage

Store SIMD and FP register (register offset). This instruction stores a single SIMD and FP register to memory. The address that is used for the store is calculated from a base register value and an offset register value. The offset can be optionally shifted and extended.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

18.59 STUR (SIMD and FP)

Store SIMD and FP register (unscaled offset).

Syntax

```
STUR <Bt>, [Xn|SP{, #simm}] ; 8-bit FP/SIMD registers

STUR Ht, [Xn|SP{, #simm}] ; 16-bit FP/SIMD registers

STUR St, [Xn|SP{, #simm}] ; 32-bit FP/SIMD registers

STUR Dt, [Xn|SP{, #simm}] ; 64-bit FP/SIMD registers

STUR Qt, [Xn|SP{, #simm}] ; 128-bit FP/SIMD registers
```

Where:

<Bt>
Is the 8-bit name of the SIMD and FP register to be transferred.

Ht
Is the 16-bit name of the SIMD and FP register to be transferred.

St
Is the 32-bit name of the SIMD and FP register to be transferred.

Dt
Is the 64-bit name of the SIMD and FP register to be transferred.

Qt
Is the 128-bit name of the SIMD and FP register to be transferred.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

simm
Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0.

Usage

Store SIMD and FP register (unscaled offset). This instruction stores a single SIMD and FP register to memory. The address that is used for the store is calculated from a base register value and an optional immediate offset.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

17.1 A64 data transfer instructions in alphabetical order on page 17-994.

18.60 UCVTF (scalar, fixed-point)

Unsigned fixed-point Convert to Floating-point (scalar).

Syntax

```
UCVTF Hd, Wn, #fbits ; 32-bit to half-precision
UCVTF Sd, Wn, #fbits ; 32-bit to single-precision
UCVTF Dd, Wn, #fbits ; 32-bit to double-precision
UCVTF Hd, Xn, #fbits ; 64-bit to half-precision
UCVTF Sd, Xn, #fbits ; 64-bit to single-precision
UCVTF Dd, Xn, #fbits ; 64-bit to double-precision
```

Where:

Hd
Is the 16-bit name of the SIMD and FP destination register.

Wn
Is the 32-bit name of the general-purpose source register.

fbits
Depends on the instruction variant:

32-bit
Is the number of bits after the binary point in the fixed-point source, in the range 1 to 32.

64-bit
Is the number of bits after the binary point in the fixed-point source, in the range 1 to 64.

Sd
Is the 32-bit name of the SIMD and FP destination register.

Dd
Is the 64-bit name of the SIMD and FP destination register.

Xn
Is the 64-bit name of the general-purpose source register.

Operation

Unsigned fixed-point Convert to Floating-point (scalar). This instruction converts the unsigned value in the 32-bit or 64-bit general-purpose source register to a floating-point value using the rounding mode that is specified by the FPCR, and writes the result to the SIMD and FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

`Vd = unsigned_convertFromInt(Rn/(2^fbits))`, where *R* is either `W` or `X`.

Related references

18.1 A64 floating-point instructions in alphabetical order on page 18-1143.

18.61 UCVTF (scalar, integer)

Unsigned integer Convert to Floating-point (scalar).

Syntax

```
UCVTF Hd, Wn ; 32-bit to half-precision
UCVTF Sd, Wn ; 32-bit to single-precision
UCVTF Dd, Wn ; 32-bit to double-precision
UCVTF Hd, Xn ; 64-bit to half-precision
UCVTF Sd, Xn ; 64-bit to single-precision
UCVTF Dd, Xn ; 64-bit to double-precision
```

Where:

Hd
Is the 16-bit name of the SIMD and FP destination register.

Wn
Is the 32-bit name of the general-purpose source register.

Sd
Is the 32-bit name of the SIMD and FP destination register.

Dd
Is the 64-bit name of the SIMD and FP destination register.

Xn
Is the 64-bit name of the general-purpose source register.

Operation

Unsigned integer Convert to Floating-point (scalar). This instruction converts the unsigned integer value in the general-purpose source register to a floating-point value using the rounding mode that is specified by the FPCR, and writes the result to the SIMD and FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

`Vd = unsigned_convertFromInt(Rn)`, where `R` is either `W` or `X`.

Related references

18.1 A64 floating-point instructions in alphabetical order on page 18-1143.

Chapter 19
A64 SIMD Scalar Instructions

Describes the A64 SIMD scalar instructions.

It contains the following sections:

19.1 A64 SIMD scalar instructions in alphabetical order

A summary of the A64 SIMD scalar instructions that are supported.

Table 19-1 Summary of A64 SIMD scalar instructions

Mnemonic	Brief description	See
ABS (scalar)	Absolute value (vector)	*19.2 ABS (scalar)* on page 19-1220
ADD (scalar)	Add (vector)	*19.3 ADD (scalar)* on page 19-1221
ADDP (scalar)	Add Pair of elements (scalar)	*19.4 ADDP (scalar)* on page 19-1222
CMEQ (scalar, register)	Compare bitwise Equal (vector)	*19.5 CMEQ (scalar, register)* on page 19-1223
CMEQ (scalar, zero)	Compare bitwise Equal to zero (vector)	*19.6 CMEQ (scalar, zero)* on page 19-1224
CMGE (scalar, register)	Compare signed Greater than or Equal (vector)	*19.7 CMGE (scalar, register)* on page 19-1225
CMGE (scalar, zero)	Compare signed Greater than or Equal to zero (vector)	*19.8 CMGE (scalar, zero)* on page 19-1226
CMGT (scalar, register)	Compare signed Greater than (vector)	*19.9 CMGT (scalar, register)* on page 19-1227
CMGT (scalar, zero)	Compare signed Greater than zero (vector)	*19.10 CMGT (scalar, zero)* on page 19-1228
CMHI (scalar, register)	Compare unsigned Higher (vector)	*19.11 CMHI (scalar, register)* on page 19-1229
CMHS (scalar, register)	Compare unsigned Higher or Same (vector)	*19.12 CMHS (scalar, register)* on page 19-1230
CMLE (scalar, zero)	Compare signed Less than or Equal to zero (vector)	*19.13 CMLE (scalar, zero)* on page 19-1231
CMLT (scalar, zero)	Compare signed Less than zero (vector)	*19.14 CMLT (scalar, zero)* on page 19-1232
CMTST (scalar)	Compare bitwise Test bits nonzero (vector)	*19.15 CMTST (scalar)* on page 19-1233
DUP (scalar, element)	Duplicate vector element to scalar	*19.16 DUP (scalar, element)* on page 19-1234
FABD (scalar)	Floating-point Absolute Difference (vector)	*19.17 FABD (scalar)* on page 19-1235
FACGE (scalar)	Floating-point Absolute Compare Greater than or Equal (vector)	*19.18 FACGE (scalar)* on page 19-1236
FACGT (scalar)	Floating-point Absolute Compare Greater than (vector)	*19.19 FACGT (scalar)* on page 19-1237
FADDP (scalar)	Floating-point Add Pair of elements (scalar)	*19.20 FADDP (scalar)* on page 19-1238
FCMEQ (scalar, register)	Floating-point Compare Equal (vector)	*19.21 FCMEQ (scalar, register)* on page 19-1239
FCMEQ (scalar, zero)	Floating-point Compare Equal to zero (vector)	*19.22 FCMEQ (scalar, zero)* on page 19-1240
FCMGE (scalar, register)	Floating-point Compare Greater than or Equal (vector)	*19.23 FCMGE (scalar, register)* on page 19-1241
FCMGE (scalar, zero)	Floating-point Compare Greater than or Equal to zero (vector)	*19.24 FCMGE (scalar, zero)* on page 19-1242
FCMGT (scalar, register)	Floating-point Compare Greater than (vector)	*19.25 FCMGT (scalar, register)* on page 19-1243
FCMGT (scalar, zero)	Floating-point Compare Greater than zero (vector)	*19.26 FCMGT (scalar, zero)* on page 19-1244

Table 19-1 Summary of A64 SIMD scalar instructions (continued)

Mnemonic	Brief description	See
FCMLA (scalar, by element)	Floating-point Complex Multiply Accumulate (by element)	*19.27 FCMLA (scalar, by element)* on page 19-1245
FCMLE (scalar, zero)	Floating-point Compare Less than or Equal to zero (vector)	*19.28 FCMLE (scalar, zero)* on page 19-1247
FCMLT (scalar, zero)	Floating-point Compare Less than zero (vector)	*19.29 FCMLT (scalar, zero)* on page 19-1248
FCVTAS (scalar)	Floating-point Convert to Signed integer, rounding to nearest with ties to Away (vector)	*19.30 FCVTAS (scalar)* on page 19-1249
FCVTAU (scalar)	Floating-point Convert to Unsigned integer, rounding to nearest with ties to Away (vector)	*19.31 FCVTAU (scalar)* on page 19-1250
FCVTMS (scalar)	Floating-point Convert to Signed integer, rounding toward Minus infinity (vector)	*19.32 FCVTMS (scalar)* on page 19-1251
FCVTMU (scalar)	Floating-point Convert to Unsigned integer, rounding toward Minus infinity (vector)	*19.33 FCVTMU (scalar)* on page 19-1252
FCVTNS (scalar)	Floating-point Convert to Signed integer, rounding to nearest with ties to even (vector)	*19.34 FCVTNS (scalar)* on page 19-1253
FCVTNU (scalar)	Floating-point Convert to Unsigned integer, rounding to nearest with ties to even (vector)	*19.35 FCVTNU (scalar)* on page 19-1254
FCVTPS (scalar)	Floating-point Convert to Signed integer, rounding toward Plus infinity (vector)	*19.36 FCVTPS (scalar)* on page 19-1255
FCVTPU (scalar)	Floating-point Convert to Unsigned integer, rounding toward Plus infinity (vector)	*19.37 FCVTPU (scalar)* on page 19-1256
FCVTXN (scalar)	Floating-point Convert to lower precision Narrow, rounding to odd (vector)	*19.38 FCVTXN (scalar)* on page 19-1257
FCVTZS (scalar, fixed-point)	Floating-point Convert to Signed fixed-point, rounding toward Zero (vector)	*19.39 FCVTZS (scalar, fixed-point)* on page 19-1258
FCVTZS (scalar, integer)	Floating-point Convert to Signed integer, rounding toward Zero (vector)	*19.40 FCVTZS (scalar, integer)* on page 19-1259
FCVTZU (scalar, fixed-point)	Floating-point Convert to Unsigned fixed-point, rounding toward Zero (vector)	*19.41 FCVTZU (scalar, fixed-point)* on page 19-1260
FCVTZU (scalar, integer)	Floating-point Convert to Unsigned integer, rounding toward Zero (vector)	*19.42 FCVTZU (scalar, integer)* on page 19-1261
FMAXNMP (scalar)	Floating-point Maximum Number of Pair of elements (scalar)	*19.43 FMAXNMP (scalar)* on page 19-1262
FMAXP (scalar)	Floating-point Maximum of Pair of elements (scalar)	*19.44 FMAXP (scalar)* on page 19-1263
FMINNMP (scalar)	Floating-point Minimum Number of Pair of elements (scalar)	*19.45 FMINNMP (scalar)* on page 19-1264
FMINP (scalar)	Floating-point Minimum of Pair of elements (scalar)	*19.46 FMINP (scalar)* on page 19-1265
FMLA (scalar, by element)	Floating-point fused Multiply-Add to accumulator (by element)	*19.47 FMLA (scalar, by element)* on page 19-1266

Table 19-1 Summary of A64 SIMD scalar instructions (continued)

Mnemonic	Brief description	See
FMLS (scalar, by element)	Floating-point fused Multiply-Subtract from accumulator (by element)	*19.48 FMLS (scalar, by element)* on page 19-1267
FMUL (scalar, by element)	Floating-point Multiply (by element)	*19.49 FMUL (scalar, by element)* on page 19-1268
FMULX (scalar, by element)	Floating-point Multiply extended (by element)	*19.50 FMULX (scalar, by element)* on page 19-1269
FMULX (scalar)	Floating-point Multiply extended	*19.51 FMULX (scalar)* on page 19-1270
FRECPE (scalar)	Floating-point Reciprocal Estimate	*19.52 FRECPE (scalar)* on page 19-1271
FRECPS (scalar)	Floating-point Reciprocal Step	*19.53 FRECPS (scalar)* on page 19-1272
FRSQRTE (scalar)	Floating-point Reciprocal Square Root Estimate	*19.54 FRSQRTE (scalar)* on page 19-1273
FRSQRTS (scalar)	Floating-point Reciprocal Square Root Step	*19.55 FRSQRTS (scalar)* on page 19-1274
MOV (scalar)	Move vector element to scalar	*19.56 MOV (scalar)* on page 19-1275
NEG (scalar)	Negate (vector)	*19.57 NEG (scalar)* on page 19-1276
SCVTF (scalar, fixed-point)	Signed fixed-point Convert to Floating-point (vector)	*19.58 SCVTF (scalar, fixed-point)* on page 19-1277
SCVTF (scalar, integer)	Signed integer Convert to Floating-point (vector)	*19.59 SCVTF (scalar, integer)* on page 19-1278
SHL (scalar)	Shift Left (immediate)	*19.60 SHL (scalar)* on page 19-1279
SLI (scalar)	Shift Left and Insert (immediate)	*19.61 SLI (scalar)* on page 19-1280
SQABS (scalar)	Signed saturating Absolute value	*19.62 SQABS (scalar)* on page 19-1281
SQADD (scalar)	Signed saturating Add	*19.63 SQADD (scalar)* on page 19-1282
SQDMLAL (scalar, by element)	Signed saturating Doubling Multiply-Add Long (by element)	*19.64 SQDMLAL (scalar, by element)* on page 19-1283
SQDMLAL (scalar)	Signed saturating Doubling Multiply-Add Long	*19.65 SQDMLAL (scalar)* on page 19-1284
SQDMLSL (scalar, by element)	Signed saturating Doubling Multiply-Subtract Long (by element)	*19.66 SQDMLSL (scalar, by element)* on page 19-1285
SQDMLSL (scalar)	Signed saturating Doubling Multiply-Subtract Long	*19.67 SQDMLSL (scalar)* on page 19-1286
SQDMULH (scalar, by element)	Signed saturating Doubling Multiply returning High half (by element)	*19.68 SQDMULH (scalar, by element)* on page 19-1287
SQDMULH (scalar)	Signed saturating Doubling Multiply returning High half	*19.69 SQDMULH (scalar)* on page 19-1288
SQDMULL (scalar, by element)	Signed saturating Doubling Multiply Long (by element)	*19.70 SQDMULL (scalar, by element)* on page 19-1289
SQDMULL (scalar)	Signed saturating Doubling Multiply Long	*19.71 SQDMULL (scalar)* on page 19-1290
SQNEG (scalar)	Signed saturating Negate	*19.72 SQNEG (scalar)* on page 19-1291

Table 19-1 Summary of A64 SIMD scalar instructions (continued)

Mnemonic	Brief description	See
SQRDMLAH (scalar, by element)	Signed Saturating Rounding Doubling Multiply Accumulate returning High Half (by element)	*19.73 SQRDMLAH (scalar, by element)* on page 19-1292
SQRDMLAH (scalar)	Signed Saturating Rounding Doubling Multiply Accumulate returning High Half (vector)	*19.74 SQRDMLAH (scalar)* on page 19-1293
SQRDMLSH (scalar, by element)	Signed Saturating Rounding Doubling Multiply Subtract returning High Half (by element)	*19.75 SQRDMLSH (scalar, by element)* on page 19-1294
SQRDMLSH (scalar)	Signed Saturating Rounding Doubling Multiply Subtract returning High Half (vector)	*19.76 SQRDMLSH (scalar)* on page 19-1295
SQRDMULH (scalar, by element)	Signed saturating Rounding Doubling Multiply returning High half (by element)	*19.77 SQRDMULH (scalar, by element)* on page 19-1296
SQRDMULH (scalar)	Signed saturating Rounding Doubling Multiply returning High half	*19.78 SQRDMULH (scalar)* on page 19-1297
SQRSHL (scalar)	Signed saturating Rounding Shift Left (register)	*19.79 SQRSHL (scalar)* on page 19-1298
SQRSHRN (scalar)	Signed saturating Rounded Shift Right Narrow (immediate)	*19.80 SQRSHRN (scalar)* on page 19-1299
SQRSHRUN (scalar)	Signed saturating Rounded Shift Right Unsigned Narrow (immediate)	*19.81 SQRSHRUN (scalar)* on page 19-1300
SQSHL (scalar, immediate)	Signed saturating Shift Left (immediate)	*19.82 SQSHL (scalar, immediate)* on page 19-1301
SQSHL (scalar, register)	Signed saturating Shift Left (register)	*19.83 SQSHL (scalar, register)* on page 19-1302
SQSHLU (scalar)	Signed saturating Shift Left Unsigned (immediate)	*19.84 SQSHLU (scalar)* on page 19-1303
SQSHRN (scalar)	Signed saturating Shift Right Narrow (immediate)	*19.85 SQSHRN (scalar)* on page 19-1304
SQSHRUN (scalar)	Signed saturating Shift Right Unsigned Narrow (immediate)	*19.86 SQSHRUN (scalar)* on page 19-1305
SQSUB (scalar)	Signed saturating Subtract	*19.87 SQSUB (scalar)* on page 19-1306
SQXTN (scalar)	Signed saturating extract Narrow	*19.88 SQXTN (scalar)* on page 19-1307
SQXTUN (scalar)	Signed saturating extract Unsigned Narrow	*19.89 SQXTUN (scalar)* on page 19-1308
SRI (scalar)	Shift Right and Insert (immediate)	*19.90 SRI (scalar)* on page 19-1309
SRSHL (scalar)	Signed Rounding Shift Left (register)	*19.91 SRSHL (scalar)* on page 19-1310
SRSHR (scalar)	Signed Rounding Shift Right (immediate)	*19.92 SRSHR (scalar)* on page 19-1311
SRSRA (scalar)	Signed Rounding Shift Right and Accumulate (immediate)	*19.93 SRSRA (scalar)* on page 19-1312
SSHL (scalar)	Signed Shift Left (register)	*19.94 SSHL (scalar)* on page 19-1313
SSHR (scalar)	Signed Shift Right (immediate)	*19.95 SSHR (scalar)* on page 19-1314

Table 19-1 Summary of A64 SIMD scalar instructions (continued)

Mnemonic	Brief description	See
SSRA (scalar)	Signed Shift Right and Accumulate (immediate)	*19.96 SSRA (scalar)* on page 19-1315
SUB (scalar)	Subtract (vector)	*19.97 SUB (scalar)* on page 19-1316
SUQADD (scalar)	Signed saturating Accumulate of Unsigned value	*19.98 SUQADD (scalar)* on page 19-1317
UCVTF (scalar, fixed-point)	Unsigned fixed-point Convert to Floating-point (vector)	*19.99 UCVTF (scalar, fixed-point)* on page 19-1318
UCVTF (scalar, integer)	Unsigned integer Convert to Floating-point (vector)	*19.100 UCVTF (scalar, integer)* on page 19-1319
UQADD (scalar)	Unsigned saturating Add	*19.101 UQADD (scalar)* on page 19-1320
UQRSHL (scalar)	Unsigned saturating Rounding Shift Left (register)	*19.102 UQRSHL (scalar)* on page 19-1321
UQRSHRN (scalar)	Unsigned saturating Rounded Shift Right Narrow (immediate)	*19.103 UQRSHRN (scalar)* on page 19-1322
UQSHL (scalar, immediate)	Unsigned saturating Shift Left (immediate)	*19.104 UQSHL (scalar, immediate)* on page 19-1323
UQSHL (scalar, register)	Unsigned saturating Shift Left (register)	*19.105 UQSHL (scalar, register)* on page 19-1324
UQSHRN (scalar)	Unsigned saturating Shift Right Narrow (immediate)	*19.106 UQSHRN (scalar)* on page 19-1325
UQSUB (scalar)	Unsigned saturating Subtract	*19.107 UQSUB (scalar)* on page 19-1326
UQXTN (scalar)	Unsigned saturating extract Narrow	*19.108 UQXTN (scalar)* on page 19-1327
URSHL (scalar)	Unsigned Rounding Shift Left (register)	*19.109 URSHL (scalar)* on page 19-1328
URSHR (scalar)	Unsigned Rounding Shift Right (immediate)	*19.110 URSHR (scalar)* on page 19-1329
URSRA (scalar)	Unsigned Rounding Shift Right and Accumulate (immediate)	*19.111 URSRA (scalar)* on page 19-1330
USHL (scalar)	Unsigned Shift Left (register)	*19.112 USHL (scalar)* on page 19-1331
USHR (scalar)	Unsigned Shift Right (immediate)	*19.113 USHR (scalar)* on page 19-1332
USQADD (scalar)	Unsigned saturating Accumulate of Signed value	*19.114 USQADD (scalar)* on page 19-1333
USRA (scalar)	Unsigned Shift Right and Accumulate (immediate)	*19.115 USRA (scalar)* on page 19-1334

19.2 ABS (scalar)

Absolute value (vector).

Syntax

```
ABS Vd, Vn
```

Where:

V
: Is a width specifier, `D`.

d
: Is the number of the SIMD and FP destination register.

n
: Is the number of the SIMD and FP source register.

Usage

Absolute value (vector). This instruction calculates the absolute value of each vector element in the source SIMD and FP register, puts the result into a vector, and writes the vector to the destination SIMD and FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.3 ADD (scalar)

Add (vector).

Syntax

`ADD Vd, Vn, Vm`

Where:

V
Is a width specifier, `D`.

d
Is the number of the SIMD and FP destination register.

n
Is the number of the first SIMD and FP source register.

m
Is the number of the second SIMD and FP source register.

Usage

Add (vector). This instruction adds corresponding elements in the two source SIMD and FP registers, places the results into a vector, and writes the vector to the destination SIMD and FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.4 ADDP (scalar)

Add Pair of elements (scalar).

Syntax

`ADDP Vd, Vn.T`

Where:

V
Is the destination width specifier, `D`.

d
Is the number of the SIMD and FP destination register.

Vn
Is the name of the SIMD and FP source register.

T
Is the source arrangement specifier, `2D`.

Usage

Add Pair of elements (scalar). This instruction adds two vector elements in the source SIMD and FP register and writes the scalar result into the destination SIMD and FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.5 CMEQ (scalar, register)

Compare bitwise Equal (vector).

Syntax

`CMEQ Vd, Vn, Vm`

Where:

V
Is a width specifier, `D`.

d
Is the number of the SIMD and FP destination register.

n
Is the number of the first SIMD and FP source register.

m
Is the number of the second SIMD and FP source register.

Usage

Compare bitwise Equal (vector). This instruction compares each vector element from the first source SIMD and FP register with the corresponding vector element from the second source SIMD and FP register, and if the comparison is equal sets every bit of the corresponding vector element in the destination SIMD and FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD and FP register to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.6 CMEQ (scalar, zero)

Compare bitwise Equal to zero (vector).

Syntax

`CMEQ Vd, Vn, #0`

Where:

V
Is a width specifier, `D`.

d
Is the number of the SIMD and FP destination register.

n
Is the number of the SIMD and FP source register.

Usage

Compare bitwise Equal to zero (vector). This instruction reads each vector element in the source SIMD and FP register and if the value is equal to zero sets every bit of the corresponding vector element in the destination SIMD and FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD and FP register to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.7 CMGE (scalar, register)

Compare signed Greater than or Equal (vector).

Syntax

`CMGE Vd, Vn, Vm`

Where:

V
Is a width specifier, `D`.

d
Is the number of the SIMD and FP destination register.

n
Is the number of the first SIMD and FP source register.

m
Is the number of the second SIMD and FP source register.

Usage

Compare signed Greater than or Equal (vector). This instruction compares each vector element in the first source SIMD and FP register with the corresponding vector element in the second source SIMD and FP register and if the first signed integer value is greater than or equal to the second signed integer value sets every bit of the corresponding vector element in the destination SIMD and FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD and FP register to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.8 CMGE (scalar, zero)

Compare signed Greater than or Equal to zero (vector).

Syntax

```
CMGE Vd, Vn, #0
```

Where:

V
Is a width specifier, `D`.

d
Is the number of the SIMD and FP destination register.

n
Is the number of the SIMD and FP source register.

Usage

Compare signed Greater than or Equal to zero (vector). This instruction reads each vector element in the source SIMD and FP register and if the signed integer value is greater than or equal to zero sets every bit of the corresponding vector element in the destination SIMD and FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD and FP register to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.9 CMGT (scalar, register)

Compare signed Greater than (vector).

Syntax

`CMGT Vd, Vn, Vm`

Where:

V
Is a width specifier, `D`.

d
Is the number of the SIMD and FP destination register.

n
Is the number of the first SIMD and FP source register.

m
Is the number of the second SIMD and FP source register.

Usage

Compare signed Greater than (vector). This instruction compares each vector element in the first source SIMD and FP register with the corresponding vector element in the second source SIMD and FP register and if the first signed integer value is greater than the second signed integer value sets every bit of the corresponding vector element in the destination SIMD and FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD and FP register to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.10 CMGT (scalar, zero)

Compare signed Greater than zero (vector).

Syntax

```
CMGT Vd, Vn, #0
```

Where:

V
Is a width specifier, `D`.

d
Is the number of the SIMD and FP destination register.

n
Is the number of the SIMD and FP source register.

Usage

Compare signed Greater than zero (vector). This instruction reads each vector element in the source SIMD and FP register and if the signed integer value is greater than zero sets every bit of the corresponding vector element in the destination SIMD and FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD and FP register to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.11 CMHI (scalar, register)

Compare unsigned Higher (vector).

Syntax

`CMHI Vd, Vn, Vm`

Where:

V
Is a width specifier, `D`.

d
Is the number of the SIMD and FP destination register.

n
Is the number of the first SIMD and FP source register.

m
Is the number of the second SIMD and FP source register.

Usage

Compare unsigned Higher (vector). This instruction compares each vector element in the first source SIMD and FP register with the corresponding vector element in the second source SIMD and FP register and if the first unsigned integer value is greater than the second unsigned integer value sets every bit of the corresponding vector element in the destination SIMD and FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD and FP register to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.12 CMHS (scalar, register)

Compare unsigned Higher or Same (vector).

Syntax

`CMHS Vd, Vn, Vm`

Where:

V
Is a width specifier, `D`.

d
Is the number of the SIMD and FP destination register.

n
Is the number of the first SIMD and FP source register.

m
Is the number of the second SIMD and FP source register.

Usage

Compare unsigned Higher or Same (vector). This instruction compares each vector element in the first source SIMD and FP register with the corresponding vector element in the second source SIMD and FP register and if the first unsigned integer value is greater than or equal to the second unsigned integer value sets every bit of the corresponding vector element in the destination SIMD and FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD and FP register to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.13 CMLE (scalar, zero)

Compare signed Less than or Equal to zero (vector).

Syntax

`CMLE Vd, Vn, #0`

Where:

V
Is a width specifier, `D`.

d
Is the number of the SIMD and FP destination register.

n
Is the number of the SIMD and FP source register.

Usage

Compare signed Less than or Equal to zero (vector). This instruction reads each vector element in the source SIMD and FP register and if the signed integer value is less than or equal to zero sets every bit of the corresponding vector element in the destination SIMD and FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD and FP register to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.14 CMLT (scalar, zero)

Compare signed Less than zero (vector).

Syntax

`CMLT Vd, Vn, #0`

Where:

V
Is a width specifier, `D`.

d
Is the number of the SIMD and FP destination register.

n
Is the number of the SIMD and FP source register.

Usage

Compare signed Less than zero (vector). This instruction reads each vector element in the source SIMD and FP register and if the signed integer value is less than zero sets every bit of the corresponding vector element in the destination SIMD and FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD and FP register to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.15 CMTST (scalar)

Compare bitwise Test bits nonzero (vector).

Syntax

`CMTST Vd, Vn, Vm`

Where:

V
Is a width specifier, `D`.

d
Is the number of the SIMD and FP destination register.

n
Is the number of the first SIMD and FP source register.

m
Is the number of the second SIMD and FP source register.

Usage

Compare bitwise Test bits nonzero (vector). This instruction reads each vector element in the first source SIMD and FP register, performs an AND with the corresponding vector element in the second source SIMD and FP register, and if the result is not zero, sets every bit of the corresponding vector element in the destination SIMD and FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD and FP register to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.16 DUP (scalar, element)

Duplicate vector element to scalar.

This instruction is used by the alias `MOV` (scalar).

Syntax

```
DUP Vd, Vn.T[index]
```

Where:

V
Is the destination width specifier, and can be one of the values shown in Usage.

d
Is the number of the SIMD and FP destination register.

T
Is the element width specifier, and can be one of the values shown in Usage.

Vn
Is the name of the SIMD and FP source register.

index
Is the element index, in the range shown in Usage.

Usage

Duplicate vector element to vector or scalar. This instruction duplicates the vector element at the specified element index in the source SIMD and FP register into a scalar or each element in a vector, and writes the result to the destination SIMD and FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 19-2 DUP (Scalar) specifier combinations

V	*T*	*index*
B	B	0 to 15
H	H	0 to 7
S	S	0 to 3
D	D	0 or 1

Related references

19.56 MOV (scalar) on page 19-1275.
19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.17 FABD (scalar)

Floating-point Absolute Difference (vector).

Syntax

```
FABD Hd, Hn, Hm ; Scalar half precision

FABD Vd, Vn, Vm ; Scalar single-precision and double-precision
```

Where:

Hd
Is the 16-bit name of the SIMD and FP destination register.

Hn
Is the 16-bit name of the first SIMD and FP source register.

Hm
Is the 16-bit name of the second SIMD and FP source register.

V
Is a width specifier, and can be either `S` or `D`.

d
Is the number of the SIMD and FP destination register.

n
Is the number of the first SIMD and FP source register.

m
Is the number of the second SIMD and FP source register.

Architectures supported (scalar)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point Absolute Difference (vector). This instruction subtracts the floating-point values in the elements of the second source SIMD and FP register, from the corresponding floating-point values in the elements of the first source SIMD and FP register, places the absolute value of each result in a vector, and writes the vector to the destination SIMD and FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.18 FACGE (scalar)

Floating-point Absolute Compare Greater than or Equal (vector).

Syntax

```
FACGE Hd, Hn, Hm ; Scalar half precision

FACGE Vd, Vn, Vm ; Scalar single-precision and double-precision
```

Where:

Hd
Is the 16-bit name of the SIMD and FP destination register.

Hn
Is the 16-bit name of the first SIMD and FP source register.

Hm
Is the 16-bit name of the second SIMD and FP source register.

V
Is a width specifier, and can be either `S` or `D`.

d
Is the number of the SIMD and FP destination register.

n
Is the number of the first SIMD and FP source register.

m
Is the number of the second SIMD and FP source register.

Architectures supported (scalar)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point Absolute Compare Greater than or Equal (vector). This instruction compares the absolute value of each floating-point value in the first source SIMD and FP register with the absolute value of the corresponding floating-point value in the second source SIMD and FP register and if the first value is greater than or equal to the second value sets every bit of the corresponding vector element in the destination SIMD and FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD and FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.19 FACGT (scalar)

Floating-point Absolute Compare Greater than (vector).

Syntax

```
FACGT Hd, Hn, Hm ; Scalar half precision

FACGT Vd, Vn, Vm ; Scalar single-precision and double-precision
```

Where:

Hd
Is the 16-bit name of the SIMD and FP destination register.

Hn
Is the 16-bit name of the first SIMD and FP source register.

Hm
Is the 16-bit name of the second SIMD and FP source register.

V
Is a width specifier, and can be either `S` or `D`.

d
Is the number of the SIMD and FP destination register.

n
Is the number of the first SIMD and FP source register.

m
Is the number of the second SIMD and FP source register.

Architectures supported (scalar)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point Absolute Compare Greater than (vector). This instruction compares the absolute value of each vector element in the first source SIMD and FP register with the absolute value of the corresponding vector element in the second source SIMD and FP register and if the first value is greater than the second value sets every bit of the corresponding vector element in the destination SIMD and FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD and FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.20 FADDP (scalar)

Floating-point Add Pair of elements (scalar).

Syntax

```
FADDP Vd, Vn.T ; Half-precision

FADDP Vd, Vn.T ; Single-precision and double-precision
```

Where:

V

Is the destination width specifier:

Half-precision
Must be `H`.

Single-precision and double-precision
Can be one of `S` or `D`.

T

Is the source arrangement specifier:

Half-precision
Must be `2H`.

Single-precision and double-precision
Can be one of `2S` or `2D`.

d

Is the number of the SIMD and FP destination register.

Vn

Is the name of the SIMD and FP source register.

Architectures supported (scalar)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point Add Pair of elements (scalar). This instruction adds two floating-point vector elements in the source SIMD and FP register and writes the scalar result into the destination SIMD and FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.21 FCMEQ (scalar, register)

Floating-point Compare Equal (vector).

Syntax

```
FCMEQ Hd, Hn, Hm ; Scalar half precision

FCMEQ Vd, Vn, Vm ; Scalar single-precision and double-precision
```

Where:

Hd
Is the 16-bit name of the SIMD and FP destination register.

Hn
Is the 16-bit name of the first SIMD and FP source register.

Hm
Is the 16-bit name of the second SIMD and FP source register.

V
Is a width specifier, and can be either `S` or `D`.

d
Is the number of the SIMD and FP destination register.

n
Is the number of the first SIMD and FP source register.

m
Is the number of the second SIMD and FP source register.

Architectures supported (scalar)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point Compare Equal (vector). This instruction compares each floating-point value from the first source SIMD and FP register, with the corresponding floating-point value from the second source SIMD and FP register, and if the comparison is equal sets every bit of the corresponding vector element in the destination SIMD and FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD and FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.22 FCMEQ (scalar, zero)

Floating-point Compare Equal to zero (vector).

Syntax

```
FCMEQ Hd, Hn, #0.0 ; Scalar half precision
FCMEQ Vd, Vn, #0.0 ; Scalar single-precision and double-precision
```

Where:

Hd
Is the 16-bit name of the SIMD and FP destination register.

Hn
Is the 16-bit name of the SIMD and FP source register.

V
Is a width specifier, and can be either `S` or `D`.

d
Is the number of the SIMD and FP destination register.

n
Is the number of the SIMD and FP source register.

Architectures supported (scalar)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point Compare Equal to zero (vector). This instruction reads each floating-point value in the source SIMD and FP register and if the value is equal to zero sets every bit of the corresponding vector element in the destination SIMD and FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD and FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.23 FCMGE (scalar, register)

Floating-point Compare Greater than or Equal (vector).

Syntax

```
FCMGE Hd, Hn, Hm ; Scalar half precision

FCMGE Vd, Vn, Vm ; Scalar single-precision and double-precision
```

Where:

Hd
Is the 16-bit name of the SIMD and FP destination register.

Hn
Is the 16-bit name of the first SIMD and FP source register.

Hm
Is the 16-bit name of the second SIMD and FP source register.

V
Is a width specifier, and can be either `S` or `D`.

d
Is the number of the SIMD and FP destination register.

n
Is the number of the first SIMD and FP source register.

m
Is the number of the second SIMD and FP source register.

Architectures supported (scalar)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point Compare Greater than or Equal (vector). This instruction reads each floating-point value in the first source SIMD and FP register and if the value is greater than or equal to the corresponding floating-point value in the second source SIMD and FP register sets every bit of the corresponding vector element in the destination SIMD and FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD and FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.24 FCMGE (scalar, zero)

Floating-point Compare Greater than or Equal to zero (vector).

Syntax

```
FCMGE Hd, Hn, #0.0 ; Scalar half precision

FCMGE Vd, Vn, #0.0 ; Scalar single-precision and double-precision
```

Where:

Hd
Is the 16-bit name of the SIMD and FP destination register.

Hn
Is the 16-bit name of the SIMD and FP source register.

V
Is a width specifier, and can be either `S` or `D`.

d
Is the number of the SIMD and FP destination register.

n
Is the number of the SIMD and FP source register.

Architectures supported (scalar)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point Compare Greater than or Equal to zero (vector). This instruction reads each floating-point value in the source SIMD and FP register and if the value is greater than or equal to zero sets every bit of the corresponding vector element in the destination SIMD and FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD and FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.25 FCMGT (scalar, register)

Floating-point Compare Greater than (vector).

Syntax

```
FCMGT Hd, Hn, Hm ; Scalar half precision

FCMGT Vd, Vn, Vm ; Scalar single-precision and double-precision
```

Where:

Hd
Is the 16-bit name of the SIMD and FP destination register.

Hn
Is the 16-bit name of the first SIMD and FP source register.

Hm
Is the 16-bit name of the second SIMD and FP source register.

V
Is a width specifier, and can be either `S` or `D`.

d
Is the number of the SIMD and FP destination register.

n
Is the number of the first SIMD and FP source register.

m
Is the number of the second SIMD and FP source register.

Architectures supported (scalar)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point Compare Greater than (vector). This instruction reads each floating-point value in the first source SIMD and FP register and if the value is greater than the corresponding floating-point value in the second source SIMD and FP register sets every bit of the corresponding vector element in the destination SIMD and FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD and FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.26 FCMGT (scalar, zero)

Floating-point Compare Greater than zero (vector).

Syntax

```
FCMGT Hd, Hn, #0.0 ; Scalar half precision

FCMGT Vd, Vn, #0.0 ; Scalar single-precision and double-precision
```

Where:

Hd
Is the 16-bit name of the SIMD and FP destination register.

Hn
Is the 16-bit name of the SIMD and FP source register.

V
Is a width specifier, and can be either `S` or `D`.

d
Is the number of the SIMD and FP destination register.

n
Is the number of the SIMD and FP source register.

Architectures supported (scalar)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point Compare Greater than zero (vector). This instruction reads each floating-point value in the source SIMD and FP register and if the value is greater than zero sets every bit of the corresponding vector element in the destination SIMD and FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD and FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.27 FCMLA (scalar, by element)

Floating-point Complex Multiply Accumulate (by element).

Syntax

```
FCMLA Vd.T, Vn.T, Vm.Ts[index], #rotate ;
```

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of the values shown in Usage.

Vn
Is the name of the first SIMD and FP source register.

Vm
Is the name of the second SIMD and FP source register in the range 0 to 31.

Ts
Is an element size specifier, and can be either `H` or `S`.

index
Is the element index, in the range shown in Usage.

rotate
Is the rotation, and can be one of the values shown in Usage.

Architectures supported (scalar)

Supported in the Armv8.3-A architecture and later.

Usage

This instruction multiplies the two source complex numbers from the *Vm* and the *Vn* vector registers and adds the result to the corresponding complex number in the destination *Vd* vector register. The number of complex numbers that can be stored in the *Vm*, the *Vn*, and the *Vd* registers is calculated as the vector register size divided by the length of each complex number. These lengths are 16 for half-precision, 32 for single-precision, and 64 for double-precision. Each complex number is represented in a SIMP&FP register as a pair of elements with the imaginary part of the number being placed in the more significant element, and the real part of the number being placed in the less significant element. Both real and imaginary parts of the source and the resulting complex number are represented as floating-point values.

None, one, or both of the two vector elements that are read from each of the numbers in the *Vm* source SIMD and FP register can be negated based on the rotation value:

- If the rotation is 0, none of the vector elements are negated.
- If the rotation is 90, the odd-numbered vector elements are negated.
- If the rotation is 180, both vector elements are negated.
- If the rotation is 270, the even-numbered vector elements are negated.

The indexed element variant of this instruction is available for half-precision and single-precision number values. For this variant, the index value determines the position in the *Vm* source vector register of the single source value that is used to multiply each of the complex numbers in the *Vn* source vector register. The index value is encoded as H:L for half-precision values, or H for single-precision values.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 19-3 FCMLA (Scalar) specifier combinations

T	*Ts*
4H	H
8H	H
4S	S

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.28 FCMLE (scalar, zero)

Floating-point Compare Less than or Equal to zero (vector).

Syntax

```
FCMLE Hd, Hn, #0.0 ; Scalar half precision

FCMLE Vd, Vn, #0.0 ; Scalar single-precision and double-precision
```

Where:

Hd
Is the 16-bit name of the SIMD and FP destination register.

Hn
Is the 16-bit name of the SIMD and FP source register.

V
Is a width specifier, and can be either `S` or `D`.

d
Is the number of the SIMD and FP destination register.

n
Is the number of the SIMD and FP source register.

Architectures supported (scalar)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point Compare Less than or Equal to zero (vector). This instruction reads each floating-point value in the source SIMD and FP register and if the value is less than or equal to zero sets every bit of the corresponding vector element in the destination SIMD and FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD and FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.29 FCMLT (scalar, zero)

Floating-point Compare Less than zero (vector).

Syntax

```
FCMLT Hd, Hn, #0.0 ; Scalar half precision

FCMLT Vd, Vn, #0.0 ; Scalar single-precision and double-precision
```

Where:

Hd
Is the 16-bit name of the SIMD and FP destination register.

Hn
Is the 16-bit name of the SIMD and FP source register.

V
Is a width specifier, and can be either `S` or `D`.

d
Is the number of the SIMD and FP destination register.

n
Is the number of the SIMD and FP source register.

Architectures supported (scalar)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point Compare Less than zero (vector). This instruction reads each floating-point value in the source SIMD and FP register and if the value is less than zero sets every bit of the corresponding vector element in the destination SIMD and FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD and FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.30 FCVTAS (scalar)

Floating-point Convert to Signed integer, rounding to nearest with ties to Away (vector).

Syntax

```
FCVTAS Hd, Hn ; Scalar half precision

FCVTAS Vd, Vn ; Scalar single-precision and double-precision
```

Where:

Hd
Is the 16-bit name of the SIMD and FP destination register.

Hn
Is the 16-bit name of the SIMD and FP source register.

V
Is a width specifier, and can be either `S` or `D`.

d
Is the number of the SIMD and FP destination register.

n
Is the number of the SIMD and FP source register.

Architectures supported (scalar)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point Convert to Signed integer, rounding to nearest with ties to Away (vector). This instruction converts each element in a vector from a floating-point value to a signed integer value using the Round to Nearest with Ties to Away rounding mode and writes the result to the SIMD and FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.31 FCVTAU (scalar)

Floating-point Convert to Unsigned integer, rounding to nearest with ties to Away (vector).

Syntax

```
FCVTAU Hd, Hn ; Scalar half precision

FCVTAU Vd, Vn ; Scalar single-precision and double-precision
```

Where:

Hd
Is the 16-bit name of the SIMD and FP destination register.

Hn
Is the 16-bit name of the SIMD and FP source register.

V
Is a width specifier, and can be either `S` or `D`.

d
Is the number of the SIMD and FP destination register.

n
Is the number of the SIMD and FP source register.

Architectures supported (scalar)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point Convert to Unsigned integer, rounding to nearest with ties to Away (vector). This instruction converts each element in a vector from a floating-point value to an unsigned integer value using the Round to Nearest with Ties to Away rounding mode and writes the result to the SIMD and FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.32 FCVTMS (scalar)

Floating-point Convert to Signed integer, rounding toward Minus infinity (vector).

Syntax

```
FCVTMS Hd, Hn ; Scalar half precision
```

```
FCVTMS Vd, Vn ; Scalar single-precision and double-precision
```

Where:

Hd
Is the 16-bit name of the SIMD and FP destination register.

Hn
Is the 16-bit name of the SIMD and FP source register.

V
Is a width specifier, and can be either `S` or `D`.

d
Is the number of the SIMD and FP destination register.

n
Is the number of the SIMD and FP source register.

Architectures supported (scalar)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point Convert to Signed integer, rounding toward Minus infinity (vector). This instruction converts a scalar or each element in a vector from a floating-point value to a signed integer value using the Round towards Minus Infinity rounding mode, and writes the result to the SIMD and FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.33 FCVTMU (scalar)

Floating-point Convert to Unsigned integer, rounding toward Minus infinity (vector).

Syntax

```
FCVTMU Hd, Hn ; Scalar half precision

FCVTMU Vd, Vn ; Scalar single-precision and double-precision
```

Where:

Hd
Is the 16-bit name of the SIMD and FP destination register.

Hn
Is the 16-bit name of the SIMD and FP source register.

V
Is a width specifier, and can be either `S` or `D`.

d
Is the number of the SIMD and FP destination register.

n
Is the number of the SIMD and FP source register.

Architectures supported (scalar)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point Convert to Unsigned integer, rounding toward Minus infinity (vector). This instruction converts a scalar or each element in a vector from a floating-point value to an unsigned integer value using the Round towards Minus Infinity rounding mode, and writes the result to the SIMD and FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.34 FCVTNS (scalar)

Floating-point Convert to Signed integer, rounding to nearest with ties to even (vector).

Syntax

```
FCVTNS Hd, Hn ; Scalar half precision

FCVTNS Vd, Vn ; Scalar single-precision and double-precision
```

Where:

Hd
Is the 16-bit name of the SIMD and FP destination register.

Hn
Is the 16-bit name of the SIMD and FP source register.

V
Is a width specifier, and can be either `S` or `D`.

d
Is the number of the SIMD and FP destination register.

n
Is the number of the SIMD and FP source register.

Architectures supported (scalar)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point Convert to Signed integer, rounding to nearest with ties to even (vector). This instruction converts a scalar or each element in a vector from a floating-point value to a signed integer value using the Round to Nearest rounding mode, and writes the result to the SIMD and FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.35 FCVTNU (scalar)

Floating-point Convert to Unsigned integer, rounding to nearest with ties to even (vector).

Syntax

```
FCVTNU Hd, Hn ; Scalar half precision

FCVTNU Vd, Vn ; Scalar single-precision and double-precision
```

Where:

Hd
Is the 16-bit name of the SIMD and FP destination register.

Hn
Is the 16-bit name of the SIMD and FP source register.

V
Is a width specifier, and can be either `S` or `D`.

d
Is the number of the SIMD and FP destination register.

n
Is the number of the SIMD and FP source register.

Architectures supported (scalar)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point Convert to Unsigned integer, rounding to nearest with ties to even (vector). This instruction converts a scalar or each element in a vector from a floating-point value to an unsigned integer value using the Round to Nearest rounding mode, and writes the result to the SIMD and FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.36 FCVTPS (scalar)

Floating-point Convert to Signed integer, rounding toward Plus infinity (vector).

Syntax

```
FCVTPS Hd, Hn ; Scalar half precision
```

```
FCVTPS Vd, Vn ; Scalar single-precision and double-precision
```

Where:

Hd
Is the 16-bit name of the SIMD and FP destination register.

Hn
Is the 16-bit name of the SIMD and FP source register.

V
Is a width specifier, and can be either `S` or `D`.

d
Is the number of the SIMD and FP destination register.

n
Is the number of the SIMD and FP source register.

Architectures supported (scalar)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point Convert to Signed integer, rounding toward Plus infinity (vector). This instruction converts a scalar or each element in a vector from a floating-point value to a signed integer value using the Round towards Plus Infinity rounding mode, and writes the result to the SIMD and FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.37 FCVTPU (scalar)

Floating-point Convert to Unsigned integer, rounding toward Plus infinity (vector).

Syntax

```
FCVTPU Hd, Hn ; Scalar half precision

FCVTPU Vd, Vn ; Scalar single-precision and double-precision
```

Where:

Hd
Is the 16-bit name of the SIMD and FP destination register.

Hn
Is the 16-bit name of the SIMD and FP source register.

V
Is a width specifier, and can be either `S` or `D`.

d
Is the number of the SIMD and FP destination register.

n
Is the number of the SIMD and FP source register.

Architectures supported (scalar)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point Convert to Unsigned integer, rounding toward Plus infinity (vector). This instruction converts a scalar or each element in a vector from a floating-point value to an unsigned integer value using the Round towards Plus Infinity rounding mode, and writes the result to the SIMD and FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.38 FCVTXN (scalar)

Floating-point Convert to lower precision Narrow, rounding to odd (vector).

Syntax

`FCVTXN` *`Vbd, Van`*

Where:

Vb
Is the destination width specifier, `S`.

d
Is the number of the SIMD and FP destination register.

Va
Is the source width specifier, `D`.

n
Is the number of the SIMD and FP source register.

Usage

Floating-point Convert to lower precision Narrow, rounding to odd (vector). This instruction reads each vector element in the source SIMD and FP register, narrows each value to half the precision of the source element using the Round to Odd rounding mode, writes the result to a vector, and writes the vector to the destination SIMD and FP register.

Note

This instruction uses the Round to Odd rounding mode which is not defined by the IEEE 754-2008 standard. This rounding mode ensures that if the result of the conversion is inexact the least significant bit of the mantissa is forced to 1. This rounding mode enables a floating-point value to be converted to a lower precision format via an intermediate precision format while avoiding double rounding errors. For example, a 64-bit floating-point value can be converted to a correctly rounded 16-bit floating-point value by first using this instruction to produce a 32-bit value and then using another instruction with the wanted rounding mode to convert the 32-bit value to the final 16-bit floating-point value.

The `FCVTXN` instruction writes the vector to the lower half of the destination register and clears the upper half, while the `FCVTXN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.39 FCVTZS (scalar, fixed-point)

Floating-point Convert to Signed fixed-point, rounding toward Zero (vector).

Syntax

```
FCVTZS Vd, Vn, #fbits
```

Where:

V
Is a width specifier, and can be one of the values shown in Usage.

d
Is the number of the SIMD and FP destination register.

n
Is the number of the first SIMD and FP source register.

fbits
Is the number of fractional bits, in the range 1 to the operand width.

Usage

Floating-point Convert to Signed fixed-point, rounding toward Zero (vector). This instruction converts a scalar or each element in a vector from floating-point to fixed-point signed integer using the Round towards Zero rounding mode, and writes the result to the SIMD and FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 19-4 FCVTZS (Scalar) specifier combinations

V	*fbits*
H	
S	1 to 32
D	1 to 64

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.40 FCVTZS (scalar, integer)

Floating-point Convert to Signed integer, rounding toward Zero (vector).

Syntax

```
FCVTZS Hd, Hn ; Scalar half precision

FCVTZS Vd, Vn ; Scalar single-precision and double-precision
```

Where:

Hd
Is the 16-bit name of the SIMD and FP destination register.

Hn
Is the 16-bit name of the SIMD and FP source register.

V
Is a width specifier, and can be either `S` or `D`.

d
Is the number of the SIMD and FP destination register.

n
Is the number of the SIMD and FP source register.

Architectures supported (scalar)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point Convert to Signed integer, rounding toward Zero (vector). This instruction converts a scalar or each element in a vector from a floating-point value to a signed integer value using the Round towards Zero rounding mode, and writes the result to the SIMD and FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.41 FCVTZU (scalar, fixed-point)

Floating-point Convert to Unsigned fixed-point, rounding toward Zero (vector).

Syntax

```
FCVTZU Vd, Vn, #fbits
```

Where:

V
Is a width specifier, and can be one of the values shown in Usage.

d
Is the number of the SIMD and FP destination register.

n
Is the number of the first SIMD and FP source register.

fbits
Is the number of fractional bits, in the range 1 to the operand width.

Usage

Floating-point Convert to Unsigned fixed-point, rounding toward Zero (vector). This instruction converts a scalar or each element in a vector from floating-point to fixed-point unsigned integer using the Round towards Zero rounding mode, and writes the result to the general-purpose destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 19-5 FCVTZU (Scalar) specifier combinations

V	*fbits*
H	
S	1 to 32
D	1 to 64

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.42 FCVTZU (scalar, integer)

Floating-point Convert to Unsigned integer, rounding toward Zero (vector).

Syntax

```
FCVTZU Hd, Hn ; Scalar half precision

FCVTZU Vd, Vn ; Scalar single-precision and double-precision
```

Where:

Hd
Is the 16-bit name of the SIMD and FP destination register.

Hn
Is the 16-bit name of the SIMD and FP source register.

V
Is a width specifier, and can be either `S` or `D`.

d
Is the number of the SIMD and FP destination register.

n
Is the number of the SIMD and FP source register.

Architectures supported (scalar)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point Convert to Unsigned integer, rounding toward Zero (vector). This instruction converts a scalar or each element in a vector from a floating-point value to an unsigned integer value using the Round towards Zero rounding mode, and writes the result to the SIMD and FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.43 FMAXNMP (scalar)

Floating-point Maximum Number of Pair of elements (scalar).

Syntax

```
FMAXNMP Vd, Vn.T ; Half-precision
```

```
FMAXNMP Vd, Vn.T ; Single-precision and double-precision
```

Where:

V

Is the destination width specifier:

Half-precision
Must be `H`.

Single-precision and double-precision
Can be one of `S` or `D`.

T

Is the source arrangement specifier:

Half-precision
Must be `2H`.

Single-precision and double-precision
Can be one of `2S` or `2D`.

d

Is the number of the SIMD and FP destination register.

Vn

Is the name of the SIMD and FP source register.

Architectures supported (scalar)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point Maximum Number of Pair of elements (scalar). This instruction compares two vector elements in the source SIMD and FP register and writes the largest of the floating-point values as a scalar to the destination SIMD and FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.44 FMAXP (scalar)

Floating-point Maximum of Pair of elements (scalar).

Syntax

```
FMAXP Vd, Vn.T ; Half-precision

FMAXP Vd, Vn.T ; Single-precision and double-precision
```

Where:

V

Is the destination width specifier:

Half-precision
Must be `H`.

Single-precision and double-precision
Can be one of `S` or `D`.

T

Is the source arrangement specifier:

Half-precision
Must be `2H`.

Single-precision and double-precision
Can be one of `2S` or `2D`.

d

Is the number of the SIMD and FP destination register.

Vn

Is the name of the SIMD and FP source register.

Architectures supported (scalar)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point Maximum of Pair of elements (scalar). This instruction compares two vector elements in the source SIMD and FP register and writes the largest of the floating-point values as a scalar to the destination SIMD and FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.45 FMINNMP (scalar)

Floating-point Minimum Number of Pair of elements (scalar).

Syntax

```
FMINNMP Vd, Vn.T ; Half-precision
```

```
FMINNMP Vd, Vn.T ; Single-precision and double-precision
```

Where:

V

Is the destination width specifier:

Half-precision
Must be `H`.

Single-precision and double-precision
Can be one of `S` or `D`.

T

Is the source arrangement specifier:

Half-precision
Must be `2H`.

Single-precision and double-precision
Can be one of `2S` or `2D`.

d

Is the number of the SIMD and FP destination register.

Vn

Is the name of the SIMD and FP source register.

Architectures supported (scalar)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point Minimum Number of Pair of elements (scalar). This instruction compares two vector elements in the source SIMD and FP register and writes the smallest of the floating-point values as a scalar to the destination SIMD and FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.46 FMINP (scalar)

Floating-point Minimum of Pair of elements (scalar).

Syntax

```
FMINP Vd, Vn.T ; Half-precision

FMINP Vd, Vn.T ; Single-precision and double-precision
```

Where:

V

Is the destination width specifier:

Half-precision
Must be `H`.

Single-precision and double-precision
Can be one of `S` or `D`.

T

Is the source arrangement specifier:

Half-precision
Must be `2H`.

Single-precision and double-precision
Can be one of `2S` or `2D`.

d

Is the number of the SIMD and FP destination register.

Vn

Is the name of the SIMD and FP source register.

Architectures supported (scalar)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point Minimum of Pair of elements (scalar). This instruction compares two vector elements in the source SIMD and FP register and writes the smallest of the floating-point values as a scalar to the destination SIMD and FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.47 FMLA (scalar, by element)

Floating-point fused Multiply-Add to accumulator (by element).

Syntax

```
FMLA Vd, Vn, Vm.Ts[index]
```

Where:

V
Is a width specifier, and can be H, S, or D.

d
Is the number of the SIMD and FP destination register.

n
Is the number of the first SIMD and FP source register.

Ts
Is an element size specifier, and can be H, S, or D.

index
Is the element index, H.

Vm
Is the name of the second SIMD and FP source register in the range 0 to 31.

Architectures supported (scalar)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point fused Multiply-Add to accumulator (by element). 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 results in the vector elements of the destination SIMD and FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 19-6 FMLA (Scalar, single-precision and double-precision) specifier combinations

V	*Ts*	*index*
S	S	0 to 3
D	D	0 or 1

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.48 FMLS (scalar, by element)

Floating-point fused Multiply-Subtract from accumulator (by element).

Syntax

```
FMLS Vd, Vn, Vm.Ts[index]
```

Where:

V
Is a width specifier, and can be H, S, or D.

d
Is the number of the SIMD and FP destination register.

n
Is the number of the first SIMD and FP source register.

Ts
Is an element size specifier, and can be H, S, or D.

index
Is the element index, H.

Vm
Is the name of the second SIMD and FP source register in the range 0 to 31.

Architectures supported (scalar)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point fused Multiply-Subtract from accumulator (by element). 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 subtracts the results from the vector elements of the destination SIMD and FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 19-7 FMLS (Scalar, single-precision and double-precision) specifier combinations

V	*Ts*	*index*
S	S	0 to 3
D	D	0 or 1

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.49 FMUL (scalar, by element)

Floating-point Multiply (by element).

Syntax

```
FMUL Vd, Vn, Vm.Ts[index]
```

Where:

V
Is a width specifier, and can be H, S, or D.

d
Is the number of the SIMD and FP destination register.

n
Is the number of the first SIMD and FP source register.

Ts
Is an element size specifier, and can be H, S, or D.

index
Is the element index, H.

Vm
Is the name of the second SIMD and FP source register in the range 0 to 31.

Architectures supported (scalar)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point Multiply (by element). 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, places the results in a vector, and writes the vector to the destination SIMD and FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 19-8 FMUL (Scalar, single-precision and double-precision) specifier combinations

V	*Ts*	*index*
S	S	0 to 3
D	D	0 or 1

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.50 FMULX (scalar, by element)

Floating-point Multiply extended (by element).

Syntax

```
FMULX Vd, Vn, Vm.Ts[index]
```

Where:

V
Is a width specifier, and can be H, S, or D.

d
Is the number of the SIMD and FP destination register.

n
Is the number of the first SIMD and FP source register.

Ts
Is an element size specifier, and can be H, S, or D.

index
Is the element index, H.

Vm
Is the name of the second SIMD and FP source register in the range 0 to 31.

Architectures supported (scalar)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point Multiply extended (by element). This instruction multiplies the floating-point values in the vector elements in the first source SIMD and FP register by the specified floating-point value in the second source SIMD and FP register, places the results in a vector, and writes the vector to the destination SIMD and FP register.

Before each multiplication, a check is performed for whether one value is infinite and the other is zero. In this case, if only one of the values is negative, the result is 2.0, otherwise the result is -2.0.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 19-9 FMULX (Scalar, single-precision and double-precision) specifier combinations

V	*Ts*	*index*
S	S	0 to 3
D	D	0 or 1

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.51 FMULX (scalar)

Floating-point Multiply extended.

Syntax

```
FMULX Hd, Hn, Hm ; Scalar half precision

FMULX Vd, Vn, Vm ; Scalar single-precision and double-precision
```

Where:

Hd
Is the 16-bit name of the SIMD and FP destination register.

Hn
Is the 16-bit name of the first SIMD and FP source register.

Hm
Is the 16-bit name of the second SIMD and FP source register.

V
Is a width specifier, and can be either `S` or `D`.

d
Is the number of the SIMD and FP destination register.

n
Is the number of the first SIMD and FP source register.

m
Is the number of the second SIMD and FP source register.

Architectures supported (scalar)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point Multiply extended. This instruction multiplies corresponding floating-point values in the vectors of the two source SIMD and FP registers, places the resulting floating-point values in a vector, and writes the vector to the destination SIMD and FP register.

If one value is zero and the other value is infinite, the result is 2.0. In this case, the result is negative if only one of the values is negative, otherwise the result is positive.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.52 FRECPE (scalar)

Floating-point Reciprocal Estimate.

Syntax

```
FRECPE Hd, Hn ; Scalar half precision

FRECPE Vd, Vn ; Scalar single-precision and double-precision
```

Where:

Hd
Is the 16-bit name of the SIMD and FP destination register.

Hn
Is the 16-bit name of the SIMD and FP source register.

V
Is a width specifier, and can be either `S` or `D`.

d
Is the number of the SIMD and FP destination register.

n
Is the number of the SIMD and FP source register.

Architectures supported (scalar)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point Reciprocal Estimate. This instruction finds an approximate reciprocal estimate for each vector element in the source SIMD and FP register, places the result in a vector, and writes the vector to the destination SIMD and FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.53 FRECPS (scalar)

Floating-point Reciprocal Step.

Syntax

```
FRECPS Hd, Hn, Hm ; Scalar half precision

FRECPS Vd, Vn, Vm ; Scalar single-precision and double-precision
```

Where:

Hd
Is the 16-bit name of the SIMD and FP destination register.

Hn
Is the 16-bit name of the first SIMD and FP source register.

Hm
Is the 16-bit name of the second SIMD and FP source register.

V
Is a width specifier, and can be either `S` or `D`.

d
Is the number of the SIMD and FP destination register.

n
Is the number of the first SIMD and FP source register.

m
Is the number of the second SIMD and FP source register.

Architectures supported (scalar)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point Reciprocal Step. This instruction multiplies the corresponding floating-point values in the vectors of the two source SIMD and FP registers, subtracts each of the products from 2.0, places the resulting floating-point values in a vector, and writes the vector to the destination SIMD and FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.54 FRSQRTE (scalar)

Floating-point Reciprocal Square Root Estimate.

Syntax

```
FRSQRTE Hd, Hn ; Scalar half precision

FRSQRTE Vd, Vn ; Scalar single-precision and double-precision
```

Where:

Hd
Is the 16-bit name of the SIMD and FP destination register.

Hn
Is the 16-bit name of the SIMD and FP source register.

V
Is a width specifier, and can be either `S` or `D`.

d
Is the number of the SIMD and FP destination register.

n
Is the number of the SIMD and FP source register.

Architectures supported (scalar)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point Reciprocal Square Root Estimate. This instruction calculates an approximate square root for each vector element in the source SIMD and FP register, places the result in a vector, and writes the vector to the destination SIMD and FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.55 FRSQRTS (scalar)

Floating-point Reciprocal Square Root Step.

Syntax

```
FRSQRTS Hd, Hn, Hm ; Scalar half precision

FRSQRTS Vd, Vn, Vm ; Scalar single-precision and double-precision
```

Where:

Hd
Is the 16-bit name of the SIMD and FP destination register.

Hn
Is the 16-bit name of the first SIMD and FP source register.

Hm
Is the 16-bit name of the second SIMD and FP source register.

V
Is a width specifier, and can be either `S` or `D`.

d
Is the number of the SIMD and FP destination register.

n
Is the number of the first SIMD and FP source register.

m
Is the number of the second SIMD and FP source register.

Architectures supported (scalar)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point Reciprocal Square Root Step. This instruction multiplies corresponding floating-point values in the vectors of the two source SIMD and FP registers, subtracts each of the products from 3.0, divides these results by 2.0, places the results into a vector, and writes the vector to the destination SIMD and FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.56 MOV (scalar)

Move vector element to scalar.

This instruction is an alias of DUP (element).

The equivalent instruction is `DUP Vd, Vn.T[index]`.

Syntax

`MOV Vd, Vn.T[index]`

Where:

V
Is the destination width specifier, and can be one of the values shown in Usage.

d
Is the number of the SIMD and FP destination register.

Vn
Is the name of the SIMD and FP source register.

T
Is the element width specifier, and can be one of the values shown in Usage.

index
Is the element index, in the range shown in Usage.

Usage

Move vector element to scalar. This instruction duplicates the specified vector element in the SIMD and FP source register into a scalar, and writes the result to the SIMD and FP destination register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 19-10 MOV (Scalar) specifier combinations

V	*T*	*index*
B	B	0 to 15
H	H	0 to 7
S	S	0 to 3
D	D	0 or 1

Related references

19.16 DUP (scalar, element) on page 19-1234.
19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.57 NEG (scalar)

Negate (vector).

Syntax

`NEG Vd, Vn`

Where:

V
Is a width specifier, `D`.

d
Is the number of the SIMD and FP destination register.

n
Is the number of the SIMD and FP source register.

Usage

Negate (vector). This instruction reads each vector element from the source SIMD and FP register, negates each value, puts the result into a vector, and writes the vector to the destination SIMD and FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.58 SCVTF (scalar, fixed-point)

Signed fixed-point Convert to Floating-point (vector).

Syntax

```
SCVTF Vd, Vn, #fbits
```

Where:

V
Is a width specifier, and can be one of the values shown in Usage.

d
Is the number of the SIMD and FP destination register.

n
Is the number of the first SIMD and FP source register.

fbits
Is the number of fractional bits, in the range 1 to the operand width.

Usage

Signed fixed-point Convert to Floating-point (vector). This instruction converts each element in a vector from fixed-point to floating-point using the rounding mode that is specified by the FPCR, and writes the result to the SIMD and FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 19-11 SCVTF (Scalar) specifier combinations

V	*fbits*
H	
S	1 to 32
D	1 to 64

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.59 SCVTF (scalar, integer)

Signed integer Convert to Floating-point (vector).

Syntax

```
SCVTF Hd, Hn ; Scalar half precision

SCVTF Vd, Vn ; Scalar single-precision and double-precision
```

Where:

Hd
Is the 16-bit name of the SIMD and FP destination register.

Hn
Is the 16-bit name of the SIMD and FP source register.

V
Is a width specifier, and can be either `S` or `D`.

d
Is the number of the SIMD and FP destination register.

n
Is the number of the SIMD and FP source register.

Architectures supported (scalar)

Supported in the Armv8.2 architecture and later.

Usage

Signed integer Convert to Floating-point (vector). This instruction converts each element in a vector from signed integer to floating-point using the rounding mode that is specified by the FPCR, and writes the result to the SIMD and FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.60 SHL (scalar)

Shift Left (immediate).

Syntax

```
SHL Vd, Vn, #shift
```

Where:

V
: Is a width specifier, `D`.

d
: Is the number of the SIMD and FP destination register.

n
: Is the number of the first SIMD and FP source register.

shift
: Is the left shift amount, in the range 0 to 63.

Usage

Shift Left (immediate). This instruction reads each value from a vector, right shifts each result by an immediate value, writes the final result to a vector, and writes the vector to the destination SIMD and FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.61 SLI (scalar)

Shift Left and Insert (immediate).

Syntax

```
SLI Vd, Vn, #shift
```

Where:

V
: Is a width specifier, `D`.

d
: Is the number of the SIMD and FP destination register.

n
: Is the number of the first SIMD and FP source register.

shift
: Is the left shift amount, in the range 0 to 63.

Usage

Shift Left and Insert (immediate). This instruction reads each vector element in the source SIMD and FP register, left shifts each vector element by an immediate value, and inserts the result into the corresponding vector element in the destination SIMD and FP register such that the new zero bits created by the shift are not inserted but retain their existing value. Bits shifted out of the left of each vector element in the source register are lost.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.62 SQABS (scalar)

Signed saturating Absolute value.

Syntax

`SQABS Vd, Vn`

Where:

V
Is a width specifier, and can be one of `B`, `H`, `S` or `D`.

d
Is the number of the SIMD and FP destination register.

n
Is the number of the SIMD and FP source register.

Usage

Signed saturating Absolute value. This instruction reads each vector element from the source SIMD and FP register, puts the absolute value of the result into a vector, and writes the vector to the destination SIMD and FP register. All the values in this instruction are signed integer values.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.63 SQADD (scalar)

Signed saturating Add.

Syntax

`SQADD Vd, Vn, Vm`

Where:

V
Is a width specifier, and can be one of `B`, `H`, `S` or `D`.

d
Is the number of the SIMD and FP destination register.

n
Is the number of the first SIMD and FP source register.

m
Is the number of the second SIMD and FP source register.

Usage

Signed saturating Add. This instruction adds the values of corresponding elements of the two source SIMD and FP registers, places the results into a vector, and writes the vector to the destination SIMD and FP register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.64 SQDMLAL (scalar, by element)

Signed saturating Doubling Multiply-Add Long (by element).

Syntax

`SQDMLAL` *`Vad, Vbn, Vm.Ts`*`[`*`index`*`]`

Where:

Va
Is the destination width specifier, and can be either `S` or `D`.

d
Is the number of the SIMD and FP destination register.

Vb
Is the source width specifier, and can be either `H` or `S`.

n
Is the number of the first SIMD and FP source register.

Vm
Is the name of the second SIMD and FP source register:
- If *`Ts`* is `H`, then *`Vm`* must be in the range V0 to V15.
- If *`Ts`* is `S`, then *`Vm`* must be in the range V0 to V31.

Ts
Is an element size specifier, and can be either `H` or `S`.

index
Is the element index, in the range shown in Usage.

Usage

Signed saturating Doubling Multiply-Add Long (by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD and FP register by the specified vector element of the second source SIMD and FP register, doubles the results, and accumulates the final results with the vector elements of the destination SIMD and FP register. The destination vector elements are twice as long as the elements that are multiplied.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

The `SQDMLAL` instruction extracts vector elements from the lower half of the first source register, while the `SQDMLAL2` instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 19-12 SQDMLAL (Scalar) specifier combinations

Va	*Vb*	*Ts*	*index*
S	H	H	0 to 7
D	S	S	0 to 3

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.65 SQDMLAL (scalar)

Signed saturating Doubling Multiply-Add Long.

Syntax

`SQDMLAL` *`Vad, Vbn, Vbm`*

Where:

Va
Is the destination width specifier, and can be either `S` or `D`.

d
Is the number of the SIMD and FP destination register.

Vb
Is the source width specifier, and can be either `H` or `S`.

n
Is the number of the first SIMD and FP source register.

m
Is the number of the second SIMD and FP source register.

Usage

Signed saturating Doubling Multiply-Add Long. This instruction multiplies corresponding signed integer values in the lower or upper half of the vectors of the two source SIMD and FP registers, doubles the results, and accumulates the final results with the vector elements of the destination SIMD and FP register. The destination vector elements are twice as long as the elements that are multiplied.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

The `SQDMLAL` instruction extracts each source vector from the lower half of each source register, while the `SQDMLAL2` instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 19-13 SQDMLAL (Scalar) specifier combinations

Va	*Vb*
S	H
D	S

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.66 SQDMLSL (scalar, by element)

Signed saturating Doubling Multiply-Subtract Long (by element).

Syntax

`SQDMLSL` *Vad, Vbn, Vm.Ts*[*index*]

Where:

Va
Is the destination width specifier, and can be either `S` or `D`.

d
Is the number of the SIMD and FP destination register.

Vb
Is the source width specifier, and can be either `H` or `S`.

n
Is the number of the first SIMD and FP source register.

Vm
Is the name of the second SIMD and FP source register:
- If *Ts* is `H`, then *Vm* must be in the range V0 to V15.
- If *Ts* is `S`, then *Vm* must be in the range V0 to V31.

Ts
Is an element size specifier, and can be either `H` or `S`.

index
Is the element index, in the range shown in Usage.

Usage

Signed saturating Doubling Multiply-Subtract Long (by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD and FP register by the specified vector element of the second source SIMD and FP register, doubles the results, and subtracts the final results from the vector elements of the destination SIMD and FP register. The destination vector elements are twice as long as the elements that are multiplied. All the values in this instruction are signed integer values.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

The `SQDMLSL` instruction extracts vector elements from the lower half of the first source register, while the `SQDMLSL2` instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 19-14 SQDMLSL (Scalar) specifier combinations

Va	*Vb*	*Ts*	*index*
S	H	H	0 to 7
D	S	S	0 to 3

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.67 SQDMLSL (scalar)

Signed saturating Doubling Multiply-Subtract Long.

Syntax

`SQDMLSL` *`Vad, Vbn, Vbm`*

Where:

Va
Is the destination width specifier, and can be either `S` or `D`.

d
Is the number of the SIMD and FP destination register.

Vb
Is the source width specifier, and can be either `H` or `S`.

n
Is the number of the first SIMD and FP source register.

m
Is the number of the second SIMD and FP source register.

Usage

Signed saturating Doubling Multiply-Subtract Long. This instruction multiplies corresponding signed integer values in the lower or upper half of the vectors of the two source SIMD and FP registers, doubles the results, and subtracts the final results from the vector elements of the destination SIMD and FP register. The destination vector elements are twice as long as the elements that are multiplied.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

The `SQDMLSL` instruction extracts each source vector from the lower half of each source register, while the `SQDMLSL2` instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 19-15 SQDMLSL (Scalar) specifier combinations

Va	*Vb*
S	H
D	S

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.68 SQDMULH (scalar, by element)

Signed saturating Doubling Multiply returning High half (by element).

Syntax

`SQDMULH Vd, Vn, Vm.Ts[index]`

Where:

V
Is a width specifier, and can be either `H` or `S`.

d
Is the number of the SIMD and FP destination register.

n
Is the number of the first SIMD and FP source register.

Vm
Is the name of the second SIMD and FP source register:
- If *Ts* is `H`, then *Vm* must be in the range V0 to V15.
- If *Ts* is `S`, then *Vm* must be in the range V0 to V31.

Ts
Is an element size specifier, and can be either `H` or `S`.

index
Is the element index, in the range shown in Usage.

Usage

Signed saturating Doubling Multiply returning High half (by element). This instruction multiplies each vector element in the first source SIMD and FP register by the specified vector element of the second source SIMD and FP register, doubles the results, places the most significant half of the final results into a vector, and writes the vector to the destination SIMD and FP register.

The results are truncated. For rounded results, see *19.77 SQRDMULH (scalar, by element)* on page 19-1296.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 19-16 SQDMULH (Scalar) specifier combinations

V	*Ts*	*index*
H	H	0 to 7
S	S	0 to 3

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.69 SQDMULH (scalar)

Signed saturating Doubling Multiply returning High half.

Syntax

`SQDMULH` *`Vd, Vn, Vm`*

Where:

V
Is a width specifier, and can be either `H` or `S`.

d
Is the number of the SIMD and FP destination register.

n
Is the number of the first SIMD and FP source register.

m
Is the number of the second SIMD and FP source register.

Usage

Signed saturating Doubling Multiply returning High half. This instruction multiplies the values of corresponding elements of the two source SIMD and FP registers, doubles the results, places the most significant half of the final results into a vector, and writes the vector to the destination SIMD and FP register.

The results are truncated. For rounded results, see *19.78 SQRDMULH (scalar)* on page 19-1297.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.70 SQDMULL (scalar, by element)

Signed saturating Doubling Multiply Long (by element).

Syntax

`SQDMULL` *`Vad, Vbn, Vm.Ts`*`[`*`index`*`]`

Where:

Va
Is the destination width specifier, and can be either `S` or `D`.

d
Is the number of the SIMD and FP destination register.

Vb
Is the source width specifier, and can be either `H` or `S`.

n
Is the number of the first SIMD and FP source register.

Vm
Is the name of the second SIMD and FP source register:
- If *`Ts`* is `H`, then *`Vm`* must be in the range V0 to V15.
- If *`Ts`* is `S`, then *`Vm`* must be in the range V0 to V31.

Ts
Is an element size specifier, and can be either `H` or `S`.

index
Is the element index, in the range shown in Usage.

Usage

Signed saturating Doubling Multiply Long (by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD and FP register by the specified vector element of the second source SIMD and FP register, doubles the results, places the final results in a vector, and writes the vector to the destination SIMD and FP register. All the values in this instruction are signed integer values.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

The `SQDMULL` instruction extracts the first source vector from the lower half of the first source register, while the `SQDMULL2` instruction extracts the first source vector from the upper half of the first source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 19-17 SQDMULL (Scalar) specifier combinations

Va	*Vb*	*Ts*	*index*
S	H	H	0 to 7
D	S	S	0 to 3

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.71 SQDMULL (scalar)

Signed saturating Doubling Multiply Long.

Syntax

`SQDMULL` *`Vad, Vbn, Vbm`*

Where:

Va
Is the destination width specifier, and can be either `S` or `D`.

d
Is the number of the SIMD and FP destination register.

Vb
Is the source width specifier, and can be either `H` or `S`.

n
Is the number of the first SIMD and FP source register.

m
Is the number of the second SIMD and FP source register.

Usage

Signed saturating Doubling Multiply Long. This instruction multiplies corresponding vector elements in the lower or upper half of the two source SIMD and FP registers, doubles the results, places the final results in a vector, and writes the vector to the destination SIMD and FP register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

The `SQDMULL` instruction extracts each source vector from the lower half of each source register, while the `SQDMULL2` instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 19-18 SQDMULL (Scalar) specifier combinations

Va	*Vb*
S	H
D	S

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.72 SQNEG (scalar)

Signed saturating Negate.

Syntax

`SQNEG Vd, Vn`

Where:

V
Is a width specifier, and can be one of `B`, `H`, `S` or `D`.

d
Is the number of the SIMD and FP destination register.

n
Is the number of the SIMD and FP source register.

Usage

Signed saturating Negate. This instruction reads each vector element from the source SIMD and FP register, negates each value, places the result into a vector, and writes the vector to the destination SIMD and FP register. All the values in this instruction are signed integer values.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.73 SQRDMLAH (scalar, by element)

Signed Saturating Rounding Doubling Multiply Accumulate returning High Half (by element).

Syntax

```
SQRDMLAH Vd, Vn, Vm.Ts[index]
```

Where:

V
Is a width specifier, and can be either `H` or `S`.

d
Is the number of the SIMD and FP destination register.

n
Is the number of the first SIMD and FP source register.

Vm
Is the name of the second SIMD and FP source register:
- If *Ts* is `H`, then *Vm* must be in the range V0 to V15.
- If *Ts* is `S`, then *Vm* must be in the range V0 to V31.

Ts
Is an element size specifier, and can be either `H` or `S`.

index
Is the element index, in the range shown in Usage.

Architectures supported (scalar)

Supported in the Armv8.1 architecture and later.

Usage

Signed Saturating Rounding Doubling Multiply Accumulate returning High Half (by element). This instruction multiplies the vector elements of the first source SIMD and FP register with the value of a vector element of the second source SIMD and FP register without saturating the multiply results, doubles the results, and accumulates the most significant half of the final results with the vector elements of the destination SIMD and FP register. The results are rounded.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSR.QC, is set if saturation occurs.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 19-19 SQRDMLAH (Scalar) specifier combinations

V	*Ts*	*index*
H	H	0 to 7
S	S	0 to 3

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.74 SQRDMLAH (scalar)

Signed Saturating Rounding Doubling Multiply Accumulate returning High Half (vector).

Syntax

```
SQRDMLAH Vd, Vn, Vm
```

Where:

V
Is a width specifier, and can be either `H` or `S`.

d
Is the number of the SIMD and FP destination register.

n
Is the number of the first SIMD and FP source register.

m
Is the number of the second SIMD and FP source register.

Architectures supported (scalar)

Supported in the Armv8.1 architecture and later.

Usage

Signed Saturating Rounding Doubling Multiply Accumulate returning High Half (vector). This instruction multiplies the vector elements of the first source SIMD and FP register with the corresponding vector elements of the second source SIMD and FP register without saturating the multiply results, doubles the results, and accumulates the most significant half of the final results with the vector elements of the destination SIMD and FP register. The results are rounded.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSR.QC, is set if saturation occurs.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.75 SQRDMLSH (scalar, by element)

Signed Saturating Rounding Doubling Multiply Subtract returning High Half (by element).

Syntax

```
SQRDMLSH Vd, Vn, Vm.Ts[index]
```

Where:

V
Is a width specifier, and can be either `H` or `S`.

d
Is the number of the SIMD and FP destination register.

n
Is the number of the first SIMD and FP source register.

Vm
Is the name of the second SIMD and FP source register:
- If *Ts* is `H`, then *Vm* must be in the range V0 to V15.
- If *Ts* is `S`, then *Vm* must be in the range V0 to V31.

Ts
Is an element size specifier, and can be either `H` or `S`.

index
Is the element index, in the range shown in Usage.

Architectures supported (scalar)

Supported in the Armv8.1 architecture and later.

Usage

Signed Saturating Rounding Doubling Multiply Subtract returning High Half (by element). This instruction multiplies the vector elements of the first source SIMD and FP register with the value of a vector element of the second source SIMD and FP register without saturating the multiply results, doubles the results, and subtracts the most significant half of the final results from the vector elements of the destination SIMD and FP register. The results are rounded.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSR.QC, is set if saturation occurs.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 19-20 SQRDMLSH (Scalar) specifier combinations

V	*Ts*	*index*
H	H	0 to 7
S	S	0 to 3

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.76 SQRDMLSH (scalar)

Signed Saturating Rounding Doubling Multiply Subtract returning High Half (vector).

Syntax

```
SQRDMLSH Vd, Vn, Vm
```

Where:

V
: Is a width specifier, and can be either `H` or `S`.

d
: Is the number of the SIMD and FP destination register.

n
: Is the number of the first SIMD and FP source register.

m
: Is the number of the second SIMD and FP source register.

Architectures supported (scalar)

Supported in the Armv8.1 architecture and later.

Usage

Signed Saturating Rounding Doubling Multiply Subtract returning High Half (vector). This instruction multiplies the vector elements of the first source SIMD and FP register with the corresponding vector elements of the second source SIMD and FP register without saturating the multiply results, doubles the results, and subtracts the most significant half of the final results from the vector elements of the destination SIMD and FP register. The results are rounded.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSR.QC, is set if saturation occurs.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.77 SQRDMULH (scalar, by element)

Signed saturating Rounding Doubling Multiply returning High half (by element).

Syntax

```
SQRDMULH Vd, Vn, Vm.Ts[index]
```

Where:

V
Is a width specifier, and can be either `H` or `S`.

d
Is the number of the SIMD and FP destination register.

n
Is the number of the first SIMD and FP source register.

Vm
Is the name of the second SIMD and FP source register:
- If *Ts* is `H`, then *Vm* must be in the range V0 to V15.
- If *Ts* is `S`, then *Vm* must be in the range V0 to V31.

Ts
Is an element size specifier, and can be either `H` or `S`.

index
Is the element index, in the range shown in Usage.

Usage

Signed saturating Rounding Doubling Multiply returning High half (by element). This instruction multiplies each vector element in the first source SIMD and FP register by the specified vector element of the second source SIMD and FP register, doubles the results, places the most significant half of the final results into a vector, and writes the vector to the destination SIMD and FP register.

The results are rounded. For truncated results, see *19.68 SQDMULH (scalar, by element)* on page 19-1287.

If any of the results overflows, they are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 19-21 SQRDMULH (Scalar) specifier combinations

V	*Ts*	*index*
H	H	0 to 7
S	S	0 to 3

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.78 SQRDMULH (scalar)

Signed saturating Rounding Doubling Multiply returning High half.

Syntax

```
SQRDMULH Vd, Vn, Vm
```

Where:

V
Is a width specifier, and can be either `H` or `S`.

d
Is the number of the SIMD and FP destination register.

n
Is the number of the first SIMD and FP source register.

m
Is the number of the second SIMD and FP source register.

Usage

Signed saturating Rounding Doubling Multiply returning High half. This instruction multiplies the values of corresponding elements of the two source SIMD and FP registers, doubles the results, places the most significant half of the final results into a vector, and writes the vector to the destination SIMD and FP register.

The results are rounded. For truncated results, see *19.69 SQDMULH (scalar)* on page 19-1288.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.79 SQRSHL (scalar)

Signed saturating Rounding Shift Left (register).

Syntax

`SQRSHL Vd, Vn, Vm`

Where:

V
Is a width specifier, and can be one of `B`, `H`, `S` or `D`.

d
Is the number of the SIMD and FP destination register.

n
Is the number of the first SIMD and FP source register.

m
Is the number of the second SIMD and FP source register.

Usage

Signed saturating Rounding Shift Left (register). This instruction takes each vector element in the first source SIMD and FP register, shifts it by a value from the least significant byte of the corresponding vector element of the second source SIMD and FP register, places the results into a vector, and writes the vector to the destination SIMD and FP register.

If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift. The results are rounded. For truncated results, see *19.83 SQSHL (scalar, register)* on page 19-1302.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.80 SQRSHRN (scalar)

Signed saturating Rounded Shift Right Narrow (immediate).

Syntax

`SQRSHRN Vbd, Van, #shift`

Where:

Vb
Is the destination width specifier, and can be one of the values shown in Usage.

d
Is the number of the SIMD and FP destination register.

Va
Is the source width specifier, and can be one of the values shown in Usage.

n
Is the number of the first SIMD and FP source register.

shift
Is the right shift amount, in the range 1 to the destination operand width in bits, and can be one of the values shown in Usage.

Usage

Signed saturating Rounded Shift Right Narrow (immediate). This instruction reads each vector element in the source SIMD and FP register, right shifts each result by an immediate value, saturates each shifted result to a value that is half the original width, puts the final result into a vector, and writes the vector to the lower or upper half of the destination SIMD and FP register. All the values in this instruction are signed integer values. The destination vector elements are half as long as the source vector elements. The results are rounded. For truncated results, see *19.85 SQSHRN (scalar)* on page 19-1304.

The `SQRSHRN` instruction writes the vector to the lower half of the destination register and clears the upper half, while the `SQRSHRN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 19-22 SQRSHRN (Scalar) specifier combinations

Vb	*Va*	*shift*
B	H	1 to 8
H	S	1 to 16
S	D	1 to 32

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.81 SQRSHRUN (scalar)

Signed saturating Rounded Shift Right Unsigned Narrow (immediate).

Syntax

```
SQRSHRUN Vbd, Van, #shift
```

Where:

Vb
Is the destination width specifier, and can be one of the values shown in Usage.

d
Is the number of the SIMD and FP destination register.

Va
Is the source width specifier, and can be one of the values shown in Usage.

n
Is the number of the first SIMD and FP source register.

shift
Is the right shift amount, in the range 1 to the destination operand width in bits, and can be one of the values shown in Usage.

Usage

Signed saturating Rounded Shift Right Unsigned Narrow (immediate). This instruction reads each signed integer value in the vector of the source SIMD and FP register, right shifts each value by an immediate value, saturates the result to an unsigned integer value that is half the original width, places the final result into a vector, and writes the vector to the destination SIMD and FP register. The results are rounded. For truncated results, see *19.86 SQSHRUN (scalar)* on page 19-1305.

The `SQRSHRUN` instruction writes the vector to the lower half of the destination register and clears the upper half, while the `SQRSHRUN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 19-23 SQRSHRUN (Scalar) specifier combinations

Vb	*Va*	*shift*
B	H	1 to 8
H	S	1 to 16
S	D	1 to 32

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.82 SQSHL (scalar, immediate)

Signed saturating Shift Left (immediate).

Syntax

```
SQSHL Vd, Vn, #shift
```

Where:

V
Is a width specifier, and can be one of the values shown in Usage.

d
Is the number of the SIMD and FP destination register.

n
Is the number of the first SIMD and FP source register.

shift
Is the left shift amount, in the range 0 to the operand width in bits minus 1, and can be one of the values shown in Usage.

Usage

Signed saturating Shift Left (immediate). This instruction reads each vector element in the source SIMD and FP register, shifts each result by an immediate value, places the final result in a vector, and writes the vector to the destination SIMD and FP register. The results are truncated. For rounded results, see *19.102 UQRSHL (scalar)* on page 19-1321.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 19-24 SQSHL (Scalar) specifier combinations

V	*shift*
B	0 to 7
H	0 to 15
S	0 to 31
D	0 to 63

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.83 SQSHL (scalar, register)

Signed saturating Shift Left (register).

Syntax

`SQSHL Vd, Vn, Vm`

Where:

V
Is a width specifier, and can be one of `B`, `H`, `S` or `D`.

d
Is the number of the SIMD and FP destination register.

n
Is the number of the first SIMD and FP source register.

m
Is the number of the second SIMD and FP source register.

Usage

Signed saturating Shift Left (register). This instruction takes each element in the vector of the first source SIMD and FP register, shifts each element by a value from the least significant byte of the corresponding element of the second source SIMD and FP register, places the results in a vector, and writes the vector to the destination SIMD and FP register.

If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift. The results are truncated. For rounded results, see *19.79 SQRSHL (scalar)* on page 19-1298.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.84 SQSHLU (scalar)

Signed saturating Shift Left Unsigned (immediate).

Syntax

```
SQSHLU Vd, Vn, #shift
```

Where:

V
Is a width specifier, and can be one of the values shown in Usage.

d
Is the number of the SIMD and FP destination register.

n
Is the number of the first SIMD and FP source register.

shift
Is the left shift amount, in the range 0 to the operand width in bits minus 1, and can be one of the values shown in Usage.

Usage

Signed saturating Shift Left Unsigned (immediate). This instruction reads each signed integer value in the vector of the source SIMD and FP register, shifts each value by an immediate value, saturates the shifted result to an unsigned integer value, places the result in a vector, and writes the vector to the destination SIMD and FP register. The results are truncated. For rounded results, see *19.102 UQRSHL (scalar)* on page 19-1321.

If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 19-25 SQSHLU (Scalar) specifier combinations

V	*shift*
B	0 to 7
H	0 to 15
S	0 to 31
D	0 to 63

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.85 SQSHRN (scalar)

Signed saturating Shift Right Narrow (immediate).

Syntax

`SQSHRN Vbd, Van, #shift`

Where:

Vb
Is the destination width specifier, and can be one of the values shown in Usage.

d
Is the number of the SIMD and FP destination register.

Va
Is the source width specifier, and can be one of the values shown in Usage.

n
Is the number of the first SIMD and FP source register.

shift
Is the right shift amount, in the range 1 to the destination operand width in bits, and can be one of the values shown in Usage.

Usage

Signed saturating Shift Right Narrow (immediate). This instruction reads each vector element in the source SIMD and FP register, right shifts and truncates each result by an immediate value, saturates each shifted result to a value that is half the original width, puts the final result into a vector, and writes the vector to the lower or upper half of the destination SIMD and FP register. All the values in this instruction are signed integer values. The destination vector elements are half as long as the source vector elements. For rounded results, see *19.80 SQRSHRN (scalar)* on page 19-1299.

The `SQSHRN` instruction writes the vector to the lower half of the destination register and clears the upper half, while the `SQSHRN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 19-26 SQSHRN (Scalar) specifier combinations

Vb	*Va*	*shift*
B	H	1 to 8
H	S	1 to 16
S	D	1 to 32

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.86 SQSHRUN (scalar)

Signed saturating Shift Right Unsigned Narrow (immediate).

Syntax

`SQSHRUN Vbd, Van, #shift`

Where:

Vb
Is the destination width specifier, and can be one of the values shown in Usage.

d
Is the number of the SIMD and FP destination register.

Va
Is the source width specifier, and can be one of the values shown in Usage.

n
Is the number of the first SIMD and FP source register.

shift
Is the right shift amount, in the range 1 to the destination operand width in bits, and can be one of the values shown in Usage.

Usage

Signed saturating Shift Right Unsigned Narrow (immediate). This instruction reads each signed integer value in the vector of the source SIMD and FP register, right shifts each value by an immediate value, saturates the result to an unsigned integer value that is half the original width, places the final result into a vector, and writes the vector to the destination SIMD and FP register. The results are truncated. For rounded results, see *19.81 SQRSHRUN (scalar)* on page 19-1300.

The `SQSHRUN` instruction writes the vector to the lower half of the destination register and clears the upper half, while the `SQSHRUN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 19-27 SQSHRUN (Scalar) specifier combinations

Vb	*Va*	*shift*
B	H	1 to 8
H	S	1 to 16
S	D	1 to 32

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.87 SQSUB (scalar)

Signed saturating Subtract.

Syntax

`SQSUB Vd, Vn, Vm`

Where:

V
Is a width specifier, and can be one of `B`, `H`, `S` or `D`.

d
Is the number of the SIMD and FP destination register.

n
Is the number of the first SIMD and FP source register.

m
Is the number of the second SIMD and FP source register.

Usage

Signed saturating Subtract. This instruction subtracts the element values of the second source SIMD and FP register from the corresponding element values of the first source SIMD and FP register, places the results into a vector, and writes the vector to the destination SIMD and FP register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.88 SQXTN (scalar)

Signed saturating extract Narrow.

Syntax

`SQXTN` *Vbd, Van*

Where:

Vb
Is the destination width specifier, and can be one of the values shown in Usage.

d
Is the number of the SIMD and FP destination register.

Va
Is the source width specifier, and can be one of the values shown in Usage.

n
Is the number of the SIMD and FP source register.

Usage

Signed saturating extract Narrow. This instruction reads each vector element from the source SIMD and FP register, saturates the value to half the original width, places the result into a vector, and writes the vector to the lower or upper half of the destination SIMD and FP register. The destination vector elements are half as long as the source vector elements. All the values in this instruction are signed integer values.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

The `SQXTN` instruction writes the vector to the lower half of the destination register and clears the upper half, while the `SQXTN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 19-28 SQXTN (Scalar) specifier combinations

Vb	*Va*
B	H
H	S
S	D

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.89 SQXTUN (scalar)

Signed saturating extract Unsigned Narrow.

Syntax

`SQXTUN` *Vbd, Van*

Where:

Vb
Is the destination width specifier, and can be one of the values shown in Usage.

d
Is the number of the SIMD and FP destination register.

Va
Is the source width specifier, and can be one of the values shown in Usage.

n
Is the number of the SIMD and FP source register.

Usage

Signed saturating extract Unsigned Narrow. This instruction reads each signed integer value in the vector of the source SIMD and FP register, saturates the value to an unsigned integer value that is half the original width, places the result into a vector, and writes the vector to the lower or upper half of the destination SIMD and FP register. The destination vector elements are half as long as the source vector elements.

If saturation occurs, the cumulative saturation bit FPSR.QC is set.

The `SQXTUN` instruction writes the vector to the lower half of the destination register and clears the upper half, while the `SQXTUN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 19-29 SQXTUN (Scalar) specifier combinations

Vb	*Va*
B	H
H	S
S	D

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.90 SRI (scalar)

Shift Right and Insert (immediate).

Syntax

```
SRI Vd, Vn, #shift
```

Where:

V
: Is a width specifier, `D`.

d
: Is the number of the SIMD and FP destination register.

n
: Is the number of the first SIMD and FP source register.

shift
: Is the right shift amount, in the range 1 to 64.

Usage

Shift Right and Insert (immediate). This instruction reads each vector element in the source SIMD and FP register, right shifts each vector element by an immediate value, and inserts the result into the corresponding vector element in the destination SIMD and FP register such that the new zero bits created by the shift are not inserted but retain their existing value. Bits shifted out of the right of each vector element of the source register are lost.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.91 SRSHL (scalar)

Signed Rounding Shift Left (register).

Syntax

`SRSHL Vd, Vn, Vm`

Where:

V
Is a width specifier, `D`.

d
Is the number of the SIMD and FP destination register.

n
Is the number of the first SIMD and FP source register.

m
Is the number of the second SIMD and FP source register.

Usage

Signed Rounding Shift Left (register). This instruction takes each signed integer value in the vector of the first source SIMD and FP register, shifts it by a value from the least significant byte of the corresponding element of the second source SIMD and FP register, places the results in a vector, and writes the vector to the destination SIMD and FP register.

If the shift value is positive, the operation is a left shift. If the shift value is negative, it is a rounding right shift. For a truncating shift, see *19.94 SSHL (scalar)* on page 19-1313.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.92 SRSHR (scalar)

Signed Rounding Shift Right (immediate).

Syntax

```
SRSHR Vd, Vn, #shift
```

Where:

V
Is a width specifier, D.

d
Is the number of the SIMD and FP destination register.

n
Is the number of the first SIMD and FP source register.

shift
Is the right shift amount, in the range 1 to 64.

Usage

Signed Rounding Shift Right (immediate). This instruction reads each vector element in the source SIMD and FP register, right shifts each result by an immediate value, places the final result into a vector, and writes the vector to the destination SIMD and FP register. All the values in this instruction are signed integer values. The results are rounded. For truncated results, see *19.95 SSHR (scalar)* on page 19-1314.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.93 SRSRA (scalar)

Signed Rounding Shift Right and Accumulate (immediate).

Syntax

`SRSRA Vd, Vn, #shift`

Where:

V
Is a width specifier, `D`.

d
Is the number of the SIMD and FP destination register.

n
Is the number of the first SIMD and FP source register.

shift
Is the right shift amount, in the range 1 to 64.

Usage

Signed Rounding Shift Right and Accumulate (immediate). This instruction reads each vector element in the source SIMD and FP register, right shifts each result by an immediate value, and accumulates the final results with the vector elements of the destination SIMD and FP register. All the values in this instruction are signed integer values. The results are rounded. For truncated results, see *19.96 SSRA (scalar)* on page 19-1315.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.94 SSHL (scalar)

Signed Shift Left (register).

Syntax

`SSHL Vd, Vn, Vm`

Where:

V
Is a width specifier, `D`.

d
Is the number of the SIMD and FP destination register.

n
Is the number of the first SIMD and FP source register.

m
Is the number of the second SIMD and FP source register.

Usage

Signed Shift Left (register). This instruction takes each signed integer value in the vector of the first source SIMD and FP register, shifts each value by a value from the least significant byte of the corresponding element of the second source SIMD and FP register, places the results in a vector, and writes the vector to the destination SIMD and FP register.

If the shift value is positive, the operation is a left shift. If the shift value is negative, it is a truncating right shift. For a rounding shift, see *19.91 SRSHL (scalar)* on page 19-1310.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.95 SSHR (scalar)

Signed Shift Right (immediate).

Syntax

`SSHR Vd, Vn, #shift`

Where:

V
Is a width specifier, `D`.

d
Is the number of the SIMD and FP destination register.

n
Is the number of the first SIMD and FP source register.

shift
Is the right shift amount, in the range 1 to 64.

Usage

Signed Shift Right (immediate). This instruction reads each vector element in the source SIMD and FP register, right shifts each result by an immediate value, places the final result into a vector, and writes the vector to the destination SIMD and FP register. All the values in this instruction are signed integer values. The results are truncated. For rounded results, see *19.92 SRSHR (scalar)* on page 19-1311.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.96 SSRA (scalar)

Signed Shift Right and Accumulate (immediate).

Syntax

`SSRA Vd, Vn, #shift`

Where:

V
Is a width specifier, `D`.

d
Is the number of the SIMD and FP destination register.

n
Is the number of the first SIMD and FP source register.

shift
Is the right shift amount, in the range 1 to 64.

Usage

Signed Shift Right and Accumulate (immediate). This instruction reads each vector element in the source SIMD and FP register, right shifts each result by an immediate value, and accumulates the final results with the vector elements of the destination SIMD and FP register. All the values in this instruction are signed integer values. The results are truncated. For rounded results, see *19.93 SRSRA (scalar)* on page 19-1312.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.97 SUB (scalar)

Subtract (vector).

Syntax

`SUB Vd, Vn, Vm`

Where:

V
Is a width specifier, `D`.

d
Is the number of the SIMD and FP destination register.

n
Is the number of the first SIMD and FP source register.

m
Is the number of the second SIMD and FP source register.

Usage

Subtract (vector). This instruction subtracts each vector element in the second source SIMD and FP register from the corresponding vector element in the first source SIMD and FP register, places the result into a vector, and writes the vector to the destination SIMD and FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.98 SUQADD (scalar)

Signed saturating Accumulate of Unsigned value.

Syntax

`SUQADD` *`Vd, Vn`*

Where:

V
: Is a width specifier, and can be one of `B`, `H`, `S` or `D`.

d
: Is the number of the SIMD and FP destination register.

n
: Is the number of the SIMD and FP source register.

Usage

Signed saturating Accumulate of Unsigned value. This instruction adds the unsigned integer values of the vector elements in the source SIMD and FP register to corresponding signed integer values of the vector elements in the destination SIMD and FP register, and writes the resulting signed integer values to the destination SIMD and FP register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.99 UCVTF (scalar, fixed-point)

Unsigned fixed-point Convert to Floating-point (vector).

Syntax

```
UCVTF Vd, Vn, #fbits
```

Where:

V
Is a width specifier, and can be one of the values shown in Usage.

d
Is the number of the SIMD and FP destination register.

n
Is the number of the first SIMD and FP source register.

fbits
Is the number of fractional bits, in the range 1 to the operand width.

Usage

Unsigned fixed-point Convert to Floating-point (vector). This instruction converts each element in a vector from fixed-point to floating-point using the rounding mode that is specified by the FPCR, and writes the result to the SIMD and FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 19-30 UCVTF (Scalar) specifier combinations

V	*fbits*
H	
S	1 to 32
D	1 to 64

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.100 UCVTF (scalar, integer)

Unsigned integer Convert to Floating-point (vector).

Syntax

```
UCVTF Hd, Hn ; Scalar half precision

UCVTF Vd, Vn ; Scalar single-precision and double-precision
```

Where:

Hd
Is the 16-bit name of the SIMD and FP destination register.

Hn
Is the 16-bit name of the SIMD and FP source register.

V
Is a width specifier, and can be either `S` or `D`.

d
Is the number of the SIMD and FP destination register.

n
Is the number of the SIMD and FP source register.

Architectures supported (scalar)

Supported in the Armv8.2 architecture and later.

Usage

Unsigned integer Convert to Floating-point (vector). This instruction converts each element in a vector from an unsigned integer value to a floating-point value using the rounding mode that is specified by the FPCR, and writes the result to the SIMD and FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.101 UQADD (scalar)

Unsigned saturating Add.

Syntax

`UQADD Vd, Vn, Vm`

Where:

V
Is a width specifier, and can be one of `B`, `H`, `S` or `D`.

d
Is the number of the SIMD and FP destination register.

n
Is the number of the first SIMD and FP source register.

m
Is the number of the second SIMD and FP source register.

Usage

Unsigned saturating Add. This instruction adds the values of corresponding elements of the two source SIMD and FP registers, places the results into a vector, and writes the vector to the destination SIMD and FP register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.102 UQRSHL (scalar)

Unsigned saturating Rounding Shift Left (register).

Syntax

`UQRSHL Vd, Vn, Vm`

Where:

V
Is a width specifier, and can be one of `B`, `H`, `S` or `D`.

d
Is the number of the SIMD and FP destination register.

n
Is the number of the first SIMD and FP source register.

m
Is the number of the second SIMD and FP source register.

Usage

Unsigned saturating Rounding Shift Left (register). This instruction takes each vector element of the first source SIMD and FP register, shifts the vector element by a value from the least significant byte of the corresponding vector element of the second source SIMD and FP register, places the results into a vector, and writes the vector to the destination SIMD and FP register.

If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift. The results are rounded. For truncated results, see *19.104 UQSHL (scalar, immediate)* on page 19-1323.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.103 UQRSHRN (scalar)

Unsigned saturating Rounded Shift Right Narrow (immediate).

Syntax

```
UQRSHRN Vbd, Van, #shift
```

Where:

Vb
Is the destination width specifier, and can be one of the values shown in Usage.

d
Is the number of the SIMD and FP destination register.

Va
Is the source width specifier, and can be one of the values shown in Usage.

n
Is the number of the first SIMD and FP source register.

shift
Is the right shift amount, in the range 1 to the destination operand width in bits, and can be one of the values shown in Usage.

Usage

Unsigned saturating Rounded Shift Right Narrow (immediate). This instruction reads each vector element in the source SIMD and FP register, right shifts each result by an immediate value, puts the final result into a vector, and writes the vector to the lower or upper half of the destination SIMD and FP register. All the values in this instruction are unsigned integer values. The results are rounded. For truncated results, see *19.106 UQSHRN (scalar)* on page 19-1325.

The `UQRSHRN` instruction writes the vector to the lower half of the destination register and clears the upper half, while the `UQRSHRN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 19-31 UQRSHRN (Scalar) specifier combinations

Vb	*Va*	*shift*
B	H	1 to 8
H	S	1 to 16
S	D	1 to 32

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.104 UQSHL (scalar, immediate)

Unsigned saturating Shift Left (immediate).

Syntax

```
UQSHL Vd, Vn, #shift
```

Where:

V
Is a width specifier, and can be one of the values shown in Usage.

d
Is the number of the SIMD and FP destination register.

n
Is the number of the first SIMD and FP source register.

shift
Is the left shift amount, in the range 0 to the operand width in bits minus 1, and can be one of the values shown in Usage.

Usage

Unsigned saturating Shift Left (immediate). This instruction takes each vector element in the source SIMD and FP register, shifts it by an immediate value, places the results in a vector, and writes the vector to the destination SIMD and FP register. The results are truncated. For rounded results, see *19.102 UQRSHL (scalar)* on page 19-1321.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 19-32 UQSHL (Scalar) specifier combinations

V	*shift*
B	0 to 7
H	0 to 15
S	0 to 31
D	0 to 63

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.105 UQSHL (scalar, register)

Unsigned saturating Shift Left (register).

Syntax

`UQSHL Vd, Vn, Vm`

Where:

V
Is a width specifier, and can be one of `B`, `H`, `S` or `D`.

d
Is the number of the SIMD and FP destination register.

n
Is the number of the first SIMD and FP source register.

m
Is the number of the second SIMD and FP source register.

Usage

Unsigned saturating Shift Left (register). This instruction takes each element in the vector of the first source SIMD and FP register, shifts the element by a value from the least significant byte of the corresponding element of the second source SIMD and FP register, places the results in a vector, and writes the vector to the destination SIMD and FP register.

If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift. The results are truncated. For rounded results, see *19.102 UQRSHL (scalar)* on page 19-1321.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.106 UQSHRN (scalar)

Unsigned saturating Shift Right Narrow (immediate).

Syntax

`UQSHRN Vbd, Van, #shift`

Where:

Vb
Is the destination width specifier, and can be one of the values shown in Usage.

d
Is the number of the SIMD and FP destination register.

Va
Is the source width specifier, and can be one of the values shown in Usage.

n
Is the number of the first SIMD and FP source register.

shift
Is the right shift amount, in the range 1 to the destination operand width in bits, and can be one of the values shown in Usage.

Usage

Unsigned saturating Shift Right Narrow (immediate). This instruction reads each vector element in the source SIMD and FP register, right shifts each result by an immediate value, saturates each shifted result to a value that is half the original width, puts the final result into a vector, and writes the vector to the lower or upper half of the destination SIMD and FP register. All the values in this instruction are unsigned integer values. The results are truncated. For rounded results, see *19.103 UQRSHRN (scalar)* on page 19-1322.

The `UQSHRN` instruction writes the vector to the lower half of the destination register and clears the upper half, while the `UQSHRN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 19-33 UQSHRN (Scalar) specifier combinations

Vb	*Va*	*shift*
B	H	1 to 8
H	S	1 to 16
S	D	1 to 32

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.107 UQSUB (scalar)

Unsigned saturating Subtract.

Syntax

`UQSUB Vd, Vn, Vm`

Where:

V
Is a width specifier, and can be one of `B`, `H`, `S` or `D`.

d
Is the number of the SIMD and FP destination register.

n
Is the number of the first SIMD and FP source register.

m
Is the number of the second SIMD and FP source register.

Usage

Unsigned saturating Subtract. This instruction subtracts the element values of the second source SIMD and FP register from the corresponding element values of the first source SIMD and FP register, places the results into a vector, and writes the vector to the destination SIMD and FP register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.108 UQXTN (scalar)

Unsigned saturating extract Narrow.

Syntax

`UQXTN Vbd, Van`

Where:

Vb
Is the destination width specifier, and can be one of the values shown in Usage.

d
Is the number of the SIMD and FP destination register.

Va
Is the source width specifier, and can be one of the values shown in Usage.

n
Is the number of the SIMD and FP source register.

Usage

Unsigned saturating extract Narrow. This instruction reads each vector element from the source SIMD and FP register, saturates each value to half the original width, places the result into a vector, and writes the vector to the destination SIMD and FP register. All the values in this instruction are unsigned integer values.

If saturation occurs, the cumulative saturation bit FPSR.QC is set.

The `UQXTN` instruction writes the vector to the lower half of the destination register and clears the upper half, while the `UQXTN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 19-34 UQXTN (Scalar) specifier combinations

Vb	*Va*
B	H
H	S
S	D

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.109 URSHL (scalar)

Unsigned Rounding Shift Left (register).

Syntax

`URSHL Vd, Vn, Vm`

Where:

V
Is a width specifier, `D`.

d
Is the number of the SIMD and FP destination register.

n
Is the number of the first SIMD and FP source register.

m
Is the number of the second SIMD and FP source register.

Usage

Unsigned Rounding Shift Left (register). This instruction takes each element in the vector of the first source SIMD and FP register, shifts the vector element by a value from the least significant byte of the corresponding element of the second source SIMD and FP register, places the results in a vector, and writes the vector to the destination SIMD and FP register.

If the shift value is positive, the operation is a left shift. If the shift value is negative, it is a rounding right shift.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.110 URSHR (scalar)

Unsigned Rounding Shift Right (immediate).

Syntax

```
URSHR Vd, Vn, #shift
```

Where:

V
Is a width specifier, D.

d
Is the number of the SIMD and FP destination register.

n
Is the number of the first SIMD and FP source register.

shift
Is the right shift amount, in the range 1 to 64.

Usage

Unsigned Rounding Shift Right (immediate). This instruction reads each vector element in the source SIMD and FP register, right shifts each result by an immediate value, writes the final result to a vector, and writes the vector to the destination SIMD and FP register. All the values in this instruction are unsigned integer values. The results are rounded. For truncated results, see *19.113 USHR (scalar)* on page 19-1332.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.111 URSRA (scalar)

Unsigned Rounding Shift Right and Accumulate (immediate).

Syntax

`URSRA Vd, Vn, #shift`

Where:

V
Is a width specifier, `D`.

d
Is the number of the SIMD and FP destination register.

n
Is the number of the first SIMD and FP source register.

shift
Is the right shift amount, in the range 1 to 64.

Usage

Unsigned Rounding Shift Right and Accumulate (immediate). This instruction reads each vector element in the source SIMD and FP register, right shifts each result by an immediate value, and accumulates the final results with the vector elements of the destination SIMD and FP register. All the values in this instruction are unsigned integer values. The results are rounded. For truncated results, see *19.115 USRA (scalar)* on page 19-1334.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.112 USHL (scalar)

Unsigned Shift Left (register).

Syntax

`USHL Vd, Vn, Vm`

Where:

V
Is a width specifier, `D`.

d
Is the number of the SIMD and FP destination register.

n
Is the number of the first SIMD and FP source register.

m
Is the number of the second SIMD and FP source register.

Usage

Unsigned Shift Left (register). This instruction takes each element in the vector of the first source SIMD and FP register, shifts each element by a value from the least significant byte of the corresponding element of the second source SIMD and FP register, places the results in a vector, and writes the vector to the destination SIMD and FP register.

If the shift value is positive, the operation is a left shift. If the shift value is negative, it is a truncating right shift. For a rounding shift, see *19.109 URSHL (scalar)* on page 19-1328.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.113 USHR (scalar)

Unsigned Shift Right (immediate).

Syntax

```
USHR Vd, Vn, #shift
```

Where:

V
Is a width specifier, `D`.

d
Is the number of the SIMD and FP destination register.

n
Is the number of the first SIMD and FP source register.

shift
Is the right shift amount, in the range 1 to 64.

Usage

Unsigned Shift Right (immediate). This instruction reads each vector element in the source SIMD and FP register, right shifts each result by an immediate value, writes the final result to a vector, and writes the vector to the destination SIMD and FP register. All the values in this instruction are unsigned integer values. The results are truncated. For rounded results, see *19.110 URSHR (scalar)* on page 19-1329.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.114 USQADD (scalar)

Unsigned saturating Accumulate of Signed value.

Syntax

`USQADD Vd, Vn`

Where:

V
Is a width specifier, and can be one of `B`, `H`, `S` or `D`.

d
Is the number of the SIMD and FP destination register.

n
Is the number of the SIMD and FP source register.

Usage

Unsigned saturating Accumulate of Signed value. This instruction adds the signed integer values of the vector elements in the source SIMD and FP register to corresponding unsigned integer values of the vector elements in the destination SIMD and FP register, and accumulates the resulting unsigned integer values with the vector elements of the destination SIMD and FP register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

19.115 USRA (scalar)

Unsigned Shift Right and Accumulate (immediate).

Syntax

`USRA Vd, Vn, #shift`

Where:

V
Is a width specifier, `D`.

d
Is the number of the SIMD and FP destination register.

n
Is the number of the first SIMD and FP source register.

shift
Is the right shift amount, in the range 1 to 64.

Usage

Unsigned Shift Right and Accumulate (immediate). This instruction reads each vector element in the source SIMD and FP register, right shifts each result by an immediate value, and accumulates the final results with the vector elements of the destination SIMD and FP register. All the values in this instruction are unsigned integer values. The results are truncated. For rounded results, see *19.111 URSRA (scalar)* on page 19-1330.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

Chapter 20
A64 SIMD Vector Instructions

Describes the A64 SIMD vector instructions.

It contains the following sections:

20.1 A64 SIMD Vector instructions in alphabetical order

A summary of the A64 SIMD Vector instructions that are supported.

Table 20-1 Summary of A64 SIMD Vector instructions

Mnemonic	Brief description	See
ABS (vector)	Absolute value (vector)	*20.2 ABS (vector)* on page 20-1351
ADD (vector)	Add (vector)	*20.3 ADD (vector)* on page 20-1352
ADDHN, ADDHN2 (vector)	Add returning High Narrow	*20.4 ADDHN, ADDHN2 (vector)* on page 20-1353
ADDP (vector)	Add Pairwise (vector)	*20.5 ADDP (vector)* on page 20-1354
ADDV (vector)	Add across Vector	*20.6 ADDV (vector)* on page 20-1355
AND (vector)	Bitwise AND (vector)	*20.7 AND (vector)* on page 20-1356
BIC (vector, immediate)	Bitwise bit Clear (vector, immediate)	*20.8 BIC (vector, immediate)* on page 20-1357
BIC (vector, register)	Bitwise bit Clear (vector, register)	*20.9 BIC (vector, register)* on page 20-1358
BIF (vector)	Bitwise Insert if False	*20.10 BIF (vector)* on page 20-1359
BIT (vector)	Bitwise Insert if True	*20.11 BIT (vector)* on page 20-1360
BSL (vector)	Bitwise Select	*20.12 BSL (vector)* on page 20-1361
CLS (vector)	Count Leading Sign bits (vector)	*20.13 CLS (vector)* on page 20-1362
CLZ (vector)	Count Leading Zero bits (vector)	*20.14 CLZ (vector)* on page 20-1363
CMEQ (vector, register)	Compare bitwise Equal (vector)	*20.15 CMEQ (vector, register)* on page 20-1364
CMEQ (vector, zero)	Compare bitwise Equal to zero (vector)	*20.16 CMEQ (vector, zero)* on page 20-1365
CMGE (vector, register)	Compare signed Greater than or Equal (vector)	*20.17 CMGE (vector, register)* on page 20-1366
CMGE (vector, zero)	Compare signed Greater than or Equal to zero (vector)	*20.18 CMGE (vector, zero)* on page 20-1367
CMGT (vector, register)	Compare signed Greater than (vector)	*20.19 CMGT (vector, register)* on page 20-1368
CMGT (vector, zero)	Compare signed Greater than zero (vector)	*20.20 CMGT (vector, zero)* on page 20-1369
CMHI (vector, register)	Compare unsigned Higher (vector)	*20.21 CMHI (vector, register)* on page 20-1370
CMHS (vector, register)	Compare unsigned Higher or Same (vector)	*20.22 CMHS (vector, register)* on page 20-1371
CMLE (vector, zero)	Compare signed Less than or Equal to zero (vector)	*20.23 CMLE (vector, zero)* on page 20-1372
CMLT (vector, zero)	Compare signed Less than zero (vector)	*20.24 CMLT (vector, zero)* on page 20-1373
CMTST (vector)	Compare bitwise Test bits nonzero (vector)	*20.25 CMTST (vector)* on page 20-1374
CNT (vector)	Population Count per byte	*20.26 CNT (vector)* on page 20-1375
DUP (vector, element)	vector	*20.27 DUP (vector, element)* on page 20-1376
DUP (vector, general)	Duplicate general-purpose register to vector	*20.28 DUP (vector, general)* on page 20-1377
EOR (vector)	Bitwise Exclusive OR (vector)	*20.29 EOR (vector)* on page 20-1378
EXT (vector)	Extract vector from pair of vectors	*20.30 EXT (vector)* on page 20-1379
FABD (vector)	Floating-point Absolute Difference (vector)	*20.31 FABD (vector)* on page 20-1380

Table 20-1 Summary of A64 SIMD Vector instructions (continued)

Mnemonic	Brief description	See
FABS (vector)	Floating-point Absolute value (vector)	*20.32 FABS (vector)* on page 20-1381
FACGE (vector)	Floating-point Absolute Compare Greater than or Equal (vector)	*20.33 FACGE (vector)* on page 20-1382
FACGT (vector)	Floating-point Absolute Compare Greater than (vector)	*20.34 FACGT (vector)* on page 20-1383
FADD (vector)	Floating-point Add (vector)	*20.35 FADD (vector)* on page 20-1384
FADDP (vector)	Floating-point Add Pairwise (vector)	*20.36 FADDP (vector)* on page 20-1385
FCADD (vector)	Floating-point Complex Add	*20.37 FCADD (vector)* on page 20-1386
FCMEQ (vector, register)	Floating-point Compare Equal (vector)	*20.38 FCMEQ (vector, register)* on page 20-1387
FCMEQ (vector, zero)	Floating-point Compare Equal to zero (vector)	*20.39 FCMEQ (vector, zero)* on page 20-1388
FCMGE (vector, register)	Floating-point Compare Greater than or Equal (vector)	*20.40 FCMGE (vector, register)* on page 20-1389
FCMGE (vector, zero)	Floating-point Compare Greater than or Equal to zero (vector)	*20.41 FCMGE (vector, zero)* on page 20-1390
FCMGT (vector, register)	Floating-point Compare Greater than (vector)	*20.42 FCMGT (vector, register)* on page 20-1391
FCMGT (vector, zero)	Floating-point Compare Greater than zero (vector)	*20.43 FCMGT (vector, zero)* on page 20-1392
FCMLA (vector)	Floating-point Complex Multiply Accumulate	*20.44 FCMLA (vector)* on page 20-1393
FCMLE (vector, zero)	Floating-point Compare Less than or Equal to zero (vector)	*20.45 FCMLE (vector, zero)* on page 20-1394
FCMLT (vector, zero)	Floating-point Compare Less than zero (vector)	*20.46 FCMLT (vector, zero)* on page 20-1395
FCVTAS (vector)	Floating-point Convert to Signed integer, rounding to nearest with ties to Away (vector)	*20.47 FCVTAS (vector)* on page 20-1396
FCVTAU (vector)	Floating-point Convert to Unsigned integer, rounding to nearest with ties to Away (vector)	*20.48 FCVTAU (vector)* on page 20-1397
FCVTL, FCVTL2 (vector)	Floating-point Convert to higher precision Long (vector)	*20.49 FCVTL, FCVTL2 (vector)* on page 20-1398
FCVTMS (vector)	Floating-point Convert to Signed integer, rounding toward Minus infinity (vector)	*20.50 FCVTMS (vector)* on page 20-1399
FCVTMU (vector)	Floating-point Convert to Unsigned integer, rounding toward Minus infinity (vector)	*20.51 FCVTMU (vector)* on page 20-1400
FCVTN, FCVTN2 (vector)	Floating-point Convert to lower precision Narrow (vector)	*20.52 FCVTN, FCVTN2 (vector)* on page 20-1401
FCVTNS (vector)	Floating-point Convert to Signed integer, rounding to nearest with ties to even (vector)	*20.53 FCVTNS (vector)* on page 20-1402
FCVTNU (vector)	Floating-point Convert to Unsigned integer, rounding to nearest with ties to even (vector)	*20.54 FCVTNU (vector)* on page 20-1403
FCVTPS (vector)	Floating-point Convert to Signed integer, rounding toward Plus infinity (vector)	*20.55 FCVTPS (vector)* on page 20-1404

Table 20-1 Summary of A64 SIMD Vector instructions (continued)

Mnemonic	Brief description	See
FCVTPU (vector)	Floating-point Convert to Unsigned integer, rounding toward Plus infinity (vector)	*20.56 FCVTPU (vector)* on page 20-1405
FCVTXN, FCVTXN2 (vector)	Floating-point Convert to lower precision Narrow, rounding to odd (vector)	*20.57 FCVTXN, FCVTXN2 (vector)* on page 20-1406
FCVTZS (vector, fixed-point)	Floating-point Convert to Signed fixed-point, rounding toward Zero (vector)	*20.58 FCVTZS (vector, fixed-point)* on page 20-1407
FCVTZS (vector, integer)	Floating-point Convert to Signed integer, rounding toward Zero (vector)	*20.59 FCVTZS (vector, integer)* on page 20-1408
FCVTZU (vector, fixed-point)	Floating-point Convert to Unsigned fixed-point, rounding toward Zero (vector)	*20.60 FCVTZU (vector, fixed-point)* on page 20-1409
FCVTZU (vector, integer)	Floating-point Convert to Unsigned integer, rounding toward Zero (vector)	*20.61 FCVTZU (vector, integer)* on page 20-1410
FDIV (vector)	Floating-point Divide (vector)	*20.62 FDIV (vector)* on page 20-1411
FMAX (vector)	Floating-point Maximum (vector)	*20.63 FMAX (vector)* on page 20-1412
FMAXNM (vector)	Floating-point Maximum Number (vector)	*20.64 FMAXNM (vector)* on page 20-1413
FMAXNMP (vector)	Floating-point Maximum Number Pairwise (vector)	*20.65 FMAXNMP (vector)* on page 20-1414
FMAXNMV (vector)	Floating-point Maximum Number across Vector	*20.66 FMAXNMV (vector)* on page 20-1415
FMAXP (vector)	Floating-point Maximum Pairwise (vector)	*20.67 FMAXP (vector)* on page 20-1416
FMAXV (vector)	Floating-point Maximum across Vector	*20.68 FMAXV (vector)* on page 20-1417
FMIN (vector)	Floating-point minimum (vector)	*20.69 FMIN (vector)* on page 20-1418
FMINNM (vector)	Floating-point Minimum Number (vector)	*20.70 FMINNM (vector)* on page 20-1419
FMINNMP (vector)	Floating-point Minimum Number Pairwise (vector)	*20.71 FMINNMP (vector)* on page 20-1420
FMINNMV (vector)	Floating-point Minimum Number across Vector	*20.72 FMINNMV (vector)* on page 20-1421
FMINP (vector)	Floating-point Minimum Pairwise (vector)	*20.73 FMINP (vector)* on page 20-1422
FMINV (vector)	Floating-point Minimum across Vector	*20.74 FMINV (vector)* on page 20-1423
FMLA (vector, by element)	Floating-point fused Multiply-Add to accumulator (by element)	*20.75 FMLA (vector, by element)* on page 20-1424
FMLA (vector)	Floating-point fused Multiply-Add to accumulator (vector)	*20.76 FMLA (vector)* on page 20-1426
FMLS (vector, by element)	Floating-point fused Multiply-Subtract from accumulator (by element)	*20.77 FMLS (vector, by element)* on page 20-1427
FMLS (vector)	Floating-point fused Multiply-Subtract from accumulator (vector)	*20.78 FMLS (vector)* on page 20-1429
FMOV (vector, immediate)	Floating-point move immediate (vector)	*20.79 FMOV (vector, immediate)* on page 20-1430
FMUL (vector, by element)	Floating-point Multiply (by element)	*20.80 FMUL (vector, by element)* on page 20-1432
FMUL (vector)	Floating-point Multiply (vector)	*20.81 FMUL (vector)* on page 20-1434
FMULX (vector, by element)	Floating-point Multiply extended (by element)	*20.82 FMULX (vector, by element)* on page 20-1435

Table 20-1 Summary of A64 SIMD Vector instructions (continued)

Mnemonic	Brief description	See
FMULX (vector)	Floating-point Multiply extended	*20.83 FMULX (vector)* on page 20-1437
FNEG (vector)	Floating-point Negate (vector)	*20.84 FNEG (vector)* on page 20-1438
FRECPE (vector)	Floating-point Reciprocal Estimate	*20.85 FRECPE (vector)* on page 20-1439
FRECPS (vector)	Floating-point Reciprocal Step	*20.86 FRECPS (vector)* on page 20-1440
FRECPX (vector)	Floating-point Reciprocal exponent (scalar)	*20.87 FRECPX (vector)* on page 20-1441
FRINTA (vector)	Floating-point Round to Integral, to nearest with ties to Away (vector)	*20.88 FRINTA (vector)* on page 20-1442
FRINTI (vector)	Floating-point Round to Integral, using current rounding mode (vector)	*20.89 FRINTI (vector)* on page 20-1443
FRINTM (vector)	Floating-point Round to Integral, toward Minus infinity (vector)	*20.90 FRINTM (vector)* on page 20-1444
FRINTN (vector)	Floating-point Round to Integral, to nearest with ties to even (vector)	*20.91 FRINTN (vector)* on page 20-1445
FRINTP (vector)	Floating-point Round to Integral, toward Plus infinity (vector)	*20.92 FRINTP (vector)* on page 20-1446
FRINTX (vector)	Floating-point Round to Integral exact, using current rounding mode (vector)	*20.93 FRINTX (vector)* on page 20-1447
FRINTZ (vector)	Floating-point Round to Integral, toward Zero (vector)	*20.94 FRINTZ (vector)* on page 20-1448
FRSQRTE (vector)	Floating-point Reciprocal Square Root Estimate	*20.95 FRSQRTE (vector)* on page 20-1449
FRSQRTS (vector)	Floating-point Reciprocal Square Root Step	*20.96 FRSQRTS (vector)* on page 20-1450
FSQRT (vector)	Floating-point Square Root (vector)	*20.97 FSQRT (vector)* on page 20-1451
FSUB (vector)	Floating-point Subtract (vector)	*20.98 FSUB (vector)* on page 20-1452
INS (vector, element)	Insert vector element from another vector element	*20.99 INS (vector, element)* on page 20-1453
INS (vector, general)	Insert vector element from general-purpose register	*20.100 INS (vector, general)* on page 20-1454
LD1 (vector, multiple structures)	Load multiple single-element structures to one, two, three, or four registers	*20.101 LD1 (vector, multiple structures)* on page 20-1455
LD1 (vector, single structure)	Load one single-element structure to one lane of one register	*20.102 LD1 (vector, single structure)* on page 20-1458
LD1R (vector)	Load one single-element structure and Replicate to all lanes (of one register)	*20.103 LD1R (vector)* on page 20-1459
LD2 (vector, multiple structures)	Load multiple 2-element structures to two registers	*20.104 LD2 (vector, multiple structures)* on page 20-1460
LD2 (vector, single structure)	Load single 2-element structure to one lane of two registers	*20.105 LD2 (vector, single structure)* on page 20-1461
LD2R (vector)	Load single 2-element structure and Replicate to all lanes of two registers	*20.106 LD2R (vector)* on page 20-1462
LD3 (vector, multiple structures)	Load multiple 3-element structures to three registers	*20.107 LD3 (vector, multiple structures)* on page 20-1463

Table 20-1 Summary of A64 SIMD Vector instructions (continued)

Mnemonic	Brief description	See
LD3 (vector, single structure)	Load single 3-element structure to one lane of three registers)	*20.108 LD3 (vector, single structure)* on page 20-1464
LD3R (vector)	Load single 3-element structure and Replicate to all lanes of three registers	*20.109 LD3R (vector)* on page 20-1465
LD4 (vector, multiple structures)	Load multiple 4-element structures to four registers	*20.110 LD4 (vector, multiple structures)* on page 20-1466
LD4 (vector, single structure)	Load single 4-element structure to one lane of four registers	*20.111 LD4 (vector, single structure)* on page 20-1467
LD4R (vector)	Load single 4-element structure and Replicate to all lanes of four registers	*20.112 LD4R (vector)* on page 20-1469
MLA (vector, by element)	Multiply-Add to accumulator (vector, by element)	*20.113 MLA (vector, by element)* on page 20-1470
MLA (vector)	Multiply-Add to accumulator (vector)	*20.114 MLA (vector)* on page 20-1471
MLS (vector, by element)	Multiply-Subtract from accumulator (vector, by element)	*20.115 MLS (vector, by element)* on page 20-1472
MLS (vector)	Multiply-Subtract from accumulator (vector)	*20.116 MLS (vector)* on page 20-1473
MOV (vector, element)	Move vector element to another vector element	*20.117 MOV (vector, element)* on page 20-1474
MOV (vector, from general)	Move general-purpose register to a vector element	*20.118 MOV (vector, from general)* on page 20-1475
MOV (vector)	Move vector	*20.119 MOV (vector)* on page 20-1476
MOV (vector, to general)	Move vector element to general-purpose register	*20.120 MOV (vector, to general)* on page 20-1477
MOVI (vector)	Move Immediate (vector)	*20.121 MOVI (vector)* on page 20-1478
MUL (vector, by element)	Multiply (vector, by element)	*20.122 MUL (vector, by element)* on page 20-1479
MUL (vector)	Multiply (vector)	*20.123 MUL (vector)* on page 20-1480
MVN (vector)	Bitwise NOT (vector)	*20.124 MVN (vector)* on page 20-1481
MVNI (vector)	Move inverted Immediate (vector)	*20.125 MVNI (vector)* on page 20-1482
NEG (vector)	Negate (vector)	*20.126 NEG (vector)* on page 20-1483
NOT (vector)	Bitwise NOT (vector)	*20.127 NOT (vector)* on page 20-1484
ORN (vector)	Bitwise inclusive OR NOT (vector)	*20.128 ORN (vector)* on page 20-1485
ORR (vector, immediate)	Bitwise inclusive OR (vector, immediate)	*20.129 ORR (vector, immediate)* on page 20-1486
ORR (vector, register)	Bitwise inclusive OR (vector, register)	*20.130 ORR (vector, register)* on page 20-1487
PMUL (vector)	Polynomial Multiply	*20.131 PMUL (vector)* on page 20-1488
PMULL, PMULL2 (vector)	Polynomial Multiply Long	*20.132 PMULL, PMULL2 (vector)* on page 20-1489
RADDHN, RADDHN2 (vector)	Rounding Add returning High Narrow	*20.133 RADDHN, RADDHN2 (vector)* on page 20-1490
RBIT (vector)	Reverse Bit order (vector)	*20.134 RBIT (vector)* on page 20-1491
REV16 (vector)	Reverse elements in 16-bit halfwords (vector)	*20.135 REV16 (vector)* on page 20-1492
REV32 (vector)	Reverse elements in 32-bit words (vector)	*20.136 REV32 (vector)* on page 20-1493

Table 20-1 Summary of A64 SIMD Vector instructions (continued)

Mnemonic	Brief description	See
REV64 (vector)	Reverse elements in 64-bit doublewords (vector)	*20.137 REV64 (vector)* on page 20-1494
RSHRN, RSHRN2 (vector)	Rounding Shift Right Narrow (immediate)	*20.138 RSHRN, RSHRN2 (vector)* on page 20-1495
RSUBHN, RSUBHN2 (vector)	Rounding Subtract returning High Narrow	*20.139 RSUBHN, RSUBHN2 (vector)* on page 20-1496
SABA (vector)	Signed Absolute difference and Accumulate	*20.140 SABA (vector)* on page 20-1497
SABAL, SABAL2 (vector)	Signed Absolute difference and Accumulate Long	*20.141 SABAL, SABAL2 (vector)* on page 20-1498
SABD (vector)	Signed Absolute Difference	*20.142 SABD (vector)* on page 20-1499
SABDL, SABDL2 (vector)	Signed Absolute Difference Long	*20.143 SABDL, SABDL2 (vector)* on page 20-1500
SADALP (vector)	Signed Add and Accumulate Long Pairwise	*20.144 SADALP (vector)* on page 20-1501
SADDL, SADDL2 (vector)	Signed Add Long (vector)	*20.145 SADDL, SADDL2 (vector)* on page 20-1502
SADDLP (vector)	Signed Add Long Pairwise	*20.146 SADDLP (vector)* on page 20-1503
SADDLV (vector)	Signed Add Long across Vector	*20.147 SADDLV (vector)* on page 20-1504
SADDW, SADDW2 (vector)	Signed Add Wide	*20.148 SADDW, SADDW2 (vector)* on page 20-1505
SCVTF (vector, fixed-point)	Signed fixed-point Convert to Floating-point (vector)	*20.149 SCVTF (vector, fixed-point)* on page 20-1506
SCVTF (vector, integer)	Signed integer Convert to Floating-point (vector)	*20.150 SCVTF (vector, integer)* on page 20-1507
SHADD (vector)	Signed Halving Add	*20.151 SHADD (vector)* on page 20-1508
SHL (vector)	Shift Left (immediate)	*20.152 SHL (vector)* on page 20-1509
SHLL, SHLL2 (vector)	Shift Left Long (by element size)	*20.153 SHLL, SHLL2 (vector)* on page 20-1510
SHRN, SHRN2 (vector)	Shift Right Narrow (immediate)	*20.154 SHRN, SHRN2 (vector)* on page 20-1511
SHSUB (vector)	Signed Halving Subtract	*20.155 SHSUB (vector)* on page 20-1512
SLI (vector)	Shift Left and Insert (immediate)	*20.156 SLI (vector)* on page 20-1513
SMAX (vector)	Signed Maximum (vector)	*20.157 SMAX (vector)* on page 20-1514
SMAXP (vector)	Signed Maximum Pairwise	*20.158 SMAXP (vector)* on page 20-1515
SMAXV (vector)	Signed Maximum across Vector	*20.159 SMAXV (vector)* on page 20-1516
SMIN (vector)	Signed Minimum (vector)	*20.160 SMIN (vector)* on page 20-1517
SMINP (vector)	Signed Minimum Pairwise	*20.161 SMINP (vector)* on page 20-1518
SMINV (vector)	Signed Minimum across Vector	*20.162 SMINV (vector)* on page 20-1519
SMLAL, SMLAL2 (vector, by element)	Signed Multiply-Add Long (vector, by element)	*20.163 SMLAL, SMLAL2 (vector, by element)* on page 20-1520
SMLAL, SMLAL2 (vector)	Signed Multiply-Add Long (vector)	*20.164 SMLAL, SMLAL2 (vector)* on page 20-1521
SMLSL, SMLSL2 (vector, by element)	Signed Multiply-Subtract Long (vector, by element)	*20.165 SMLSL, SMLSL2 (vector, by element)* on page 20-1522
SMLSL, SMLSL2 (vector)	Signed Multiply-Subtract Long (vector)	*20.166 SMLSL, SMLSL2 (vector)* on page 20-1523
SMOV (vector)	Signed Move vector element to general-purpose register	*20.167 SMOV (vector)* on page 20-1524

Table 20-1 Summary of A64 SIMD Vector instructions (continued)

Mnemonic	Brief description	See
SMULL, SMULL2 (vector, by element)	Signed Multiply Long (vector, by element)	*20.168 SMULL, SMULL2 (vector, by element)* on page 20-1525
SMULL, SMULL2 (vector)	Signed Multiply Long (vector)	*20.169 SMULL, SMULL2 (vector)* on page 20-1526
SQABS (vector)	Signed saturating Absolute value	*20.170 SQABS (vector)* on page 20-1527
SQADD (vector)	Signed saturating Add	*20.171 SQADD (vector)* on page 20-1528
SQDMLAL, SQDMLAL2 (vector, by element)	Signed saturating Doubling Multiply-Add Long (by element)	*20.172 SQDMLAL, SQDMLAL2 (vector, by element)* on page 20-1529
SQDMLAL, SQDMLAL2 (vector)	Signed saturating Doubling Multiply-Add Long	*20.173 SQDMLAL, SQDMLAL2 (vector)* on page 20-1531
SQDMLSL, SQDMLSL2 (vector, by element)	Signed saturating Doubling Multiply-Subtract Long (by element)	*20.174 SQDMLSL, SQDMLSL2 (vector, by element)* on page 20-1532
SQDMLSL, SQDMLSL2 (vector)	Signed saturating Doubling Multiply-Subtract Long	*20.175 SQDMLSL, SQDMLSL2 (vector)* on page 20-1534
SQDMULH (vector, by element)	Signed saturating Doubling Multiply returning High half (by element)	*20.176 SQDMULH (vector, by element)* on page 20-1535
SQDMULH (vector)	Signed saturating Doubling Multiply returning High half	*20.177 SQDMULH (vector)* on page 20-1536
SQDMULL, SQDMULL2 (vector, by element)	Signed saturating Doubling Multiply Long (by element)	*20.178 SQDMULL, SQDMULL2 (vector, by element)* on page 20-1537
SQDMULL, SQDMULL2 (vector)	Signed saturating Doubling Multiply Long	*20.179 SQDMULL, SQDMULL2 (vector)* on page 20-1539
SQNEG (vector)	Signed saturating Negate	*20.180 SQNEG (vector)* on page 20-1540
SQRDMLAH (vector, by element)	Signed Saturating Rounding Doubling Multiply Accumulate returning High Half (by element)	*20.181 SQRDMLAH (vector, by element)* on page 20-1541
SQRDMLAH (vector)	Signed Saturating Rounding Doubling Multiply Accumulate returning High Half (vector)	*20.182 SQRDMLAH (vector)* on page 20-1542
SQRDMLSH (vector, by element)	Signed Saturating Rounding Doubling Multiply Subtract returning High Half (by element)	*20.183 SQRDMLSH (vector, by element)* on page 20-1543
SQRDMLSH (vector)	Signed Saturating Rounding Doubling Multiply Subtract returning High Half (vector)	*20.184 SQRDMLSH (vector)* on page 20-1544
SQRDMULH (vector, by element)	Signed saturating Rounding Doubling Multiply returning High half (by element)	*20.185 SQRDMULH (vector, by element)* on page 20-1545
SQRDMULH (vector)	Signed saturating Rounding Doubling Multiply returning High half	*20.186 SQRDMULH (vector)* on page 20-1546
SQRSHL (vector)	Signed saturating Rounding Shift Left (register)	*20.187 SQRSHL (vector)* on page 20-1547
SQRSHRN, SQRSHRN2 (vector)	Signed saturating Rounded Shift Right Narrow (immediate)	*20.188 SQRSHRN, SQRSHRN2 (vector)* on page 20-1548
SQRSHRUN, SQRSHRUN2 (vector)	Signed saturating Rounded Shift Right Unsigned Narrow (immediate)	*20.189 SQRSHRUN, SQRSHRUN2 (vector)* on page 20-1549
SQSHL (vector, immediate)	Signed saturating Shift Left (immediate)	*20.190 SQSHL (vector, immediate)* on page 20-1550

Table 20-1 Summary of A64 SIMD Vector instructions (continued)

Mnemonic	Brief description	See
SQSHL (vector, register)	Signed saturating Shift Left (register)	*20.191 SQSHL (vector, register)* on page 20-1551
SQSHLU (vector)	Signed saturating Shift Left Unsigned (immediate)	*20.192 SQSHLU (vector)* on page 20-1552
SQSHRN, SQSHRN2 (vector)	Signed saturating Shift Right Narrow (immediate)	*20.193 SQSHRN, SQSHRN2 (vector)* on page 20-1553
SQSHRUN, SQSHRUN2 (vector)	Signed saturating Shift Right Unsigned Narrow (immediate)	*20.194 SQSHRUN, SQSHRUN2 (vector)* on page 20-1554
SQSUB (vector)	Signed saturating Subtract	*20.195 SQSUB (vector)* on page 20-1555
SQXTN, SQXTN2 (vector)	Signed saturating extract Narrow	*20.196 SQXTN, SQXTN2 (vector)* on page 20-1556
SQXTUN, SQXTUN2 (vector)	Signed saturating extract Unsigned Narrow	*20.197 SQXTUN, SQXTUN2 (vector)* on page 20-1557
SRHADD (vector)	Signed Rounding Halving Add	*20.198 SRHADD (vector)* on page 20-1558
SRI (vector)	Shift Right and Insert (immediate)	*20.199 SRI (vector)* on page 20-1559
SRSHL (vector)	Signed Rounding Shift Left (register)	*20.200 SRSHL (vector)* on page 20-1560
SRSHR (vector)	Signed Rounding Shift Right (immediate)	*20.201 SRSHR (vector)* on page 20-1561
SRSRA (vector)	Signed Rounding Shift Right and Accumulate (immediate)	*20.202 SRSRA (vector)* on page 20-1562
SSHL (vector)	Signed Shift Left (register)	*20.203 SSHL (vector)* on page 20-1563
SSHLL, SSHLL2 (vector)	Signed Shift Left Long (immediate)	*20.204 SSHLL, SSHLL2 (vector)* on page 20-1564
SSHR (vector)	Signed Shift Right (immediate)	*20.205 SSHR (vector)* on page 20-1565
SSRA (vector)	Signed Shift Right and Accumulate (immediate)	*20.206 SSRA (vector)* on page 20-1566
SSUBL, SSUBL2 (vector)	Signed Subtract Long	*20.207 SSUBL, SSUBL2 (vector)* on page 20-1567
SSUBW, SSUBW2 (vector)	Signed Subtract Wide	*20.208 SSUBW, SSUBW2 (vector)* on page 20-1568
ST1 (vector, multiple structures)	Store multiple single-element structures from one, two, three, or four registers	*20.209 ST1 (vector, multiple structures)* on page 20-1569
ST1 (vector, single structure)	Store a single-element structure from one lane of one register	*20.210 ST1 (vector, single structure)* on page 20-1572
ST2 (vector, multiple structures)	Store multiple 2-element structures from two registers	*20.211 ST2 (vector, multiple structures)* on page 20-1573
ST2 (vector, single structure)	Store single 2-element structure from one lane of two registers	*20.212 ST2 (vector, single structure)* on page 20-1574
ST3 (vector, multiple structures)	Store multiple 3-element structures from three registers	*20.213 ST3 (vector, multiple structures)* on page 20-1575
ST3 (vector, single structure)	Store single 3-element structure from one lane of three registers	*20.214 ST3 (vector, single structure)* on page 20-1576
ST4 (vector, multiple structures)	Store multiple 4-element structures from four registers	*20.215 ST4 (vector, multiple structures)* on page 20-1577
ST4 (vector, single structure)	Store single 4-element structure from one lane of four registers	*20.216 ST4 (vector, single structure)* on page 20-1578

Table 20-1 Summary of A64 SIMD Vector instructions (continued)

Mnemonic	Brief description	See
SUB (vector)	Subtract (vector)	*20.217 SUB (vector)* on page 20-1580
SUBHN, SUBHN2 (vector)	Subtract returning High Narrow	*20.218 SUBHN, SUBHN2 (vector)* on page 20-1581
SUQADD (vector)	Signed saturating Accumulate of Unsigned value	*20.219 SUQADD (vector)* on page 20-1582
SXTL, SXTL2 (vector)	Signed extend Long	*20.220 SXTL, SXTL2 (vector)* on page 20-1583
TBL (vector)	Table vector Lookup	*20.221 TBL (vector)* on page 20-1584
TBX (vector)	Table vector lookup extension	*20.222 TBX (vector)* on page 20-1585
TRN1 (vector)	Transpose vectors (primary)	*20.223 TRN1 (vector)* on page 20-1586
TRN2 (vector)	Transpose vectors (secondary)	*20.224 TRN2 (vector)* on page 20-1587
UABA (vector)	Unsigned Absolute difference and Accumulate	*20.225 UABA (vector)* on page 20-1588
UABAL, UABAL2 (vector)	Unsigned Absolute difference and Accumulate Long	*20.226 UABAL, UABAL2 (vector)* on page 20-1589
UABD (vector)	Unsigned Absolute Difference (vector)	*20.227 UABD (vector)* on page 20-1590
UABDL, UABDL2 (vector)	Unsigned Absolute Difference Long	*20.228 UABDL, UABDL2 (vector)* on page 20-1591
UADALP (vector)	Unsigned Add and Accumulate Long Pairwise	*20.229 UADALP (vector)* on page 20-1592
UADDL, UADDL2 (vector)	Unsigned Add Long (vector)	*20.230 UADDL, UADDL2 (vector)* on page 20-1593
UADDLP (vector)	Unsigned Add Long Pairwise	*20.231 UADDLP (vector)* on page 20-1594
UADDLV (vector)	Unsigned sum Long across Vector	*20.232 UADDLV (vector)* on page 20-1595
UADDW, UADDW2 (vector)	Unsigned Add Wide	*20.233 UADDW, UADDW2 (vector)* on page 20-1596
UCVTF (vector, fixed-point)	Unsigned fixed-point Convert to Floating-point (vector)	*20.234 UCVTF (vector, fixed-point)* on page 20-1597
UCVTF (vector, integer)	Unsigned integer Convert to Floating-point (vector)	*20.235 UCVTF (vector, integer)* on page 20-1598
UHADD (vector)	Unsigned Halving Add	*20.236 UHADD (vector)* on page 20-1599
UHSUB (vector)	Unsigned Halving Subtract	*20.237 UHSUB (vector)* on page 20-1600
UMAX (vector)	Unsigned Maximum (vector)	*20.238 UMAX (vector)* on page 20-1601
UMAXP (vector)	Unsigned Maximum Pairwise	*20.239 UMAXP (vector)* on page 20-1602
UMAXV (vector)	Unsigned Maximum across Vector	*20.240 UMAXV (vector)* on page 20-1603
UMIN (vector)	Unsigned Minimum (vector)	*20.241 UMIN (vector)* on page 20-1604
UMINP (vector)	Unsigned Minimum Pairwise	*20.242 UMINP (vector)* on page 20-1605
UMINV (vector)	Unsigned Minimum across Vector	*20.243 UMINV (vector)* on page 20-1606
UMLAL, UMLAL2 (vector, by element)	Unsigned Multiply-Add Long (vector, by element)	*20.244 UMLAL, UMLAL2 (vector, by element)* on page 20-1607
UMLAL, UMLAL2 (vector)	Unsigned Multiply-Add Long (vector)	*20.245 UMLAL, UMLAL2 (vector)* on page 20-1608
UMLSL, UMLSL2 (vector, by element)	Unsigned Multiply-Subtract Long (vector, by element)	*20.246 UMLSL, UMLSL2 (vector, by element)* on page 20-1609
UMLSL, UMLSL2 (vector)	Unsigned Multiply-Subtract Long (vector)	*20.247 UMLSL, UMLSL2 (vector)* on page 20-1610

Table 20-1 Summary of A64 SIMD Vector instructions (continued)

Mnemonic	Brief description	See
UMOV (vector)	Unsigned Move vector element to general-purpose register	*20.248 UMOV (vector)* on page 20-1611
UMULL, UMULL2 (vector, by element)	Unsigned Multiply Long (vector, by element)	*20.249 UMULL, UMULL2 (vector, by element)* on page 20-1612
UMULL, UMULL2 (vector)	Unsigned Multiply long (vector)	*20.250 UMULL, UMULL2 (vector)* on page 20-1613
UQADD (vector)	Unsigned saturating Add	*20.251 UQADD (vector)* on page 20-1614
UQRSHL (vector)	Unsigned saturating Rounding Shift Left (register)	*20.252 UQRSHL (vector)* on page 20-1615
UQRSHRN, UQRSHRN2 (vector)	Unsigned saturating Rounded Shift Right Narrow (immediate)	*20.253 UQRSHRN, UQRSHRN2 (vector)* on page 20-1616
UQSHL (vector, immediate)	Unsigned saturating Shift Left (immediate)	*20.254 UQSHL (vector, immediate)* on page 20-1617
UQSHL (vector, register)	Unsigned saturating Shift Left (register)	*20.255 UQSHL (vector, register)* on page 20-1618
UQSHRN, UQSHRN2 (vector)	Unsigned saturating Shift Right Narrow (immediate)	*20.256 UQSHRN, UQSHRN2 (vector)* on page 20-1619
UQSUB (vector)	Unsigned saturating Subtract	*20.257 UQSUB (vector)* on page 20-1621
UQXTN, UQXTN2 (vector)	Unsigned saturating extract Narrow	*20.258 UQXTN, UQXTN2 (vector)* on page 20-1622
URECPE (vector)	Unsigned Reciprocal Estimate	*20.259 URECPE (vector)* on page 20-1623
URHADD (vector)	Unsigned Rounding Halving Add	*20.260 URHADD (vector)* on page 20-1624
URSHL (vector)	Unsigned Rounding Shift Left (register)	*20.261 URSHL (vector)* on page 20-1625
URSHR (vector)	Unsigned Rounding Shift Right (immediate)	*20.262 URSHR (vector)* on page 20-1626
URSQRTE (vector)	Unsigned Reciprocal Square Root Estimate	*20.263 URSQRTE (vector)* on page 20-1627
URSRA (vector)	Unsigned Rounding Shift Right and Accumulate (immediate)	*20.264 URSRA (vector)* on page 20-1628
USHL (vector)	Unsigned Shift Left (register)	*20.265 USHL (vector)* on page 20-1629
USHLL, USHLL2 (vector)	Unsigned Shift Left Long (immediate)	*20.266 USHLL, USHLL2 (vector)* on page 20-1630
USHR (vector)	Unsigned Shift Right (immediate)	*20.267 USHR (vector)* on page 20-1631
USQADD (vector)	Unsigned saturating Accumulate of Signed value	*20.268 USQADD (vector)* on page 20-1632
USRA (vector)	Unsigned Shift Right and Accumulate (immediate)	*20.269 USRA (vector)* on page 20-1633
USUBL, USUBL2 (vector)	Unsigned Subtract Long	*20.270 USUBL, USUBL2 (vector)* on page 20-1634
USUBW, USUBW2 (vector)	Unsigned Subtract Wide	*20.271 USUBW, USUBW2 (vector)* on page 20-1635
UXTL, UXTL2 (vector)	Unsigned extend Long	*20.272 UXTL, UXTL2 (vector)* on page 20-1636
UZP1 (vector)	Unzip vectors (primary)	*20.273 UZP1 (vector)* on page 20-1637
UZP2 (vector)	Unzip vectors (secondary)	*20.274 UZP2 (vector)* on page 20-1638
XTN, XTN2 (vector)	Extract Narrow	*20.275 XTN, XTN2 (vector)* on page 20-1639
ZIP1 (vector)	Zip vectors (primary)	*20.276 ZIP1 (vector)* on page 20-1640
ZIP2 (vector)	Zip vectors (secondary)	*20.277 ZIP2 (vector)* on page 20-1641

20.2 ABS (vector)

Absolute value (vector).

Syntax

`ABS Vd.T, Vn.T`

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of `8B`, `16B`, `4H`, `8H`, `2S`, `4S` or `2D`.

Vn
Is the name of the SIMD and FP source register.

Usage

Absolute value (vector). This instruction calculates the absolute value of each vector element in the source SIMD and FP register, puts the result into a vector, and writes the vector to the destination SIMD and FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.3 ADD (vector)

Add (vector).

Syntax

`ADD Vd.T, Vn.T, Vm.T`

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of `8B`, `16B`, `4H`, `8H`, `2S`, `4S` or `2D`.

Vn
Is the name of the first SIMD and FP source register.

Vm
Is the name of the second SIMD and FP source register.

Usage

Add (vector). This instruction adds corresponding elements in the two source SIMD and FP registers, places the results into a vector, and writes the vector to the destination SIMD and FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.4 ADDHN, ADDHN2 (vector)

Add returning High Narrow.

Syntax

`ADDHN{2} Vd.Tb, Vn.Ta, Vm.Ta`

Where:

2
Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements. See <Q> in the Usage table.

Vd
Is the name of the SIMD and FP destination register.

Tb
Is an arrangement specifier, and can be one of the values shown in Usage.

Vn
Is the name of the first SIMD and FP source register.

Ta
Is an arrangement specifier, and can be one of the values shown in Usage.

Vm
Is the name of the second SIMD and FP source register.

Usage

Add returning High Narrow. This instruction adds each vector element in the first source SIMD and FP register to the corresponding vector element in the second source SIMD and FP register, places the most significant half of the result into a vector, and writes the vector to the lower or upper half of the destination SIMD and FP register.

The results are truncated. For rounded results, see *20.133 RADDHN, RADDHN2 (vector)* on page 20-1490.

The `ADDHN` instruction writes the vector to the lower half of the destination register and clears the upper half, while the `ADDHN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-2 ADDHN, ADDHN2 (Vector) specifier combinations

<Q>	*Tb*	*Ta*
-	8B	8H
2	16B	8H
-	4H	4S
2	8H	4S
-	2S	2D
2	4S	2D

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.5 ADDP (vector)

Add Pairwise (vector).

Syntax

`ADDP Vd.T, Vn.T, Vm.T`

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of `8B`, `16B`, `4H`, `8H`, `2S`, `4S` or `2D`.

Vn
Is the name of the first SIMD and FP source register.

Vm
Is the name of the second SIMD and FP source register.

Usage

Add Pairwise (vector). This instruction creates a vector by concatenating the vector elements of the first source SIMD and FP register after the vector elements of the second source SIMD and FP register, reads each pair of adjacent vector elements from the concatenated vector, adds each pair of values together, places the result into a vector, and writes the vector to the destination SIMD and FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.6 ADDV (vector)

Add across Vector.

Syntax

`ADDV Vd, Vn.T`

Where:

V
Is the destination width specifier, and can be one of the values shown in Usage.

d
Is the number of the SIMD and FP destination register.

Vn
Is the name of the SIMD and FP source register.

T
Is an arrangement specifier, and can be one of the values shown in Usage.

Usage

Add across Vector. This instruction adds every vector element in the source SIMD and FP register together, and writes the scalar result to the destination SIMD and FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-3 ADDV (Vector) specifier combinations

V	*T*
B	8B
B	16B
H	4H
H	8H
S	4S

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.7 AND (vector)

Bitwise AND (vector).

Syntax

`AND Vd.T, Vn.T, Vm.T`

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be either `8B` or `16B`.

Vn
Is the name of the first SIMD and FP source register.

Vm
Is the name of the second SIMD and FP source register.

Usage

Bitwise AND (vector). This instruction performs a bitwise AND between the two source SIMD and FP registers, and writes the result to the destination SIMD and FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.8 BIC (vector, immediate)

Bitwise bit Clear (vector, immediate).

Syntax

```
BIC Vd.T, #imm8{, LSL #amount} ; 16-bit
```

```
BIC Vd.T, #imm8{, LSL #amount} ; 32-bit
```

Where:

T

Is an arrangement specifier:

16-bit

Can be one of `4H` or `8H`.

32-bit

Can be one of `2S` or `4S`.

amount

Is the shift amount:

16-bit

Can be one of `0` or `8`.

32-bit

Can be one of `0`, `8`, `16` or `24`.

Defaults to zero if `LSL` is omitted.

Vd

Is the name of the SIMD and FP register.

imm8

Is an 8-bit immediate.

Usage

Bitwise bit Clear (vector, immediate). This instruction reads each vector element from the destination SIMD and FP register, performs a bitwise AND between each result and the complement of an immediate constant, places the result into a vector, and writes the vector to the destination SIMD and FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.9 BIC (vector, register)

Bitwise bit Clear (vector, register).

Syntax

`BIC` *`Vd.T, Vn.T, Vm.T`*

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be either `8B` or `16B`.

Vn
Is the name of the first SIMD and FP source register.

Vm
Is the name of the second SIMD and FP source register.

Usage

Bitwise bit Clear (vector, register). This instruction performs a bitwise AND between the first source SIMD and FP register and the complement of the second source SIMD and FP register, and writes the result to the destination SIMD and FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.10 BIF (vector)

Bitwise Insert if False.

Syntax

`BIF` *`Vd.T, Vn.T, Vm.T`*

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be either `8B` or `16B`.

Vn
Is the name of the first SIMD and FP source register.

Vm
Is the name of the second SIMD and FP source register.

Usage

Bitwise Insert if False. This instruction inserts each bit from the first source SIMD and FP register into the destination SIMD and FP register if the corresponding bit of the second source SIMD and FP register is 0, otherwise leaves the bit in the destination register unchanged.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.11 BIT (vector)

Bitwise Insert if True.

Syntax

`BIT` *`Vd.T, Vn.T, Vm.T`*

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be either `8B` or `16B`.

Vn
Is the name of the first SIMD and FP source register.

Vm
Is the name of the second SIMD and FP source register.

Usage

Bitwise Insert if True. This instruction inserts each bit from the first source SIMD and FP register into the SIMD and FP destination register if the corresponding bit of the second source SIMD and FP register is 1, otherwise leaves the bit in the destination register unchanged.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.12 BSL (vector)

Bitwise Select.

Syntax

`BSL` *`Vd.T, Vn.T, Vm.T`*

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be either `8B` or `16B`.

Vn
Is the name of the first SIMD and FP source register.

Vm
Is the name of the second SIMD and FP source register.

Usage

Bitwise Select. This instruction sets each bit in the destination SIMD and FP register to the corresponding bit from the first source SIMD and FP register when the original destination bit was 1, otherwise from the second source SIMD and FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.13 CLS (vector)

Count Leading Sign bits (vector).

Syntax

`CLS Vd.T, Vn.T`

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of `8B`, `16B`, `4H`, `8H`, `2S` or `4S`.

Vn
Is the name of the SIMD and FP source register.

Usage

Count Leading Sign bits (vector). This instruction counts the number of consecutive bits following the most significant bit that are the same as the most significant bit in each vector element in the source SIMD and FP register, places the result into a vector, and writes the vector to the destination SIMD and FP register. The count does not include the most significant bit itself.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.14 CLZ (vector)

Count Leading Zero bits (vector).

Syntax

`CLZ Vd.T, Vn.T`

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of `8B`, `16B`, `4H`, `8H`, `2S` or `4S`.

Vn
Is the name of the SIMD and FP source register.

Usage

Count Leading Zero bits (vector). This instruction counts the number of consecutive zeros, starting from the most significant bit, in each vector element in the source SIMD and FP register, places the result into a vector, and writes the vector to the destination SIMD and FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.15 CMEQ (vector, register)

Compare bitwise Equal (vector).

Syntax

`CMEQ Vd.T, Vn.T, Vm.T`

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of `8B`, `16B`, `4H`, `8H`, `2S`, `4S` or `2D`.

Vn
Is the name of the first SIMD and FP source register.

Vm
Is the name of the second SIMD and FP source register.

Usage

Compare bitwise Equal (vector). This instruction compares each vector element from the first source SIMD and FP register with the corresponding vector element from the second source SIMD and FP register, and if the comparison is equal sets every bit of the corresponding vector element in the destination SIMD and FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD and FP register to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.16 CMEQ (vector, zero)

Compare bitwise Equal to zero (vector).

Syntax

`CMEQ Vd.T, Vn.T, #0`

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of `8B`, `16B`, `4H`, `8H`, `2S`, `4S` or `2D`.

Vn
Is the name of the SIMD and FP source register.

Usage

Compare bitwise Equal to zero (vector). This instruction reads each vector element in the source SIMD and FP register and if the value is equal to zero sets every bit of the corresponding vector element in the destination SIMD and FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD and FP register to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.17 CMGE (vector, register)

Compare signed Greater than or Equal (vector).

Syntax

`CMGE` *`Vd.T, Vn.T, Vm.T`*

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of `8B`, `16B`, `4H`, `8H`, `2S`, `4S` or `2D`.

Vn
Is the name of the first SIMD and FP source register.

Vm
Is the name of the second SIMD and FP source register.

Usage

Compare signed Greater than or Equal (vector). This instruction compares each vector element in the first source SIMD and FP register with the corresponding vector element in the second source SIMD and FP register and if the first signed integer value is greater than or equal to the second signed integer value sets every bit of the corresponding vector element in the destination SIMD and FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD and FP register to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.18 CMGE (vector, zero)

Compare signed Greater than or Equal to zero (vector).

Syntax

`CMGE` *`Vd.T, Vn.T, #0`*

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of `8B`, `16B`, `4H`, `8H`, `2S`, `4S` or `2D`.

Vn
Is the name of the SIMD and FP source register.

Usage

Compare signed Greater than or Equal to zero (vector). This instruction reads each vector element in the source SIMD and FP register and if the signed integer value is greater than or equal to zero sets every bit of the corresponding vector element in the destination SIMD and FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD and FP register to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.19 CMGT (vector, register)

Compare signed Greater than (vector).

Syntax

`CMGT Vd.T, Vn.T, Vm.T`

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of `8B`, `16B`, `4H`, `8H`, `2S`, `4S` or `2D`.

Vn
Is the name of the first SIMD and FP source register.

Vm
Is the name of the second SIMD and FP source register.

Usage

Compare signed Greater than (vector). This instruction compares each vector element in the first source SIMD and FP register with the corresponding vector element in the second source SIMD and FP register and if the first signed integer value is greater than the second signed integer value sets every bit of the corresponding vector element in the destination SIMD and FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD and FP register to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.20 CMGT (vector, zero)

Compare signed Greater than zero (vector).

Syntax

`CMGT` *`Vd.T, Vn.T, #0`*

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of `8B`, `16B`, `4H`, `8H`, `2S`, `4S` or `2D`.

Vn
Is the name of the SIMD and FP source register.

Usage

Compare signed Greater than zero (vector). This instruction reads each vector element in the source SIMD and FP register and if the signed integer value is greater than zero sets every bit of the corresponding vector element in the destination SIMD and FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD and FP register to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.21 CMHI (vector, register)

Compare unsigned Higher (vector).

Syntax

`CMHI Vd.T, Vn.T, Vm.T`

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of `8B`, `16B`, `4H`, `8H`, `2S`, `4S` or `2D`.

Vn
Is the name of the first SIMD and FP source register.

Vm
Is the name of the second SIMD and FP source register.

Usage

Compare unsigned Higher (vector). This instruction compares each vector element in the first source SIMD and FP register with the corresponding vector element in the second source SIMD and FP register and if the first unsigned integer value is greater than the second unsigned integer value sets every bit of the corresponding vector element in the destination SIMD and FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD and FP register to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.22 CMHS (vector, register)

Compare unsigned Higher or Same (vector).

Syntax

`CMHS` *`Vd.T, Vn.T, Vm.T`*

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of `8B`, `16B`, `4H`, `8H`, `2S`, `4S` or `2D`.

Vn
Is the name of the first SIMD and FP source register.

Vm
Is the name of the second SIMD and FP source register.

Usage

Compare unsigned Higher or Same (vector). This instruction compares each vector element in the first source SIMD and FP register with the corresponding vector element in the second source SIMD and FP register and if the first unsigned integer value is greater than or equal to the second unsigned integer value sets every bit of the corresponding vector element in the destination SIMD and FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD and FP register to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.23 CMLE (vector, zero)

Compare signed Less than or Equal to zero (vector).

Syntax

`CMLE Vd.T, Vn.T, #0`

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of `8B`, `16B`, `4H`, `8H`, `2S`, `4S` or `2D`.

Vn
Is the name of the SIMD and FP source register.

Usage

Compare signed Less than or Equal to zero (vector). This instruction reads each vector element in the source SIMD and FP register and if the signed integer value is less than or equal to zero sets every bit of the corresponding vector element in the destination SIMD and FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD and FP register to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.24 CMLT (vector, zero)

Compare signed Less than zero (vector).

Syntax

`CMLT` *`Vd.T, Vn.T, #0`*

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of `8B`, `16B`, `4H`, `8H`, `2S`, `4S` or `2D`.

Vn
Is the name of the SIMD and FP source register.

Usage

Compare signed Less than zero (vector). This instruction reads each vector element in the source SIMD and FP register and if the signed integer value is less than zero sets every bit of the corresponding vector element in the destination SIMD and FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD and FP register to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.25 CMTST (vector)

Compare bitwise Test bits nonzero (vector).

Syntax

`CMTST` *`Vd.T, Vn.T, Vm.T`*

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of `8B`, `16B`, `4H`, `8H`, `2S`, `4S` or `2D`.

Vn
Is the name of the first SIMD and FP source register.

Vm
Is the name of the second SIMD and FP source register.

Usage

Compare bitwise Test bits nonzero (vector). This instruction reads each vector element in the first source SIMD and FP register, performs an AND with the corresponding vector element in the second source SIMD and FP register, and if the result is not zero, sets every bit of the corresponding vector element in the destination SIMD and FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD and FP register to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.26 CNT (vector)

Population Count per byte.

Syntax

`CNT Vd.T, Vn.T`

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be either `8B` or `16B`.

Vn
Is the name of the SIMD and FP source register.

Usage

Population Count per byte. This instruction counts the number of bits that have a value of one in each vector element in the source SIMD and FP register, places the result into a vector, and writes the vector to the destination SIMD and FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.27 DUP (vector, element)

vector.

Syntax

`DUP Vd.T, Vn.Ts[index]`

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of the values shown in Usage.

Ts
Is an element size specifier, and can be one of the values shown in Usage.

Vn
Is the name of the SIMD and FP source register.

index
Is the element index, in the range shown in Usage.

Usage

Duplicate vector element to vector or scalar. This instruction duplicates the vector element at the specified element index in the source SIMD and FP register into a scalar or each element in a vector, and writes the result to the destination SIMD and FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-4 DUP (Vector) specifier combinations

T	*Ts*	*index*
8B	B	0 to 15
16B	B	0 to 15
4H	H	0 to 7
8H	H	0 to 7
2S	S	0 to 3
4S	S	0 to 3
2D	D	0 or 1

Related references

19.1 A64 SIMD scalar instructions in alphabetical order on page 19-1215.

20.28 DUP (vector, general)

Duplicate general-purpose register to vector.

Syntax

`DUP Vd.T, Rn`

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of the values shown in Usage.

R
Is the width specifier for the general-purpose source register, and can be either `W` or `X`.

n
Is the number [0-30] of the general-purpose source register or ZR (31).

Usage

Duplicate general-purpose register to vector. This instruction duplicates the contents of the source general-purpose register into a scalar or each element in a vector, and writes the result to the SIMD and FP destination register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-5 DUP (Vector) specifier combinations

T	*R*
8B	W
16B	W
4H	W
8H	W
2S	W
4S	W
2D	X

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.29 EOR (vector)

Bitwise Exclusive OR (vector).

Syntax

`EOR` *`Vd.T, Vn.T, Vm.T`*

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be either `8B` or `16B`.

Vn
Is the name of the first SIMD and FP source register.

Vm
Is the name of the second SIMD and FP source register.

Usage

Bitwise Exclusive OR (vector). This instruction performs a bitwise Exclusive OR operation between the two source SIMD and FP registers, and places the result in the destination SIMD and FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.30 EXT (vector)

Extract vector from pair of vectors.

Syntax

`EXT Vd.T, Vn.T, Vm.T, #index`

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be either `8B` or `16B`.

Vn
Is the name of the first SIMD and FP source register.

Vm
Is the name of the second SIMD and FP source register.

index
Is the lowest numbered byte element to be extracted in the range shown in Usage.

Usage

Extract vector from pair of vectors. This instruction extracts the lowest vector elements from the second source SIMD and FP register and the highest vector elements from the first source SIMD and FP register, concatenates the results into a vector, and writes the vector to the destination SIMD and FP register vector. The index value specifies the lowest vector element to extract from the first source register, and consecutive elements are extracted from the first, then second, source registers until the destination vector is filled.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-6 EXT (Vector) specifier combinations

T	*index*
8B	0 to 7
16B	0 to 15

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.31 FABD (vector)

Floating-point Absolute Difference (vector).

Syntax

```
FABD Vd.T, Vn.T, Vm.T ; Vector half precision
```

```
FABD Vd.T, Vn.T, Vm.T ; Vector single-precision and double-precision
```

Where:

Vd
Is the name of the SIMD and FP destination register

T
Is an arrangement specifier:

Vector half precision
Can be one of `4H` or `8H`.

Vector single-precision and double-precision
Can be one of `2S`, `4S` or `2D`.

Vn
Is the name of the first SIMD and FP source register

Vm
Is the name of the second SIMD and FP source register

Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point Absolute Difference (vector). This instruction subtracts the floating-point values in the elements of the second source SIMD and FP register, from the corresponding floating-point values in the elements of the first source SIMD and FP register, places the absolute value of each result in a vector, and writes the vector to the destination SIMD and FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.32 FABS (vector)

Floating-point Absolute value (vector).

Syntax

```
FABS Vd.T, Vn.T ; Half-precision
```

```
FABS Vd.T, Vn.T ; Single-precision and double-precision
```

Where:

T

Is an arrangement specifier:

Half-precision

Can be one of `4H` or `8H`.

Single-precision and double-precision

Can be one of `2S`, `4S` or `2D`.

Vd

Is the name of the SIMD and FP destination register.

Vn

Is the name of the SIMD and FP source register.

Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point Absolute value (vector). This instruction calculates the absolute value of each vector element in the source SIMD and FP register, writes the result to a vector, and writes the vector to the destination SIMD and FP register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.33 FACGE (vector)

Floating-point Absolute Compare Greater than or Equal (vector).

Syntax

```
FACGE Vd.T, Vn.T, Vm.T ; Vector half precision

FACGE Vd.T, Vn.T, Vm.T ; Vector single-precision and double-precision
```

Where:

Vd
Is the name of the SIMD and FP destination register

T
Is an arrangement specifier:

Vector half precision
Can be one of `4H` or `8H`.

Vector single-precision and double-precision
Can be one of `2S`, `4S` or `2D`.

Vn
Is the name of the first SIMD and FP source register

Vm
Is the name of the second SIMD and FP source register

Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point Absolute Compare Greater than or Equal (vector). This instruction compares the absolute value of each floating-point value in the first source SIMD and FP register with the absolute value of the corresponding floating-point value in the second source SIMD and FP register and if the first value is greater than or equal to the second value sets every bit of the corresponding vector element in the destination SIMD and FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD and FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.34 FACGT (vector)

Floating-point Absolute Compare Greater than (vector).

Syntax

```
FACGT Vd.T, Vn.T, Vm.T ; Vector half precision
```

```
FACGT Vd.T, Vn.T, Vm.T ; Vector single-precision and double-precision
```

Where:

Vd
Is the name of the SIMD and FP destination register

T
Is an arrangement specifier:

Vector half precision
Can be one of `4H` or `8H`.

Vector single-precision and double-precision
Can be one of `2S`, `4S` or `2D`.

Vn
Is the name of the first SIMD and FP source register

Vm
Is the name of the second SIMD and FP source register

Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point Absolute Compare Greater than (vector). This instruction compares the absolute value of each vector element in the first source SIMD and FP register with the absolute value of the corresponding vector element in the second source SIMD and FP register and if the first value is greater than the second value sets every bit of the corresponding vector element in the destination SIMD and FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD and FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.35 FADD (vector)

Floating-point Add (vector).

Syntax

```
FADD Vd.T, Vn.T, Vm.T ; Half-precision
```

```
FADD Vd.T, Vn.T, Vm.T ; Single-precision and double-precision
```

Where:

T

Is an arrangement specifier:

Half-precision

Can be one of `4H` or `8H`.

Single-precision and double-precision

Can be one of `2S`, `4S` or `2D`.

Vd

Is the name of the SIMD and FP destination register.

Vn

Is the name of the first SIMD and FP source register.

Vm

Is the name of the second SIMD and FP source register.

Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point Add (vector). This instruction adds corresponding vector elements in the two source SIMD and FP registers, writes the result into a vector, and writes the vector to the destination SIMD and FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.36 FADDP (vector)

Floating-point Add Pairwise (vector).

Syntax

```
FADDP Vd.T, Vn.T, Vm.T ; Half-precision

FADDP Vd.T, Vn.T, Vm.T ; Single-precision and double-precision
```

Where:

T

Is an arrangement specifier:

Half-precision

Can be one of `4H` or `8H`.

Single-precision and double-precision

Can be one of `2S`, `4S` or `2D`.

Vd

Is the name of the SIMD and FP destination register.

Vn

Is the name of the first SIMD and FP source register.

Vm

Is the name of the second SIMD and FP source register.

Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point Add Pairwise (vector). This instruction creates a vector by concatenating the vector elements of the first source SIMD and FP register after the vector elements of the second source SIMD and FP register, reads each pair of adjacent vector elements from the concatenated vector, adds each pair of values together, places the result into a vector, and writes the vector to the destination SIMD and FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.37 FCADD (vector)

Floating-point Complex Add.

Syntax

`FCADD Vd.T, Vn.T, Vm.T, #rotate`

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of `4H`, `8H`, `2S`, `4S` or `2D`.

Vn
Is the name of the first SIMD and FP source register.

Vm
Is the name of the second SIMD and FP source register.

rotate
Is the rotation, and can be either `90` or `270`.

Architectures supported (vector)

Supported in the Armv8.3-A architecture and later.

Usage

Floating-point Complex Add.

This instruction adds two source complex numbers from the *Vm* and the *Vn* vector registers and places the resulting complex number in the destination *Vd* vector register. The number of complex numbers that can be stored in the *Vm*, the *Vn*, and the *Vd* registers is calculated as the vector register size divided by the length of each complex number. These lengths are 16 for half-precision, 32 for single-precision, and 64 for double-precision. Each complex number is represented in a SIMP&FP register as a pair of elements with the imaginary part of the number being placed in the more significant element, and the real part of the number being placed in the less significant element. Both real and imaginary parts of the source and the resulting complex number are represented as floating-point values.

One of the two vector elements that are read from each of the numbers in the *Vm* source SIMD and FP register can be optionally negated based on the rotation value:

- If the rotation is 90, the odd-numbered vector elements are negated.
- If the rotation is 270, the even-numbered vector elements are negated.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.38 FCMEQ (vector, register)

Floating-point Compare Equal (vector).

Syntax

`FCMEQ Vd.T, Vn.T, Vm.T`

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier:

Vector half precision
Can be one of `4H` or `8H`.

Vector single-precision and double-precision
Can be one of `2S`, `4S` or `2D`.

Vn
Is the name of the first SIMD and FP source register.

Vm
Is the name of the second SIMD and FP source register.

Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point Compare Equal (vector). This instruction compares each floating-point value from the first source SIMD and FP register, with the corresponding floating-point value from the second source SIMD and FP register, and if the comparison is equal sets every bit of the corresponding vector element in the destination SIMD and FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD and FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.39 FCMEQ (vector, zero)

Floating-point Compare Equal to zero (vector).

Syntax

`FCMEQ Vd.T, Vn.T, #0.0`

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier:

Vector half precision
Can be one of `4H` or `8H`.

Vector single-precision and double-precision
Can be one of `2S`, `4S` or `2D`.

Vn
Is the name of the SIMD and FP source register.

Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point Compare Equal to zero (vector). This instruction reads each floating-point value in the source SIMD and FP register and if the value is equal to zero sets every bit of the corresponding vector element in the destination SIMD and FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD and FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.40 FCMGE (vector, register)

Floating-point Compare Greater than or Equal (vector).

Syntax

`FCMGE Vd.T, Vn.T, Vm.T`

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier:

Vector half precision
Can be one of `4H` or `8H`.

Vector single-precision and double-precision
Can be one of `2S`, `4S` or `2D`.

Vn
Is the name of the first SIMD and FP source register.

Vm
Is the name of the second SIMD and FP source register.

Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point Compare Greater than or Equal (vector). This instruction reads each floating-point value in the first source SIMD and FP register and if the value is greater than or equal to the corresponding floating-point value in the second source SIMD and FP register sets every bit of the corresponding vector element in the destination SIMD and FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD and FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.41 FCMGE (vector, zero)

Floating-point Compare Greater than or Equal to zero (vector).

Syntax

```
FCMGE Vd.T, Vn.T, #0.0
```

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier:

Vector half precision
Can be one of `4H` or `8H`.

Vector single-precision and double-precision
Can be one of `2S`, `4S` or `2D`.

Vn
Is the name of the SIMD and FP source register.

Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point Compare Greater than or Equal to zero (vector). This instruction reads each floating-point value in the source SIMD and FP register and if the value is greater than or equal to zero sets every bit of the corresponding vector element in the destination SIMD and FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD and FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.42 FCMGT (vector, register)

Floating-point Compare Greater than (vector).

Syntax

`FCMGT Vd.T, Vn.T, Vm.T`

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier:

Vector half precision
Can be one of `4H` or `8H`.

Vector single-precision and double-precision
Can be one of `2S`, `4S` or `2D`.

Vn
Is the name of the first SIMD and FP source register.

Vm
Is the name of the second SIMD and FP source register.

Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point Compare Greater than (vector). This instruction reads each floating-point value in the first source SIMD and FP register and if the value is greater than the corresponding floating-point value in the second source SIMD and FP register sets every bit of the corresponding vector element in the destination SIMD and FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD and FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.43 FCMGT (vector, zero)

Floating-point Compare Greater than zero (vector).

Syntax

```
FCMGT Vd.T, Vn.T, #0.0
```

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier:

Vector half precision
Can be one of `4H` or `8H`.

Vector single-precision and double-precision
Can be one of `2S`, `4S` or `2D`.

Vn
Is the name of the SIMD and FP source register.

Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point Compare Greater than zero (vector). This instruction reads each floating-point value in the source SIMD and FP register and if the value is greater than zero sets every bit of the corresponding vector element in the destination SIMD and FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD and FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.44 FCMLA (vector)

Floating-point Complex Multiply Accumulate.

Syntax

```
FCMLA Vd.T, Vn.T, Vm.T, #rotate
```

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of `4H`, `8H`, `2S`, `4S` or `2D`.

Vn
Is the name of the first SIMD and FP source register.

Vm
Is the name of the second SIMD and FP source register.

rotate
Is the rotation, and can be one of `0`, `90`, `180` or `270`.

Architectures supported (vector)

Supported in the Armv8.3-A architecture and later.

Usage

This instruction multiplies the two source complex numbers from the *Vm* and the *Vn* vector registers and adds the result to the corresponding complex number in the destination *Vd* vector register. The number of complex numbers that can be stored in the *Vm*, the *Vn*, and the *Vd* registers is calculated as the vector register size divided by the length of each complex number. These lengths are 16 for half-precision, 32 for single-precision, and 64 for double-precision. Each complex number is represented in a SIMP&FP register as a pair of elements with the imaginary part of the number being placed in the more significant element, and the real part of the number being placed in the less significant element. Both real and imaginary parts of the source and the resulting complex number are represented as floating-point values.

None, one, or both of the two vector elements that are read from each of the numbers in the *Vm* source SIMD and FP register can be negated based on the rotation value:

- If the rotation is 0, none of the vector elements are negated.
- If the rotation is 90, the odd-numbered vector elements are negated.
- If the rotation is 180, both vector elements are negated.
- If the rotation is 270, the even-numbered vector elements are negated.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.45 FCMLE (vector, zero)

Floating-point Compare Less than or Equal to zero (vector).

Syntax

```
FCMLE Vd.T, Vn.T, #0.0 ; Vector half precision
```

```
FCMLE Vd.T, Vn.T, #0.0 ; Vector single-precision and double-precision
```

Where:

Vd
Is the name of the SIMD and FP destination register

T
Is an arrangement specifier:

Vector half precision
Can be one of `4H` or `8H`.

Vector single-precision and double-precision
Can be one of `2S`, `4S` or `2D`.

Vn
Is the name of the SIMD and FP source register

Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point Compare Less than or Equal to zero (vector). This instruction reads each floating-point value in the source SIMD and FP register and if the value is less than or equal to zero sets every bit of the corresponding vector element in the destination SIMD and FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD and FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.46 FCMLT (vector, zero)

Floating-point Compare Less than zero (vector).

Syntax

```
FCMLT Vd.T, Vn.T, #0.0 ; Vector half precision
FCMLT Vd.T, Vn.T, #0.0 ; Vector single-precision and double-precision
```

Where:

Vd
Is the name of the SIMD and FP destination register

T
Is an arrangement specifier:

Vector half precision
Can be one of `4H` or `8H`.

Vector single-precision and double-precision
Can be one of `2S`, `4S` or `2D`.

Vn
Is the name of the SIMD and FP source register

Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point Compare Less than zero (vector). This instruction reads each floating-point value in the source SIMD and FP register and if the value is less than zero sets every bit of the corresponding vector element in the destination SIMD and FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD and FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.47 FCVTAS (vector)

Floating-point Convert to Signed integer, rounding to nearest with ties to Away (vector).

Syntax

```
FCVTAS Vd.T, Vn.T ; Vector half precision
```

```
FCVTAS Vd.T, Vn.T ; Vector single-precision and double-precision
```

Where:

Vd
Is the name of the SIMD and FP destination register

T
Is an arrangement specifier:

Vector half precision
Can be one of `4H` or `8H`.

Vector single-precision and double-precision
Can be one of `2S`, `4S` or `2D`.

Vn
Is the name of the SIMD and FP source register

Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point Convert to Signed integer, rounding to nearest with ties to Away (vector). This instruction converts each element in a vector from a floating-point value to a signed integer value using the Round to Nearest with Ties to Away rounding mode and writes the result to the SIMD and FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.48 FCVTAU (vector)

Floating-point Convert to Unsigned integer, rounding to nearest with ties to Away (vector).

Syntax

```
FCVTAU Vd.T, Vn.T ; Vector half precision
```

```
FCVTAU Vd.T, Vn.T ; Vector single-precision and double-precision
```

Where:

Vd
Is the name of the SIMD and FP destination register

T
Is an arrangement specifier:

Vector half precision
Can be one of `4H` or `8H`.

Vector single-precision and double-precision
Can be one of `2S`, `4S` or `2D`.

Vn
Is the name of the SIMD and FP source register

Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point Convert to Unsigned integer, rounding to nearest with ties to Away (vector). This instruction converts each element in a vector from a floating-point value to an unsigned integer value using the Round to Nearest with Ties to Away rounding mode and writes the result to the SIMD and FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.49 FCVTL, FCVTL2 (vector)

Floating-point Convert to higher precision Long (vector).

Syntax

`FCVTL{2} Vd.Ta, Vn.Tb`

Where:

2
Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements. See <Q> in the Usage table.

Vd
Is the name of the SIMD and FP destination register.

Ta
Is an arrangement specifier, and can be either `4S` or `2D`.

Vn
Is the name of the SIMD and FP source register.

Tb
Is an arrangement specifier, and can be one of the values shown in Usage.

Usage

Floating-point Convert to higher precision Long (vector). This instruction reads each element in a vector in the SIMD and FP source register, converts each value to double the precision of the source element using the rounding mode that is determined by the FPCR, and writes each result to the equivalent element of the vector in the SIMD and FP destination register.

Where the operation lengthens a 64-bit vector to a 128-bit vector, the `FCVTL2` variant operates on the elements in the top 64 bits of the source register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-7 FCVTL, FCVTL2 (Vector) specifier combinations

<Q>	*Ta*	*Tb*
-	4S	4H
2	4S	8H
-	2D	2S
2	2D	4S

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.50 FCVTMS (vector)

Floating-point Convert to Signed integer, rounding toward Minus infinity (vector).

Syntax

```
FCVTMS Vd.T, Vn.T ; Vector half precision
```

```
FCVTMS Vd.T, Vn.T ; Vector single-precision and double-precision
```

Where:

Vd
Is the name of the SIMD and FP destination register

T
Is an arrangement specifier:

Vector half precision
Can be one of `4H` or `8H`.

Vector single-precision and double-precision
Can be one of `2S`, `4S` or `2D`.

Vn
Is the name of the SIMD and FP source register

Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point Convert to Signed integer, rounding toward Minus infinity (vector). This instruction converts a scalar or each element in a vector from a floating-point value to a signed integer value using the Round towards Minus Infinity rounding mode, and writes the result to the SIMD and FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.51 FCVTMU (vector)

Floating-point Convert to Unsigned integer, rounding toward Minus infinity (vector).

Syntax

```
FCVTMU Vd.T, Vn.T ; Vector half precision

FCVTMU Vd.T, Vn.T ; Vector single-precision and double-precision
```

Where:

Vd
Is the name of the SIMD and FP destination register

T
Is an arrangement specifier:

Vector half precision
Can be one of `4H` or `8H`.

Vector single-precision and double-precision
Can be one of `2S`, `4S` or `2D`.

Vn
Is the name of the SIMD and FP source register

Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point Convert to Unsigned integer, rounding toward Minus infinity (vector). This instruction converts a scalar or each element in a vector from a floating-point value to an unsigned integer value using the Round towards Minus Infinity rounding mode, and writes the result to the SIMD and FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.52 FCVTN, FCVTN2 (vector)

Floating-point Convert to lower precision Narrow (vector).

Syntax

`FCVTN{2} Vd.Tb, Vn.Ta`

Where:

2
Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements. See <Q> in the Usage table.

Vd
Is the name of the SIMD and FP destination register.

Tb
Is an arrangement specifier, and can be one of the values shown in Usage.

Vn
Is the name of the SIMD and FP source register.

Ta
Is an arrangement specifier, and can be either `4S` or `2D`.

Usage

Floating-point Convert to lower precision Narrow (vector). This instruction reads each vector element in the SIMD and FP source register, converts each result to half the precision of the source element, writes the final result to a vector, and writes the vector to the lower or upper half of the destination SIMD and FP register. The destination vector elements are half as long as the source vector elements. The rounding mode is determined by the FPCR.

The `FCVTN` instruction writes the vector to the lower half of the destination register and clears the upper half, while the `FCVTN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-8 FCVTN, FCVTN2 (Vector) specifier combinations

<Q>	*Tb*	*Ta*
-	4H	4S
2	8H	4S
-	2S	2D
2	4S	2D

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.53 FCVTNS (vector)

Floating-point Convert to Signed integer, rounding to nearest with ties to even (vector).

Syntax

```
FCVTNS Vd.T, Vn.T ; Vector half precision
```

```
FCVTNS Vd.T, Vn.T ; Vector single-precision and double-precision
```

Where:

Vd
Is the name of the SIMD and FP destination register

T
Is an arrangement specifier:

Vector half precision
Can be one of `4H` or `8H`.

Vector single-precision and double-precision
Can be one of `2S`, `4S` or `2D`.

Vn
Is the name of the SIMD and FP source register

Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point Convert to Signed integer, rounding to nearest with ties to even (vector). This instruction converts a scalar or each element in a vector from a floating-point value to a signed integer value using the Round to Nearest rounding mode, and writes the result to the SIMD and FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.54 FCVTNU (vector)

Floating-point Convert to Unsigned integer, rounding to nearest with ties to even (vector).

Syntax

```
FCVTNU Vd.T, Vn.T ; Vector half precision
```

```
FCVTNU Vd.T, Vn.T ; Vector single-precision and double-precision
```

Where:

Vd
Is the name of the SIMD and FP destination register

T
Is an arrangement specifier:

Vector half precision
Can be one of `4H` or `8H`.

Vector single-precision and double-precision
Can be one of `2S`, `4S` or `2D`.

Vn
Is the name of the SIMD and FP source register

Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point Convert to Unsigned integer, rounding to nearest with ties to even (vector). This instruction converts a scalar or each element in a vector from a floating-point value to an unsigned integer value using the Round to Nearest rounding mode, and writes the result to the SIMD and FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.55 FCVTPS (vector)

Floating-point Convert to Signed integer, rounding toward Plus infinity (vector).

Syntax

```
FCVTPS Vd.T, Vn.T ; Vector half precision
```

```
FCVTPS Vd.T, Vn.T ; Vector single-precision and double-precision
```

Where:

Vd
Is the name of the SIMD and FP destination register

T
Is an arrangement specifier:

Vector half precision
Can be one of `4H` or `8H`.

Vector single-precision and double-precision
Can be one of `2S`, `4S` or `2D`.

Vn
Is the name of the SIMD and FP source register

Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point Convert to Signed integer, rounding toward Plus infinity (vector). This instruction converts a scalar or each element in a vector from a floating-point value to a signed integer value using the Round towards Plus Infinity rounding mode, and writes the result to the SIMD and FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.56 FCVTPU (vector)

Floating-point Convert to Unsigned integer, rounding toward Plus infinity (vector).

Syntax

```
FCVTPU Vd.T, Vn.T ; Vector half precision
```

```
FCVTPU Vd.T, Vn.T ; Vector single-precision and double-precision
```

Where:

Vd
Is the name of the SIMD and FP destination register

T
Is an arrangement specifier:

Vector half precision
Can be one of `4H` or `8H`.

Vector single-precision and double-precision
Can be one of `2S`, `4S` or `2D`.

Vn
Is the name of the SIMD and FP source register

Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point Convert to Unsigned integer, rounding toward Plus infinity (vector). This instruction converts a scalar or each element in a vector from a floating-point value to an unsigned integer value using the Round towards Plus Infinity rounding mode, and writes the result to the SIMD and FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.57 FCVTXN, FCVTXN2 (vector)

Floating-point Convert to lower precision Narrow, rounding to odd (vector).

Syntax

`FCVTXN{2} Vd.Tb, Vn.Ta`

Where:

2
Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements. See <Q> in the Usage table.

Vd
Is the name of the SIMD and FP destination register.

Tb
Is an arrangement specifier, and can be either `2S` or `4S`.

Vn
Is the name of the SIMD and FP source register.

Ta
Is an arrangement specifier, `2D`.

Usage

Floating-point Convert to lower precision Narrow, rounding to odd (vector). This instruction reads each vector element in the source SIMD and FP register, narrows each value to half the precision of the source element using the Round to Odd rounding mode, writes the result to a vector, and writes the vector to the destination SIMD and FP register.

The `FCVTXN` instruction writes the vector to the lower half of the destination register and clears the upper half, while the `FCVTXN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-9 FCVTXN{2} (Vector) specifier combinations

<Q>	*Tb*	*Ta*
-	2S	2D
2	4S	2D

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.58 FCVTZS (vector, fixed-point)

Floating-point Convert to Signed fixed-point, rounding toward Zero (vector).

Syntax

`FCVTZS Vd.T, Vn.T, #fbits`

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of the values shown in Usage.

Vn
Is the name of the SIMD and FP source register.

fbits
Is the number of fractional bits, in the range 1 to the element width.

Usage

Floating-point Convert to Signed fixed-point, rounding toward Zero (vector). This instruction converts a scalar or each element in a vector from floating-point to fixed-point signed integer using the Round towards Zero rounding mode, and writes the result to the SIMD and FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-10 FCVTZS (Vector) specifier combinations

T	*fbits*
4H	
8H	
2S	1 to 32
4S	1 to 32
2D	1 to 64

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.59 FCVTZS (vector, integer)

Floating-point Convert to Signed integer, rounding toward Zero (vector).

Syntax

```
FCVTZS Vd.T, Vn.T ; Vector half precision
```

```
FCVTZS Vd.T, Vn.T ; Vector single-precision and double-precision
```

Where:

Vd
Is the name of the SIMD and FP destination register

T
Is an arrangement specifier:

Vector half precision
Can be one of `4H` or `8H`.

Vector single-precision and double-precision
Can be one of `2S`, `4S` or `2D`.

Vn
Is the name of the SIMD and FP source register

Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point Convert to Signed integer, rounding toward Zero (vector). This instruction converts a scalar or each element in a vector from a floating-point value to a signed integer value using the Round towards Zero rounding mode, and writes the result to the SIMD and FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.60 FCVTZU (vector, fixed-point)

Floating-point Convert to Unsigned fixed-point, rounding toward Zero (vector).

Syntax

```
FCVTZU Vd.T, Vn.T, #fbits
```

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of the values shown in Usage.

Vn
Is the name of the SIMD and FP source register.

fbits
Is the number of fractional bits, in the range 1 to the element width.

Usage

Floating-point Convert to Unsigned fixed-point, rounding toward Zero (vector). This instruction converts a scalar or each element in a vector from floating-point to fixed-point unsigned integer using the Round towards Zero rounding mode, and writes the result to the general-purpose destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-11 FCVTZU (Vector) specifier combinations

T	*fbits*
4H	
8H	
2S	1 to 32
4S	1 to 32
2D	1 to 64

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.61 FCVTZU (vector, integer)

Floating-point Convert to Unsigned integer, rounding toward Zero (vector).

Syntax

```
FCVTZU Vd.T, Vn.T ; Vector half precision

FCVTZU Vd.T, Vn.T ; Vector single-precision and double-precision
```

Where:

Vd
Is the name of the SIMD and FP destination register

T
Is an arrangement specifier:

Vector half precision
Can be one of `4H` or `8H`.

Vector single-precision and double-precision
Can be one of `2S`, `4S` or `2D`.

Vn
Is the name of the SIMD and FP source register

Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point Convert to Unsigned integer, rounding toward Zero (vector). This instruction converts a scalar or each element in a vector from a floating-point value to an unsigned integer value using the Round towards Zero rounding mode, and writes the result to the SIMD and FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.62 FDIV (vector)

Floating-point Divide (vector).

Syntax

```
FDIV Vd.T, Vn.T, Vm.T ; Half-precision
```

```
FDIV Vd.T, Vn.T, Vm.T ; Single-precision and double-precision
```

Where:

T

Is an arrangement specifier:

Half-precision

Can be one of `4H` or `8H`.

Single-precision and double-precision

Can be one of `2S`, `4S` or `2D`.

Vd

Is the name of the SIMD and FP destination register.

Vn

Is the name of the first SIMD and FP source register.

Vm

Is the name of the second SIMD and FP source register.

Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point Divide (vector). This instruction divides the floating-point values in the elements in the first source SIMD and FP register, by the floating-point values in the corresponding elements in the second source SIMD and FP register, places the results in a vector, and writes the vector to the destination SIMD and FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.63 FMAX (vector)

Floating-point Maximum (vector).

Syntax

```
FMAX Vd.T, Vn.T, Vm.T ; Half-precision
```

```
FMAX Vd.T, Vn.T, Vm.T ; Single-precision and double-precision
```

Where:

T

Is an arrangement specifier:

Half-precision

Can be one of `4H` or `8H`.

Single-precision and double-precision

Can be one of `2S`, `4S` or `2D`.

Vd

Is the name of the SIMD and FP destination register.

Vn

Is the name of the first SIMD and FP source register.

Vm

Is the name of the second SIMD and FP source register.

Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point Maximum (vector). This instruction compares corresponding vector elements in the two source SIMD and FP registers, places the larger of each of the two floating-point values into a vector, and writes the vector to the destination SIMD and FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.64 FMAXNM (vector)

Floating-point Maximum Number (vector).

Syntax

```
FMAXNM Vd.T, Vn.T, Vm.T ; Half-precision
```

```
FMAXNM Vd.T, Vn.T, Vm.T ; Single-precision and double-precision
```

Where:

T

Is an arrangement specifier:

Half-precision

Can be one of `4H` or `8H`.

Single-precision and double-precision

Can be one of `2S`, `4S` or `2D`.

Vd

Is the name of the SIMD and FP destination register.

Vn

Is the name of the first SIMD and FP source register.

Vm

Is the name of the second SIMD and FP source register.

Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point Maximum Number (vector). This instruction compares corresponding vector elements in the two source SIMD and FP registers, writes the larger of the two floating-point values into a vector, and writes the vector to the destination SIMD and FP register.

NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result placed in the vector is the numerical value, otherwise the result is identical to *FMAX (scalar)*.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.65 FMAXNMP (vector)

Floating-point Maximum Number Pairwise (vector).

Syntax

```
FMAXNMP Vd.T, Vn.T, Vm.T ; Half-precision

FMAXNMP Vd.T, Vn.T, Vm.T ; Single-precision and double-precision
```

Where:

T

Is an arrangement specifier:

Half-precision

Can be one of `4H` or `8H`.

Single-precision and double-precision

Can be one of `2S`, `4S` or `2D`.

Vd

Is the name of the SIMD and FP destination register.

Vn

Is the name of the first SIMD and FP source register.

Vm

Is the name of the second SIMD and FP source register.

Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point Maximum Number Pairwise (vector). This instruction creates a vector by concatenating the vector elements of the first source SIMD and FP register after the vector elements of the second source SIMD and FP register, reads each pair of adjacent vector elements in the two source SIMD and FP registers, writes the largest of each pair of values into a vector, and writes the vector to the destination SIMD and FP register. All the values in this instruction are floating-point values.

NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result is the numerical value, otherwise the result is identical to *FMAX (scalar)*.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.66 FMAXNMV (vector)

Floating-point Maximum Number across Vector.

Syntax

```
FMAXNMV Vd, Vn.T ; Half-precision

FMAXNMV Vd, Vn.T ; Single-precision and double-precision
```

Where:

V

Is the destination width specifier:

Single-precision and double-precision
Must be `S`.

Half-precision
Must be `H`.

T

Is an arrangement specifier:

Half-precision
Can be one of `4H` or `8H`.

Single-precision and double-precision
Must be `4S`.

d

Is the number of the SIMD and FP destination register.

Vn

Is the name of the SIMD and FP source register.

Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point Maximum Number across Vector. This instruction compares all the vector elements in the source SIMD and FP register, and writes the largest of the values as a scalar to the destination SIMD and FP register. All the values in this instruction are floating-point values.

NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result of the comparison is the numerical value, otherwise the result is identical to *FMAX (scalar)*.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.67 FMAXP (vector)

Floating-point Maximum Pairwise (vector).

Syntax

```
FMAXP Vd.T, Vn.T, Vm.T ; Half-precision

FMAXP Vd.T, Vn.T, Vm.T ; Single-precision and double-precision
```

Where:

T

Is an arrangement specifier:

Half-precision

Can be one of `4H` or `8H`.

Single-precision and double-precision

Can be one of `2S`, `4S` or `2D`.

Vd

Is the name of the SIMD and FP destination register.

Vn

Is the name of the first SIMD and FP source register.

Vm

Is the name of the second SIMD and FP source register.

Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point Maximum Pairwise (vector). This instruction creates a vector by concatenating the vector elements of the first source SIMD and FP register after the vector elements of the second source SIMD and FP register, reads each pair of adjacent vector elements from the concatenated vector, writes the larger of each pair of values into a vector, and writes the vector to the destination SIMD and FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.68 FMAXV (vector)

Floating-point Maximum across Vector.

Syntax

```
FMAXV Vd, Vn.T ; Half-precision

FMAXV Vd, Vn.T ; Single-precision and double-precision
```

Where:

V

Is the destination width specifier:

Single-precision and double-precision
Must be `S`.

Half-precision
Must be `H`.

T

Is an arrangement specifier:

Half-precision
Can be one of `4H` or `8H`.

Single-precision and double-precision
Must be `4S`.

d

Is the number of the SIMD and FP destination register.

Vn

Is the name of the SIMD and FP source register.

Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point Maximum across Vector. This instruction compares all the vector elements in the source SIMD and FP register, and writes the largest of the values as a scalar to the destination SIMD and FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.69 FMIN (vector)

Floating-point minimum (vector).

Syntax

```
FMIN Vd.T, Vn.T, Vm.T ; Half-precision
```

```
FMIN Vd.T, Vn.T, Vm.T ; Single-precision and double-precision
```

Where:

T

Is an arrangement specifier:

Half-precision

Can be one of `4H` or `8H`.

Single-precision and double-precision

Can be one of `2S`, `4S` or `2D`.

Vd

Is the name of the SIMD and FP destination register.

Vn

Is the name of the first SIMD and FP source register.

Vm

Is the name of the second SIMD and FP source register.

Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point minimum (vector). This instruction compares corresponding elements in the vectors in the two source SIMD and FP registers, places the smaller of each of the two floating-point values into a vector, and writes the vector to the destination SIMD and FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.70 FMINNM (vector)

Floating-point Minimum Number (vector).

Syntax

```
FMINNM Vd.T, Vn.T, Vm.T ; Half-precision

FMINNM Vd.T, Vn.T, Vm.T ; Single-precision and double-precision
```

Where:

T

Is an arrangement specifier:

Half-precision

Can be one of `4H` or `8H`.

Single-precision and double-precision

Can be one of `2S`, `4S` or `2D`.

Vd

Is the name of the SIMD and FP destination register.

Vn

Is the name of the first SIMD and FP source register.

Vm

Is the name of the second SIMD and FP source register.

Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point Minimum Number (vector). This instruction compares corresponding vector elements in the two source SIMD and FP registers, writes the smaller of the two floating-point values into a vector, and writes the vector to the destination SIMD and FP register.

NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result placed in the vector is the numerical value, otherwise the result is identical to *FMIN (scalar)*.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.71 FMINNMP (vector)

Floating-point Minimum Number Pairwise (vector).

Syntax

```
FMINNMP Vd.T, Vn.T, Vm.T ; Half-precision

FMINNMP Vd.T, Vn.T, Vm.T ; Single-precision and double-precision
```

Where:

T

Is an arrangement specifier:

Half-precision

Can be one of `4H` or `8H`.

Single-precision and double-precision

Can be one of `2S`, `4S` or `2D`.

Vd

Is the name of the SIMD and FP destination register.

Vn

Is the name of the first SIMD and FP source register.

Vm

Is the name of the second SIMD and FP source register.

Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point Minimum Number Pairwise (vector). This instruction creates a vector by concatenating the vector elements of the first source SIMD and FP register after the vector elements of the second source SIMD and FP register, reads each pair of adjacent vector elements in the two source SIMD and FP registers, writes the smallest of each pair of floating-point values into a vector, and writes the vector to the destination SIMD and FP register. All the values in this instruction are floating-point values.

NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result is the numerical value, otherwise the result is identical to *FMIN (scalar)*.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.72 FMINNMV (vector)

Floating-point Minimum Number across Vector.

Syntax

```
FMINNMV Vd, Vn.T ; Half-precision

FMINNMV Vd, Vn.T ; Single-precision and double-precision
```

Where:

V

Is the destination width specifier:

Single-precision and double-precision
Must be `S`.

Half-precision
Must be `H`.

T

Is an arrangement specifier:

Half-precision
Can be one of `4H` or `8H`.

Single-precision and double-precision
Must be `4S`.

d

Is the number of the SIMD and FP destination register.

Vn

Is the name of the SIMD and FP source register.

Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point Minimum Number across Vector. This instruction compares all the vector elements in the source SIMD and FP register, and writes the smallest of the values as a scalar to the destination SIMD and FP register. All the values in this instruction are floating-point values.

NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result of the comparison is the numerical value, otherwise the result is identical to *FMIN (scalar)*.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.73 FMINP (vector)

Floating-point Minimum Pairwise (vector).

Syntax

```
FMINP Vd.T, Vn.T, Vm.T ; Half-precision

FMINP Vd.T, Vn.T, Vm.T ; Single-precision and double-precision
```

Where:

T

Is an arrangement specifier:

Half-precision

Can be one of `4H` or `8H`.

Single-precision and double-precision

Can be one of `2S`, `4S` or `2D`.

Vd

Is the name of the SIMD and FP destination register.

Vn

Is the name of the first SIMD and FP source register.

Vm

Is the name of the second SIMD and FP source register.

Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point Minimum Pairwise (vector). This instruction creates a vector by concatenating the vector elements of the first source SIMD and FP register after the vector elements of the second source SIMD and FP register, reads each pair of adjacent vector elements from the concatenated vector, writes the smaller of each pair of values into a vector, and writes the vector to the destination SIMD and FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.74 FMINV (vector)

Floating-point Minimum across Vector.

Syntax

```
FMINV Vd, Vn.T ; Half-precision

FMINV Vd, Vn.T ; Single-precision and double-precision
```

Where:

V

Is the destination width specifier:

Single-precision and double-precision
Must be `S`.

Half-precision
Must be `H`.

T

Is an arrangement specifier:

Half-precision
Can be one of `4H` or `8H`.

Single-precision and double-precision
Must be `4S`.

d

Is the number of the SIMD and FP destination register.

Vn

Is the name of the SIMD and FP source register.

Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point Minimum across Vector. This instruction compares all the vector elements in the source SIMD and FP register, and writes the smallest of the values as a scalar to the destination SIMD and FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.75 FMLA (vector, by element)

Floating-point fused Multiply-Add to accumulator (by element).

Syntax

`FMLA` *`Vd.T, Vn.T, Vm.Ts`*`[`*`index`*`]`

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier:

Vector, half-precision
Can be one of `4H` or `8H`.

Vector, single-precision and double-precision
Can be one of `2S`, `4S` or `2D`.

Vn
Is the name of the first SIMD and FP source register.

Ts
Is an element size specifier:

Vector, half-precision
Must be `H`.

Vector, single-precision and double-precision
Can be one of `S` or `D`.

index
Is the element index:

Vector, half-precision
Must be `H:L:M`.

Vector, single-precision and double-precision
Can be one of `H:L` or `H`.

Vm
Is the name of the second SIMD and FP source register in the range 0 to 31.

Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point fused Multiply-Add to accumulator (by element). 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 results in the vector elements of the destination SIMD and FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-12 FMLA (Vector, single-precision and double-precision) specifier combinations

T	*Ts*	*index*
2S	S	0 to 3
4S	S	0 to 3
2D	D	0 or 1

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.76 FMLA (vector)

Floating-point fused Multiply-Add to accumulator (vector).

Syntax

`FMLA Vd.T, Vn.T, Vm.T`

Where:

T
Is an arrangement specifier:

Half-precision
Can be one of `4H` or `8H`.

Single-precision and double-precision
Can be one of `2S`, `4S` or `2D`.

Vd
Is the name of the SIMD and FP destination register.

Vn
Is the name of the first SIMD and FP source register.

Vm
Is the name of the second SIMD and FP source register.

Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point fused Multiply-Add to accumulator (vector). This instruction multiplies corresponding floating-point values in the vectors in the two source SIMD and FP registers, adds the product to the corresponding vector element of the destination SIMD and FP register, and writes the result to the destination SIMD and FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.77 FMLS (vector, by element)

Floating-point fused Multiply-Subtract from accumulator (by element).

Syntax

`FMLS` *`Vd.T, Vn.T, Vm.Ts`*`[`*`index`*`]`

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier:

Vector, half-precision
Can be one of `4H` or `8H`.

Vector, single-precision and double-precision
Can be one of `2S`, `4S` or `2D`.

Vn
Is the name of the first SIMD and FP source register.

Ts
Is an element size specifier:

Vector, half-precision
Must be `H`.

Vector, single-precision and double-precision
Can be one of `S` or `D`.

index
Is the element index:

Vector, half-precision
Must be `H:L:M`.

Vector, single-precision and double-precision
Can be one of `H:L` or `H`.

Vm
Is the name of the second SIMD and FP source register in the range 0 to 31.

Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point fused Multiply-Subtract from accumulator (by element). 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 subtracts the results from the vector elements of the destination SIMD and FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-13 FMLS (Vector, single-precision and double-precision) specifier combinations

T	*Ts*	*index*
2S	S	0 to 3
4S	S	0 to 3
2D	D	0 or 1

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.78 FMLS (vector)

Floating-point fused Multiply-Subtract from accumulator (vector).

Syntax

`FMLS Vd.T, Vn.T, Vm.T`

Where:

T
Is an arrangement specifier:

Half-precision
Can be one of `4H` or `8H`.

Single-precision and double-precision
Can be one of `2S`, `4S` or `2D`.

Vd
Is the name of the SIMD and FP destination register.

Vn
Is the name of the first SIMD and FP source register.

Vm
Is the name of the second SIMD and FP source register.

Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point fused Multiply-Subtract from accumulator (vector). This instruction multiplies corresponding floating-point values in the vectors in the two source SIMD and FP registers, negates the product, adds the result to the corresponding vector element of the destination SIMD and FP register, and writes the result to the destination SIMD and FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.79 FMOV (vector, immediate)

Floating-point move immediate (vector).

Syntax

```
FMOV Vd.T, #imm ; Half-precision

FMOV Vd.T, #imm ; Single-precision

FMOV Vd.2D, #imm ; Double-precision
```

Where:

Vd

The value depends on the instruction variant:

Half-precision
Is the name of the SIMD and FP destination register

Single-precision
Is the name of the SIMD and FP destination register

Double-precision
Is the name of the SIMD and FP destination register

T

Is an arrangement specifier:

Half-precision
Can be one of `4H` or `8H`.

Single-precision
Can be one of `2S` or `4S`.

imm

The value depends on the instruction variant:

Half-precision
Is a signed floating-point constant with 3-bit exponent and normalized 4 bits of precision. For details of the range of constants available and the encoding of *imm*, see *Modified immediate constants in A64 floating-point instructions* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Single-precision
Is a signed floating-point constant with 3-bit exponent and normalized 4 bits of precision. For details of the range of constants available and the encoding of *imm*, see *Modified immediate constants in A64 floating-point instructions* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Double-precision
Is a signed floating-point constant with 3-bit exponent and normalized 4 bits of precision. For details of the range of constants available and the encoding of *imm*, see *Modified immediate constants in A64 floating-point instructions* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point move immediate (vector). This instruction copies an immediate floating-point constant into every element of the SIMD and FP destination register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.80 FMUL (vector, by element)

Floating-point Multiply (by element).

Syntax

`FMUL` *`Vd.T, Vn.T, Vm.Ts`*`[`*`index`*`]`

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier:

Vector, half-precision
Can be one of `4H` or `8H`.

Vector, single-precision and double-precision
Can be one of `2S`, `4S` or `2D`.

Vn
Is the name of the first SIMD and FP source register.

Ts
Is an element size specifier:

Vector, half-precision
Must be `H`.

Vector, single-precision and double-precision
Can be one of `S` or `D`.

index
Is the element index:

Vector, half-precision
Must be `H:L:M`.

Vector, single-precision and double-precision
Can be one of `H:L` or `H`.

Vm
Is the name of the second SIMD and FP source register in the range 0 to 31.

Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point Multiply (by element). 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, places the results in a vector, and writes the vector to the destination SIMD and FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-14 FMUL (Vector, single-precision and double-precision) specifier combinations

T	*Ts*	*index*
2S	S	0 to 3
4S	S	0 to 3
2D	D	0 or 1

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.81 FMUL (vector)

Floating-point Multiply (vector).

Syntax

`FMUL Vd.T, Vn.T, Vm.T`

Where:

T
Is an arrangement specifier:

Half-precision
Can be one of `4H` or `8H`.

Single-precision and double-precision
Can be one of `2S`, `4S` or `2D`.

Vd
Is the name of the SIMD and FP destination register.

Vn
Is the name of the first SIMD and FP source register.

Vm
Is the name of the second SIMD and FP source register.

Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point Multiply (vector). This instruction multiplies corresponding floating-point values in the vectors in the two source SIMD and FP registers, places the result in a vector, and writes the vector to the destination SIMD and FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.82 FMULX (vector, by element)

Floating-point Multiply extended (by element).

Syntax

`FMULX Vd.T, Vn.T, Vm.Ts[index]`

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier:

Vector, half-precision
Can be one of `4H` or `8H`.

Vector, single-precision and double-precision
Can be one of `2S`, `4S` or `2D`.

Vn
Is the name of the first SIMD and FP source register.

Ts
Is an element size specifier:

Vector, half-precision
Must be `H`.

Vector, single-precision and double-precision
Can be one of `S` or `D`.

index
Is the element index:

Vector, half-precision
Must be `H:L:M`.

Vector, single-precision and double-precision
Can be one of `H:L` or `H`.

Vm
Is the name of the second SIMD and FP source register in the range 0 to 31.

Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point Multiply extended (by element). This instruction multiplies the floating-point values in the vector elements in the first source SIMD and FP register by the specified floating-point value in the second source SIMD and FP register, places the results in a vector, and writes the vector to the destination SIMD and FP register.

Before each multiplication, a check is performed for whether one value is infinite and the other is zero. In this case, if only one of the values is negative, the result is 2.0, otherwise the result is -2.0.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-15 FMULX (Vector, single-precision and double-precision) specifier combinations

T	*Ts*	*index*
2S	S	0 to 3
4S	S	0 to 3
2D	D	0 or 1

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.83 FMULX (vector)

Floating-point Multiply extended.

Syntax

`FMULX Vd.T, Vn.T, Vm.T`

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier:

Vector half precision
Can be one of `4H` or `8H`.

Vector single-precision and double-precision
Can be one of `2S`, `4S` or `2D`.

Vn
Is the name of the first SIMD and FP source register.

Vm
Is the name of the second SIMD and FP source register.

Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point Multiply extended. This instruction multiplies corresponding floating-point values in the vectors of the two source SIMD and FP registers, places the resulting floating-point values in a vector, and writes the vector to the destination SIMD and FP register.

If one value is zero and the other value is infinite, the result is 2.0. In this case, the result is negative if only one of the values is negative, otherwise the result is positive.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.84 FNEG (vector)

Floating-point Negate (vector).

Syntax

```
FNEG Vd.T, Vn.T ; Half-precision
```

```
FNEG Vd.T, Vn.T ; Single-precision and double-precision
```

Where:

T

Is an arrangement specifier:

Half-precision

Can be one of `4H` or `8H`.

Single-precision and double-precision

Can be one of `2S`, `4S` or `2D`.

Vd

Is the name of the SIMD and FP destination register.

Vn

Is the name of the SIMD and FP source register.

Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point Negate (vector). This instruction negates the value of each vector element in the source SIMD and FP register, writes the result to a vector, and writes the vector to the destination SIMD and FP register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.85 FRECPE (vector)

Floating-point Reciprocal Estimate.

Syntax

```
FRECPE Vd.T, Vn.T ; Vector half precision

FRECPE Vd.T, Vn.T ; Vector single-precision and double-precision
```

Where:

Vd
Is the name of the SIMD and FP destination register

T
Is an arrangement specifier:

Vector half precision
Can be one of `4H` or `8H`.

Vector single-precision and double-precision
Can be one of `2S`, `4S` or `2D`.

Vn
Is the name of the SIMD and FP source register

Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point Reciprocal Estimate. This instruction finds an approximate reciprocal estimate for each vector element in the source SIMD and FP register, places the result in a vector, and writes the vector to the destination SIMD and FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.86 FRECPS (vector)

Floating-point Reciprocal Step.

Syntax

```
FRECPS Vd.T, Vn.T, Vm.T ; Vector half precision
```

```
FRECPS Vd.T, Vn.T, Vm.T ; Vector single-precision and double-precision
```

Where:

Vd
Is the name of the SIMD and FP destination register

T
Is an arrangement specifier:

Vector half precision
Can be one of `4H` or `8H`.

Vector single-precision and double-precision
Can be one of `2S`, `4S` or `2D`.

Vn
Is the name of the first SIMD and FP source register

Vm
Is the name of the second SIMD and FP source register

Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point Reciprocal Step. This instruction multiplies the corresponding floating-point values in the vectors of the two source SIMD and FP registers, subtracts each of the products from 2.0, places the resulting floating-point values in a vector, and writes the vector to the destination SIMD and FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.87 FRECPX (vector)

Floating-point Reciprocal exponent (scalar).

Syntax

```
FRECPX Hd, Hn ; Half-precision

FRECPX Vd, Vn ; Single-precision and double-precision
```

Where:

Hd
Is the 16-bit name of the SIMD and FP destination register.

Hn
Is the 16-bit name of the SIMD and FP source register.

V
Is a width specifier, and can be either `S` or `D`.

d
Is the number of the SIMD and FP destination register.

n
Is the number of the SIMD and FP source register.

Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point Reciprocal exponent (scalar). This instruction finds an approximate reciprocal exponent for each vector element in the source SIMD and FP register, places the result in a vector, and writes the vector to the destination SIMD and FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.88 FRINTA (vector)

Floating-point Round to Integral, to nearest with ties to Away (vector).

Syntax

```
FRINTA Vd.T, Vn.T ; Half-precision

FRINTA Vd.T, Vn.T ; Single-precision and double-precision
```

Where:

T

Is an arrangement specifier:

Half-precision

Can be one of `4H` or `8H`.

Single-precision and double-precision

Can be one of `2S`, `4S` or `2D`.

Vd

Is the name of the SIMD and FP destination register.

Vn

Is the name of the SIMD and FP source register.

Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point Round to Integral, to nearest with ties to Away (vector). This instruction rounds a vector of floating-point values in the SIMD and FP source register to integral floating-point values of the same size using the Round to Nearest with Ties to Away rounding mode, and writes the result to the SIMD and FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.89 FRINTI (vector)

Floating-point Round to Integral, using current rounding mode (vector).

Syntax

```
FRINTI Vd.T, Vn.T ; Half-precision

FRINTI Vd.T, Vn.T ; Single-precision and double-precision
```

Where:

T

Is an arrangement specifier:

Half-precision

Can be one of `4H` or `8H`.

Single-precision and double-precision

Can be one of `2S`, `4S` or `2D`.

Vd

Is the name of the SIMD and FP destination register.

Vn

Is the name of the SIMD and FP source register.

Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point Round to Integral, using current rounding mode (vector). This instruction rounds a vector of floating-point values in the SIMD and FP source register to integral floating-point values of the same size using the rounding mode that is determined by the FPCR, and writes the result to the SIMD and FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.90 FRINTM (vector)

Floating-point Round to Integral, toward Minus infinity (vector).

Syntax

```
FRINTM Vd.T, Vn.T ; Half-precision

FRINTM Vd.T, Vn.T ; Single-precision and double-precision
```

Where:

T

Is an arrangement specifier:

Half-precision

Can be one of `4H` or `8H`.

Single-precision and double-precision

Can be one of `2S`, `4S` or `2D`.

Vd

Is the name of the SIMD and FP destination register.

Vn

Is the name of the SIMD and FP source register.

Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point Round to Integral, toward Minus infinity (vector). This instruction rounds a vector of floating-point values in the SIMD and FP source register to integral floating-point values of the same size using the Round towards Minus Infinity rounding mode, and writes the result to the SIMD and FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.91 FRINTN (vector)

Floating-point Round to Integral, to nearest with ties to even (vector).

Syntax

```
FRINTN Vd.T, Vn.T ; Half-precision

FRINTN Vd.T, Vn.T ; Single-precision and double-precision
```

Where:

T

IIs an arrangement specifier:

Half-precision

Can be one of `4H` or `8H`.

Single-precision and double-precision

Can be one of `2S`, `4S` or `2D`.

Vd

Is the name of the SIMD and FP destination register.

Vn

Is the name of the SIMD and FP source register.

Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point Round to Integral, to nearest with ties to even (vector). This instruction rounds a vector of floating-point values in the SIMD and FP source register to integral floating-point values of the same size using the Round to Nearest rounding mode, and writes the result to the SIMD and FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.92 FRINTP (vector)

Floating-point Round to Integral, toward Plus infinity (vector).

Syntax

```
FRINTP Vd.T, Vn.T ; Half-precision

FRINTP Vd.T, Vn.T ; Single-precision and double-precision
```

Where:

T

Is an arrangement specifier:

Half-precision

Can be one of `4H` or `8H`.

Single-precision and double-precision

Can be one of `2S`, `4S` or `2D`.

Vd

Is the name of the SIMD and FP destination register.

Vn

Is the name of the SIMD and FP source register.

Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point Round to Integral, toward Plus infinity (vector). This instruction rounds a vector of floating-point values in the SIMD and FP source register to integral floating-point values of the same size using the Round towards Plus Infinity rounding mode, and writes the result to the SIMD and FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.93 FRINTX (vector)

Floating-point Round to Integral exact, using current rounding mode (vector).

Syntax

```
FRINTX Vd.T, Vn.T ; Half-precision

FRINTX Vd.T, Vn.T ; Single-precision and double-precision
```

Where:

T

Is an arrangement specifier:

Half-precision

Can be one of `4H` or `8H`.

Single-precision and double-precision

Can be one of `2S`, `4S` or `2D`.

Vd

Is the name of the SIMD and FP destination register.

Vn

Is the name of the SIMD and FP source register.

Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point Round to Integral exact, using current rounding mode (vector). This instruction rounds a vector of floating-point values in the SIMD and FP source register to integral floating-point values of the same size using the rounding mode that is determined by the FPCR, and writes the result to the SIMD and FP destination register.

An Inexact exception is raised when the result value is not numerically equal to the input value. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.94 FRINTZ (vector)

Floating-point Round to Integral, toward Zero (vector).

Syntax

```
FRINTZ Vd.T, Vn.T ; Half-precision

FRINTZ Vd.T, Vn.T ; Single-precision and double-precision
```

Where:

T

Is an arrangement specifier:

Half-precision

Can be one of `4H` or `8H`.

Single-precision and double-precision

Can be one of `2S`, `4S` or `2D`.

Vd

Is the name of the SIMD and FP destination register.

Vn

Is the name of the SIMD and FP source register.

Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point Round to Integral, toward Zero (vector). This instruction rounds a vector of floating-point values in the SIMD and FP source register to integral floating-point values of the same size using the Round towards Zero rounding mode, and writes the result to the SIMD and FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.95 FRSQRTE (vector)

Floating-point Reciprocal Square Root Estimate.

Syntax

```
FRSQRTE Vd.T, Vn.T ; Vector half precision
```

```
FRSQRTE Vd.T, Vn.T ; Vector single-precision and double-precision
```

Where:

Vd
Is the name of the SIMD and FP destination register

T
Is an arrangement specifier:

Vector half precision
Can be one of `4H` or `8H`.

Vector single-precision and double-precision
Can be one of `2S`, `4S` or `2D`.

Vn
Is the name of the SIMD and FP source register

Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point Reciprocal Square Root Estimate. This instruction calculates an approximate square root for each vector element in the source SIMD and FP register, places the result in a vector, and writes the vector to the destination SIMD and FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.96 FRSQRTS (vector)

Floating-point Reciprocal Square Root Step.

Syntax

```
FRSQRTS Vd.T, Vn.T, Vm.T ; Vector half precision
```

```
FRSQRTS Vd.T, Vn.T, Vm.T ; Vector single-precision and double-precision
```

Where:

Vd
Is the name of the SIMD and FP destination register

T
Is an arrangement specifier:

Vector half precision
Can be one of `4H` or `8H`.

Vector single-precision and double-precision
Can be one of `2S`, `4S` or `2D`.

Vn
Is the name of the first SIMD and FP source register

Vm
Is the name of the second SIMD and FP source register

Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point Reciprocal Square Root Step. This instruction multiplies corresponding floating-point values in the vectors of the two source SIMD and FP registers, subtracts each of the products from 3.0, divides these results by 2.0, places the results into a vector, and writes the vector to the destination SIMD and FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.97 FSQRT (vector)

Floating-point Square Root (vector).

Syntax

```
FSQRT Vd.T, Vn.T ; Half-precision
```

```
FSQRT Vd.T, Vn.T ; Single-precision and double-precision
```

Where:

T

Is an arrangement specifier:

Half-precision

Can be one of `4H` or `8H`.

Single-precision and double-precision

Can be one of `2S`, `4S` or `2D`.

Vd

Is the name of the SIMD and FP destination register.

Vn

Is the name of the SIMD and FP source register.

Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point Square Root (vector). This instruction calculates the square root for each vector element in the source SIMD and FP register, places the result in a vector, and writes the vector to the destination SIMD and FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.98 FSUB (vector)

Floating-point Subtract (vector).

Syntax

```
FSUB Vd.T, Vn.T, Vm.T ; Half-precision

FSUB Vd.T, Vn.T, Vm.T ; Single-precision and double-precision
```

Where:

T

Is an arrangement specifier:

Half-precision

Can be one of `4H` or `8H`.

Single-precision and double-precision

Can be one of `2S`, `4S` or `2D`.

Vd

Is the name of the SIMD and FP destination register.

Vn

Is the name of the first SIMD and FP source register.

Vm

Is the name of the second SIMD and FP source register.

Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

Usage

Floating-point Subtract (vector). This instruction subtracts the elements in the vector in the second source SIMD and FP register, from the corresponding elements in the vector in the first source SIMD and FP register, places each result into elements of a vector, and writes the vector to the destination SIMD and FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.99 INS (vector, element)

Insert vector element from another vector element.

This instruction is used by the alias `MOV` (element).

Syntax

`INS` *`Vd.Ts[index1], Vn.Ts[index2]`*

Where:

`Vd`
Is the name of the SIMD and FP destination register.

`Ts`
Is an element size specifier, and can be one of the values shown in Usage.

`index1`
Is the destination element index, in the range shown in Usage.

`Vn`
Is the name of the SIMD and FP source register.

`index2`
Is the source element index in the range shown in Usage.

Usage

Insert vector element from another vector element. This instruction copies the vector element of the source SIMD and FP register to the specified vector element of the destination SIMD and FP register.

This instruction can insert data into individual elements within a SIMD and FP register without clearing the remaining bits to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-16 INS (Vector) specifier combinations

`Ts`	*`index1`*	*`index2`*
B	0 to 15	0 to 15
H	0 to 7	0 to 7
S	0 to 3	0 to 3
D	0 or 1	0 or 1

Related references

20.117 MOV (vector, element) on page 20-1474.
20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.100 INS (vector, general)

Insert vector element from general-purpose register.

This instruction is used by the alias `MOV` (from general).

Syntax

`INS` *`Vd.Ts[index], Rn`*

Where:

`Vd`
Is the name of the SIMD and FP destination register.

`Ts`
Is an element size specifier, and can be one of the values shown in Usage.

`index`
Is the element index, in the range shown in Usage.

`R`
Is the width specifier for the general-purpose source register, and can be either `W` or `X`.

`n`
Is the number [0-30] of the general-purpose source register or ZR (31).

Usage

Insert vector element from general-purpose register. This instruction copies the contents of the source general-purpose register to the specified vector element in the destination SIMD and FP register.

This instruction can insert data into individual elements within a SIMD and FP register without clearing the remaining bits to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-17 INS (Vector) specifier combinations

`Ts`	*`index`*	*`R`*
B	0 to 15	W
H	0 to 7	W
S	0 to 3	W
D	0 or 1	X

Related references

20.118 MOV (vector, from general) on page 20-1475.
20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.101 LD1 (vector, multiple structures)

Load multiple single-element structures to one, two, three, or four registers.

Syntax

```
LD1 { Vt.T }, [Xn|SP] ; One register
LD1 { Vt.T, Vt2.T }, [Xn|SP] ; Two registers
LD1 { Vt.T, Vt2.T, Vt3.T }, [Xn|SP] ; Three registers
LD1 { Vt.T, Vt2.T, Vt3.T, Vt4.T }, [Xn|SP] ; Four registers
LD1 { Vt.T }, [Xn|SP], imm ; One register, immediate offset, Post-index
LD1 { Vt.T }, [Xn|SP], Xm ; One register, register offset, Post-index
LD1 { Vt.T, Vt2.T }, [Xn|SP], imm ; Two registers, immediate offset, Post-index
LD1 { Vt.T, Vt2.T }, [Xn|SP], Xm ; Two registers, register offset, Post-index
LD1 { Vt.T, Vt2.T, Vt3.T }, [Xn|SP], imm ; Three registers, immediate offset, Post-index
LD1 { Vt.T, Vt2.T, Vt3.T }, [Xn|SP], Xm ; Three registers, register offset, Post-index
LD1 { Vt.T, Vt2.T, Vt3.T, Vt4.T }, [Xn|SP], imm ; Four registers, immediate offset, Post-index
LD1 { Vt.T, Vt2.T, Vt3.T, Vt4.T }, [Xn|SP], Xm ; Four registers, register offset, Post-index
```

Where:

Vt
Is the name of the first or only SIMD and FP register to be transferred.

Vt2
Is the name of the second SIMD and FP register to be transferred.

Vt3
Is the name of the third SIMD and FP register to be transferred.

Vt4
Is the name of the fourth SIMD and FP register to be transferred.

imm
Is the post-index immediate offset:

One register, immediate offset
Can be one of `#8` or `#16`.

Two registers, immediate offset
Can be one of `#16` or `#32`.

Three registers, immediate offset
Can be one of `#24` or `#48`.

Four registers, immediate offset
Can be one of `#32` or `#64`.

Xm
Is the 64-bit name of the general-purpose post-index register, excluding XZR.

T
Is an arrangement specifier, and can be one of the values shown in Usage.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Usage

Load multiple single-element structures to one, two, three, or four registers. This instruction loads multiple single-element structures from memory and writes the result to one, two, three, or four SIMD and FP registers.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following tables show valid specifier combinations:

Table 20-18 LD1 (One register, immediate offset) specifier combinations

T	*imm*
8B	#8
16B	#16
4H	#8
8H	#16
2S	#8
4S	#16
1D	#8
2D	#16

Table 20-19 LD1 (Two registers, immediate offset) specifier combinations

T	*imm*
8B	#16
16B	#32
4H	#16
8H	#32
2S	#16
4S	#32
1D	#16
2D	#32

Table 20-20 LD1 (Three registers, immediate offset) specifier combinations

T	*imm*
8B	#24
16B	#48
4H	#24
8H	#48
2S	#24
4S	#48
1D	#24
2D	#48

Table 20-21 LD1 (Four registers, immediate offset) specifier combinations

T	*imm*
8B	#32
16B	#64
4H	#32
8H	#64
2S	#32
4S	#64
1D	#32
2D	#64

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.102 LD1 (vector, single structure)

Load one single-element structure to one lane of one register.

Syntax

```
LD1 { Vt.B }[index], [Xn|SP] ; 8-bit
LD1 { Vt.H }[index], [Xn|SP] ; 16-bit
LD1 { Vt.S }[index], [Xn|SP] ; 32-bit
LD1 { Vt.D }[index], [Xn|SP] ; 64-bit
LD1 { Vt.B }[index], [Xn|SP], #1 ; 8-bit, immediate offset, Post-index
LD1 { Vt.B }[index], [Xn|SP], Xm ; 8-bit, register offset, Post-index
LD1 { Vt.H }[index], [Xn|SP], #2 ; 16-bit, immediate offset, Post-index
LD1 { Vt.H }[index], [Xn|SP], Xm ; 16-bit, register offset, Post-index
LD1 { Vt.S }[index], [Xn|SP], #4 ; 32-bit, immediate offset, Post-index
LD1 { Vt.S }[index], [Xn|SP], Xm ; 32-bit, register offset, Post-index
LD1 { Vt.D }[index], [Xn|SP], #8 ; 64-bit, immediate offset, Post-index
LD1 { Vt.D }[index], [Xn|SP], Xm ; 64-bit, register offset, Post-index
```

Where:

Vt
Is the name of the first or only SIMD and FP register to be transferred.

index
The value depends on the instruction variant:

8-bit
Is the element index, in the range 0 to 15.

16-bit
Is the element index, in the range 0 to 7.

32-bit
Is the element index, in the range 0 to 3.

64-bit
Is the element index, and can be either 0 or 1.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Xm
Is the 64-bit name of the general-purpose post-index register, excluding XZR.

Usage

Load one single-element structure to one lane of one register. This instruction loads a single-element structure from memory and writes the result to the specified lane of the SIMD and FP register without affecting the other bits of the register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.103 LD1R (vector)

Load one single-element structure and Replicate to all lanes (of one register).

Syntax

```
LD1R { Vt.T }, [Xn|SP] ; No offset

LD1R { Vt.T }, [Xn|SP], imm ; Immediate offset, Post-index

LD1R { Vt.T }, [Xn|SP], Xm ; Register offset, Post-index
```

Where:

imm
Is the post-index immediate offset, and can be one of the values shown in Usage.

Xm
Is the 64-bit name of the general-purpose post-index register, excluding XZR.

Vt
Is the name of the first or only SIMD and FP register to be transferred.

T
Is an arrangement specifier, and can be one of the values shown in Usage.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Usage

Load one single-element structure and Replicate to all lanes (of one register). This instruction loads a single-element structure from memory and replicates the structure to all the lanes of the SIMD and FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-22 LD1R (Immediate offset) specifier combinations

T	*imm*
8B	#1
16B	#1
4H	#2
8H	#2
2S	#4
4S	#4
1D	#8
2D	#8

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.104 LD2 (vector, multiple structures)

Load multiple 2-element structures to two registers.

Syntax

```
LD2 { Vt.T, Vt2.T }, [Xn|SP] ; No offset
LD2 { Vt.T, Vt2.T }, [Xn|SP], imm ; Immediate offset, Post-index
LD2 { Vt.T, Vt2.T }, [Xn|SP], Xm ; Register offset, Post-index
```

Where:

Vt
Is the name of the first or only SIMD and FP register to be transferred.

Vt2
Is the name of the second SIMD and FP register to be transferred.

imm
Is the post-index immediate offset, and can be either `#16` or `#32`.

Xm
Is the 64-bit name of the general-purpose post-index register, excluding XZR.

T
Is an arrangement specifier, and can be one of `8B`, `16B`, `4H`, `8H`, `2S`, `4S` or `2D`.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Usage

Load multiple 2-element structures to two registers. This instruction loads multiple 2-element structures from memory and writes the result to the two SIMD and FP registers, with de-interleaving.

For an example of de-interleaving, see `LD3 (multiple structures)`.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.105 LD2 (vector, single structure)

Load single 2-element structure to one lane of two registers.

Syntax

```
LD2 { Vt.B, Vt2.B }[index], [Xn|SP] ; 8-bit
LD2 { Vt.H, Vt2.H }[index], [Xn|SP] ; 16-bit
LD2 { Vt.S, Vt2.S }[index], [Xn|SP] ; 32-bit
LD2 { Vt.D, Vt2.D }[index], [Xn|SP] ; 64-bit
LD2 { Vt.B, Vt2.B }[index], [Xn|SP], #2 ; 8-bit, immediate offset, Post-index
LD2 { Vt.B, Vt2.B }[index], [Xn|SP], Xm ; 8-bit, register offset, Post-index
LD2 { Vt.H, Vt2.H }[index], [Xn|SP], #4 ; 16-bit, immediate offset, Post-index
LD2 { Vt.H, Vt2.H }[index], [Xn|SP], Xm ; 16-bit, register offset, Post-index
LD2 { Vt.S, Vt2.S }[index], [Xn|SP], #8 ; 32-bit, immediate offset, Post-index
LD2 { Vt.S, Vt2.S }[index], [Xn|SP], Xm ; 32-bit, register offset, Post-index
LD2 { Vt.D, Vt2.D }[index], [Xn|SP], #16 ; 64-bit, immediate offset, Post-index
LD2 { Vt.D, Vt2.D }[index], [Xn|SP], Xm ; 64-bit, register offset, Post-index
```

Where:

Vt
Is the name of the first or only SIMD and FP register to be transferred.

Vt2
Is the name of the second SIMD and FP register to be transferred.

index
The value depends on the instruction variant:

8-bit
Is the element index, in the range 0 to 15.

16-bit
Is the element index, in the range 0 to 7.

32-bit
Is the element index, in the range 0 to 3.

64-bit
Is the element index, and can be either 0 or 1.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Xm
Is the 64-bit name of the general-purpose post-index register, excluding XZR.

Usage

Load single 2-element structure to one lane of two registers. This instruction loads a 2-element structure from memory and writes the result to the corresponding elements of the two SIMD and FP registers without affecting the other bits of the registers.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.106 LD2R (vector)

Load single 2-element structure and Replicate to all lanes of two registers.

Syntax

```
LD2R { Vt.T, Vt2.T }, [Xn|SP] ; No offset
LD2R { Vt.T, Vt2.T }, [Xn|SP], imm ; Immediate offset, Post-index
LD2R { Vt.T, Vt2.T }, [Xn|SP], Xm ; Register offset, Post-index
```

Where:

Vt
Is the name of the first or only SIMD and FP register to be transferred.

Vt2
Is the name of the second SIMD and FP register to be transferred.

imm
Is the post-index immediate offset, and can be one of the values shown in Usage.

Xm
Is the 64-bit name of the general-purpose post-index register, excluding XZR.

T
Is an arrangement specifier, and can be one of the values shown in Usage.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Usage

Load single 2-element structure and Replicate to all lanes of two registers. This instruction loads a 2-element structure from memory and replicates the structure to all the lanes of the two SIMD and FP registers.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-23 LD2R (Immediate offset) specifier combinations

T	*imm*
8B	#2
16B	#2
4H	#4
8H	#4
2S	#8
4S	#8
1D	#16
2D	#16

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.107 LD3 (vector, multiple structures)

Load multiple 3-element structures to three registers.

Syntax

```
LD3 { Vt.T, Vt2.T, Vt3.T }, [Xn|SP] ; No offset
LD3 { Vt.T, Vt2.T, Vt3.T }, [Xn|SP], imm ; Immediate offset, Post-index
LD3 { Vt.T, Vt2.T, Vt3.T }, [Xn|SP], Xm ; Register offset, Post-index
```

Where:

Vt
Is the name of the first or only SIMD and FP register to be transferred.

Vt2
Is the name of the second SIMD and FP register to be transferred.

Vt3
Is the name of the third SIMD and FP register to be transferred.

imm
Is the post-index immediate offset, and can be either `#24` or `#48`.

Xm
Is the 64-bit name of the general-purpose post-index register, excluding XZR.

T
Is an arrangement specifier, and can be one of `8B`, `16B`, `4H`, `8H`, `2S`, `4S` or `2D`.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Usage

Load multiple 3-element structures to three registers. This instruction loads multiple 3-element structures from memory and writes the result to the three SIMD and FP registers, with de-interleaving.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.108 LD3 (vector, single structure)

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

Syntax

```
LD3 { Vt.B, Vt2.B, Vt3.B }[index], [Xn|SP] ; 8-bit
LD3 { Vt.H, Vt2.H, Vt3.H }[index], [Xn|SP] ; 16-bit
LD3 { Vt.S, Vt2.S, Vt3.S }[index], [Xn|SP] ; 32-bit
LD3 { Vt.D, Vt2.D, Vt3.D }[index], [Xn|SP] ; 64-bit
LD3 { Vt.B, Vt2.B, Vt3.B }[index], [Xn|SP], #3 ; 8-bit, immediate offset, Post-index
LD3 { Vt.B, Vt2.B, Vt3.B }[index], [Xn|SP], Xm ; 8-bit, register offset, Post-index
LD3 { Vt.H, Vt2.H, Vt3.H }[index], [Xn|SP], #6 ; 16-bit, immediate offset, Post-index
LD3 { Vt.H, Vt2.H, Vt3.H }[index], [Xn|SP], Xm ; 16-bit, register offset, Post-index
LD3 { Vt.S, Vt2.S, Vt3.S }[index], [Xn|SP], #12 ; 32-bit, immediate offset, Post-index
LD3 { Vt.S, Vt2.S, Vt3.S }[index], [Xn|SP], Xm ; 32-bit, register offset, Post-index
LD3 { Vt.D, Vt2.D, Vt3.D }[index], [Xn|SP], #24 ; 64-bit, immediate offset, Post-index
LD3 { Vt.D, Vt2.D, Vt3.D }[index], [Xn|SP], Xm ; 64-bit, register offset, Post-index
```

Where:

Vt
Is the name of the first or only SIMD and FP register to be transferred.

Vt2
Is the name of the second SIMD and FP register to be transferred.

Vt3
Is the name of the third SIMD and FP register to be transferred.

index
The value depends on the instruction variant:

8-bit
Is the element index, in the range 0 to 15.

16-bit
Is the element index, in the range 0 to 7.

32-bit
Is the element index, in the range 0 to 3.

64-bit
Is the element index, and can be either 0 or 1.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Xm
Is the 64-bit name of the general-purpose post-index register, excluding XZR.

Usage

Load single 3-element structure to one lane of three registers). This instruction loads a 3-element structure from memory and writes the result to the corresponding elements of the three SIMD and FP registers without affecting the other bits of the registers.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.109 LD3R (vector)

Load single 3-element structure and Replicate to all lanes of three registers.

Syntax

```
LD3R { Vt.T, Vt2.T, Vt3.T }, [Xn|SP] ; No offset
LD3R { Vt.T, Vt2.T, Vt3.T }, [Xn|SP], imm ; Immediate offset, Post-index
LD3R { Vt.T, Vt2.T, Vt3.T }, [Xn|SP], Xm ; Register offset, Post-index
```

Where:

Vt
Is the name of the first or only SIMD and FP register to be transferred.

Vt2
Is the name of the second SIMD and FP register to be transferred.

Vt3
Is the name of the third SIMD and FP register to be transferred.

imm
Is the post-index immediate offset, and can be one of the values shown in Usage.

Xm
Is the 64-bit name of the general-purpose post-index register, excluding XZR.

T
Is an arrangement specifier, and can be one of the values shown in Usage.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Usage

Load single 3-element structure and Replicate to all lanes of three registers. This instruction loads a 3-element structure from memory and replicates the structure to all the lanes of the three SIMD and FP registers.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-24 LD3R (Immediate offset) specifier combinations

T	*imm*
8B	#3
16B	#3
4H	#6
8H	#6
2S	#12
4S	#12
1D	#24
2D	#24

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.110 LD4 (vector, multiple structures)

Load multiple 4-element structures to four registers.

Syntax

```
LD4 { Vt.T, Vt2.T, Vt3.T, Vt4.T }, [Xn|SP] ; No offset
LD4 { Vt.T, Vt2.T, Vt3.T, Vt4.T }, [Xn|SP], imm ; Immediate offset, Post-index
LD4 { Vt.T, Vt2.T, Vt3.T, Vt4.T }, [Xn|SP], Xm ; Register offset, Post-index
```

Where:

Vt
Is the name of the first or only SIMD and FP register to be transferred.

Vt2
Is the name of the second SIMD and FP register to be transferred.

Vt3
Is the name of the third SIMD and FP register to be transferred.

Vt4
Is the name of the fourth SIMD and FP register to be transferred.

imm
Is the post-index immediate offset, and can be either `#32` or `#64`.

Xm
Is the 64-bit name of the general-purpose post-index register, excluding XZR.

T
Is an arrangement specifier, and can be one of `8B`, `16B`, `4H`, `8H`, `2S`, `4S` or `2D`.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Usage

Load multiple 4-element structures to four registers. This instruction loads multiple 4-element structures from memory and writes the result to the four SIMD and FP registers, with de-interleaving.

For an example of de-interleaving, see `LD3 (multiple structures)`.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.111 LD4 (vector, single structure)

Load single 4-element structure to one lane of four registers.

Syntax

```
LD4 { Vt.B, Vt2.B, Vt3.B, Vt4.B }[index], [Xn|SP] ; 8-bit
LD4 { Vt.H, Vt2.H, Vt3.H, Vt4.H }[index], [Xn|SP] ; 16-bit
LD4 { Vt.S, Vt2.S, Vt3.S, Vt4.S }[index], [Xn|SP] ; 32-bit
LD4 { Vt.D, Vt2.D, Vt3.D, Vt4.D }[index], [Xn|SP] ; 64-bit
LD4 { Vt.B, Vt2.B, Vt3.B, Vt4.B }[index], [Xn|SP], #4 ; 8-bit, immediate offset, Post-index
LD4 { Vt.B, Vt2.B, Vt3.B, Vt4.B }[index], [Xn|SP], Xm ; 8-bit, register offset, Post-index
LD4 { Vt.H, Vt2.H, Vt3.H, Vt4.H }[index], [Xn|SP], #8 ; 16-bit, immediate offset, Post-index
LD4 { Vt.H, Vt2.H, Vt3.H, Vt4.H }[index], [Xn|SP], Xm ; 16-bit, register offset, Post-index
LD4 { Vt.S, Vt2.S, Vt3.S, Vt4.S }[index], [Xn|SP], #16 ; 32-bit, immediate offset, Post-index
LD4 { Vt.S, Vt2.S, Vt3.S, Vt4.S }[index], [Xn|SP], Xm ; 32-bit, register offset, Post-index
LD4 { Vt.D, Vt2.D, Vt3.D, Vt4.D }[index], [Xn|SP], #32 ; 64-bit, immediate offset, Post-index
LD4 { Vt.D, Vt2.D, Vt3.D, Vt4.D }[index], [Xn|SP], Xm ; 64-bit, register offset, Post-index
```

Where:

Vt
Is the name of the first or only SIMD and FP register to be transferred.

Vt2
Is the name of the second SIMD and FP register to be transferred.

Vt3
Is the name of the third SIMD and FP register to be transferred.

Vt4
Is the name of the fourth SIMD and FP register to be transferred.

index
The value depends on the instruction variant:

8-bit
Is the element index, in the range 0 to 15.

16-bit
Is the element index, in the range 0 to 7.

32-bit
Is the element index, in the range 0 to 3.

64-bit
Is the element index, and can be either 0 or 1.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Xm
Is the 64-bit name of the general-purpose post-index register, excluding XZR.

Usage

Load single 4-element structure to one lane of four registers. This instruction loads a 4-element structure from memory and writes the result to the corresponding elements of the four SIMD and FP registers without affecting the other bits of the registers.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.112 LD4R (vector)

Load single 4-element structure and Replicate to all lanes of four registers.

Syntax

```
LD4R { Vt.T, Vt2.T, Vt3.T, Vt4.T }, [Xn|SP] ; No offset
LD4R { Vt.T, Vt2.T, Vt3.T, Vt4.T }, [Xn|SP], imm ; Immediate offset, Post-index
LD4R { Vt.T, Vt2.T, Vt3.T, Vt4.T }, [Xn|SP], Xm ; Register offset, Post-index
```

Where:

Vt
Is the name of the first or only SIMD and FP register to be transferred.

Vt2
Is the name of the second SIMD and FP register to be transferred.

Vt3
Is the name of the third SIMD and FP register to be transferred.

Vt4
Is the name of the fourth SIMD and FP register to be transferred.

imm
Is the post-index immediate offset, and can be one of the values shown in Usage.

Xm
Is the 64-bit name of the general-purpose post-index register, excluding XZR.

T
Is an arrangement specifier, and can be one of the values shown in Usage.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Usage

Load single 4-element structure and Replicate to all lanes of four registers. This instruction loads a 4-element structure from memory and replicates the structure to all the lanes of the four SIMD and FP registers.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-25 LD4R (Immediate offset) specifier combinations

T	*imm*
8B	#4
16B	#4
4H	#8
8H	#8
2S	#16
4S	#16
1D	#32
2D	#32

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.113 MLA (vector, by element)

Multiply-Add to accumulator (vector, by element).

Syntax

```
MLA Vd.T, Vn.T, Vm.Ts[index]
```

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of the values shown in Usage.

Vn
Is the name of the first SIMD and FP source register.

Vm
Is the name of the second SIMD and FP source register:
- If *Ts* is H, then *Vm* must be in the range V0 to V15.
- If *Ts* is S, then *Vm* must be in the range V0 to V31.

Ts
Is an element size specifier, and can be either H or S.

index
Is the element index, in the range shown in Usage.

Usage

Multiply-Add to accumulator (vector, by element). 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 results with the vector elements of the destination SIMD and FP register. All the values in this instruction are unsigned integer values.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-26 MLA (Vector) specifier combinations

T	*Ts*	*index*
4H	H	0 to 7
8H	H	0 to 7
2S	S	0 to 3
4S	S	0 to 3

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.114 MLA (vector)

Multiply-Add to accumulator (vector).

Syntax

`MLA Vd.T, Vn.T, Vm.T`

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of `8B`, `16B`, `4H`, `8H`, `2S` or `4S`.

Vn
Is the name of the first SIMD and FP source register.

Vm
Is the name of the second SIMD and FP source register.

Usage

Multiply-Add to accumulator (vector). This instruction multiplies corresponding elements in the vectors of the two source SIMD and FP registers, and accumulates the results with the vector elements of the destination SIMD and FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.115 MLS (vector, by element)

Multiply-Subtract from accumulator (vector, by element).

Syntax

```
MLS Vd.T, Vn.T, Vm.Ts[index]
```

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of the values shown in Usage.

Vn
Is the name of the first SIMD and FP source register.

Vm
Is the name of the second SIMD and FP source register:

- If *Ts* is H, then *Vm* must be in the range V0 to V15.
- If *Ts* is S, then *Vm* must be in the range V0 to V31.

Ts
Is an element size specifier, and can be either H or S.

index
Is the element index, in the range shown in Usage.

Usage

Multiply-Subtract from accumulator (vector, by element). 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 subtracts the results from the vector elements of the destination SIMD and FP register. All the values in this instruction are unsigned integer values.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-27 MLS (Vector) specifier combinations

T	*Ts*	*index*
4H	H	0 to 7
8H	H	0 to 7
2S	S	0 to 3
4S	S	0 to 3

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.116 MLS (vector)

Multiply-Subtract from accumulator (vector).

Syntax

`MLS Vd.T, Vn.T, Vm.T`

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of `8B`, `16B`, `4H`, `8H`, `2S` or `4S`.

Vn
Is the name of the first SIMD and FP source register.

Vm
Is the name of the second SIMD and FP source register.

Usage

Multiply-Subtract from accumulator (vector). This instruction multiplies corresponding elements in the vectors of the two source SIMD and FP registers, and subtracts the results from the vector elements of the destination SIMD and FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.117 MOV (vector, element)

Move vector element to another vector element.

This instruction is an alias of `INS` (element).

The equivalent instruction is `INS Vd.Ts[index1], Vn.Ts[index2]`.

Syntax

`MOV Vd.Ts[index1], Vn.Ts[index2]`

Where:

Vd
Is the name of the SIMD and FP destination register.

Ts
Is an element size specifier, and can be one of the values shown in Usage.

index1
Is the destination element index, in the range shown in Usage.

Vn
Is the name of the SIMD and FP source register.

index2
Is the source element index in the range shown in Usage.

Usage

Move vector element to another vector element. This instruction copies the vector element of the source SIMD and FP register to the specified vector element of the destination SIMD and FP register.

This instruction can insert data into individual elements within a SIMD and FP register without clearing the remaining bits to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-28 MOV (Vector) specifier combinations

Ts	*index1*	*index2*
B	0 to 15	0 to 15
H	0 to 7	0 to 7
S	0 to 3	0 to 3
D	0 or 1	0 or 1

Related references

20.99 INS (vector, element) on page 20-1453.
20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.118 MOV (vector, from general)

Move general-purpose register to a vector element.

This instruction is an alias of `INS` (general).

The equivalent instruction is `INS Vd.Ts[index], Rn`.

Syntax

`MOV Vd.Ts[index], Rn`

Where:

Vd
Is the name of the SIMD and FP destination register.

Ts
Is an element size specifier, and can be one of the values shown in Usage.

index
Is the element index, in the range shown in Usage.

R
Is the width specifier for the general-purpose source register, and can be either `W` or `X`.

n
Is the number [0-30] of the general-purpose source register or ZR (31).

Usage

Move general-purpose register to a vector element. This instruction copies the contents of the source general-purpose register to the specified vector element in the destination SIMD and FP register.

This instruction can insert data into individual elements within a SIMD and FP register without clearing the remaining bits to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-29 MOV (Vector) specifier combinations

Ts	*index*	*R*
B	0 to 15	W
H	0 to 7	W
S	0 to 3	W
D	0 or 1	X

Related references

20.100 INS (vector, general) on page 20-1454.
20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.119 MOV (vector)

Move vector.

This instruction is an alias of `ORR` (vector, register).

The equivalent instruction is `ORR Vd.T, Vn.T, Vn.T`.

Syntax

`MOV Vd.T, Vn.T`

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be either `8B` or `16B`.

Vn
Is the name of the first SIMD and FP source register.

Usage

Move vector. This instruction copies the vector in the source SIMD and FP register into the destination SIMD and FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.130 ORR (vector, register) on page 20-1487.
20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.120 MOV (vector, to general)

Move vector element to general-purpose register.

This instruction is an alias of `UMOV`.

The equivalent instruction is `UMOV Wd, Vn.S[index]`.

Syntax

```
MOV Wd, Vn.S[index] ; 32-bit

MOV Xd, Vn.D[index] ; 64-bit
```

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

index
The value depends on the instruction variant:

32-bit
Is the element index.

64-bit
Is the element index and can be either 0 or 1.

Xd
Is the 64-bit name of the general-purpose destination register.

Vn
Is the name of the SIMD and FP source register.

Usage

Move vector element to general-purpose register. This instruction reads the unsigned integer from the source SIMD and FP register, zero-extends it to form a 32-bit or 64-bit value, and writes the result to the destination general-purpose register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.248 UMOV (vector) on page 20-1611.
20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.121 MOVI (vector)

Move Immediate (vector).

Syntax

```
MOVI Vd.T, #imm8{, LSL #0} ; 8-bit

MOVI Vd.T, #imm8{, LSL #amount} ; 16-bit shifted immediate

MOVI Vd.T, #imm8{, LSL #amount} ; 32-bit shifted immediate

MOVI Vd.T, #imm8, MSL #amount ; 32-bit shifting ones

MOVI Dd, #imm ; 64-bit scalar

MOVI Vd.2D, #imm ; 64-bit vector
```

Where:

Vd
Is the name of the SIMD and FP destination register

T
Is an arrangement specifier:

8-bit
Can be one of `8B` or `16B`.

16-bit shifted immediate
Can be one of `4H` or `8H`.

32-bit shifted immediate
Can be one of `2S` or `4S`.

32-bit shifting ones
Can be one of `2S` or `4S`.

imm8
Is an 8-bit immediate.

amount
Is the shift amount:

16-bit shifted immediate
Can be one of `0` or `8`.

32-bit shifted immediate
Can be one of `0`, `8`, `16` or `24`.

32-bit shifting ones
Can be one of `8` or `16`.

Defaults to zero if `LSL` is omitted.

Dd
Is the 64-bit name of the SIMD and FP destination register.

imm
Is a 64-bit immediate.

Usage

Move Immediate (vector). This instruction places an immediate constant into every vector element of the destination SIMD and FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.122 MUL (vector, by element)

Multiply (vector, by element).

Syntax

```
MUL Vd.T, Vn.T, Vm.Ts[index]
```

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of the values shown in Usage.

Vn
Is the name of the first SIMD and FP source register.

Vm
Is the name of the second SIMD and FP source register:

- If *Ts* is H, then *Vm* must be in the range V0 to V15.
- If *Ts* is S, then *Vm* must be in the range V0 to V31.

Ts
Is an element size specifier, and can be either H or S.

index
Is the element index, in the range shown in Usage.

Usage

Multiply (vector, by element). 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, places the results in a vector, and writes the vector to the destination SIMD and FP register. All the values in this instruction are unsigned integer values.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-30 MUL (Vector) specifier combinations

T	*Ts*	*index*
4H	H	0 to 7
8H	H	0 to 7
2S	S	0 to 3
4S	S	0 to 3

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.123 MUL (vector)

Multiply (vector).

Syntax

`MUL Vd.T, Vn.T, Vm.T`

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of `8B`, `16B`, `4H`, `8H`, `2S` or `4S`.

Vn
Is the name of the first SIMD and FP source register.

Vm
Is the name of the second SIMD and FP source register.

Usage

Multiply (vector). This instruction multiplies corresponding elements in the vectors of the two source SIMD and FP registers, places the results in a vector, and writes the vector to the destination SIMD and FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.124 MVN (vector)

Bitwise NOT (vector).

This instruction is an alias of `NOT`.

The equivalent instruction is `NOT Vd.T, Vn.T`.

Syntax

`MVN Vd.T, Vn.T`

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be either `8B` or `16B`.

Vn
Is the name of the SIMD and FP source register.

Usage

Bitwise NOT (vector). This instruction reads each vector element from the source SIMD and FP register, places the inverse of each value into a vector, and writes the vector to the destination SIMD and FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.127 NOT (vector) on page 20-1484.
20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.125 MVNI (vector)

Move inverted Immediate (vector).

Syntax

```
MVNI Vd.T, #imm8{, LSL #amount} ; 16-bit shifted immediate

MVNI Vd.T, #imm8{, LSL #amount} ; 32-bit shifted immediate

MVNI Vd.T, #imm8, MSL #amount ; 32-bit shifting ones
```

Where:

T

Is an arrangement specifier:

16-bit shifted immediate
Can be one of `4H` or `8H`.

32-bit shifted immediate
Can be one of `2S` or `4S`.

32-bit shifting ones
Can be one of `2S` or `4S`.

amount

Is the shift amount:

16-bit shifted immediate
Can be one of `0` or `8`.

32-bit shifted immediate
Can be one of `0`, `8`, `16` or `24`.

32-bit shifting ones
Can be one of `8` or `16`.

Defaults to zero if `LSL` is omitted.

Vd

Is the name of the SIMD and FP destination register.

imm8

Is an 8-bit immediate.

Usage

Move inverted Immediate (vector). This instruction places the inverse of an immediate constant into every vector element of the destination SIMD and FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.126 NEG (vector)

Negate (vector).

Syntax

`NEG Vd.T, Vn.T`

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of `8B`, `16B`, `4H`, `8H`, `2S`, `4S` or `2D`.

Vn
Is the name of the SIMD and FP source register.

Usage

Negate (vector). This instruction reads each vector element from the source SIMD and FP register, negates each value, puts the result into a vector, and writes the vector to the destination SIMD and FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.127 NOT (vector)

Bitwise NOT (vector).

This instruction is used by the alias `MVN`.

Syntax

`NOT Vd.T, Vn.T`

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be either `8B` or `16B`.

Vn
Is the name of the SIMD and FP source register.

Usage

Bitwise NOT (vector). This instruction reads each vector element from the source SIMD and FP register, places the inverse of each value into a vector, and writes the vector to the destination SIMD and FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.124 MVN (vector) on page 20-1481.
20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.128 ORN (vector)

Bitwise inclusive OR NOT (vector).

Syntax

`ORN` *`Vd.T, Vn.T, Vm.T`*

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be either `8B` or `16B`.

Vn
Is the name of the first SIMD and FP source register.

Vm
Is the name of the second SIMD and FP source register.

Usage

Bitwise inclusive OR NOT (vector). This instruction performs a bitwise OR NOT between the two source SIMD and FP registers, and writes the result to the destination SIMD and FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.129 ORR (vector, immediate)

Bitwise inclusive OR (vector, immediate).

Syntax

```
ORR Vd.T, #imm8{, LSL #amount}
```

Where:

T
Is an arrangement specifier:

16-bit
Can be one of `4H` or `8H`.

32-bit
Can be one of `2S` or `4S`.

amount
Is the shift amount:

16-bit
Can be one of `0` or `8`.

32-bit
Can be one of `0`, `8`, `16` or `24`.

Defaults to zero if `LSL` is omitted.

Vd
Is the name of the SIMD and FP register.

imm8
Is an 8-bit immediate.

Usage

Bitwise inclusive OR (vector, immediate). This instruction reads each vector element from the destination SIMD and FP register, performs a bitwise OR between each result and an immediate constant, places the result into a vector, and writes the vector to the destination SIMD and FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.130 ORR (vector, register)

Bitwise inclusive OR (vector, register).

This instruction is used by the alias `MOV` (vector).

Syntax

`ORR` *`Vd.T, Vn.T, Vm.T`*

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be either `8B` or `16B`.

Vn
Is the name of the first SIMD and FP source register.

Vm
Is the name of the second SIMD and FP source register.

Usage

Bitwise inclusive OR (vector, register). This instruction performs a bitwise OR between the two source SIMD and FP registers, and writes the result to the destination SIMD and FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.119 MOV (vector) on page 20-1476.
20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.131 PMUL (vector)

Polynomial Multiply.

Syntax

`PMUL Vd.T, Vn.T, Vm.T`

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be either `8B` or `16B`.

Vn
Is the name of the first SIMD and FP source register.

Vm
Is the name of the second SIMD and FP source register.

Usage

Polynomial Multiply. This instruction multiplies corresponding elements in the vectors of the two source SIMD and FP registers, places the results in a vector, and writes the vector to the destination SIMD and FP register.

For information about multiplying polynomials see *Polynomial arithmetic over {0, 1}* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.132 PMULL, PMULL2 (vector)

Polynomial Multiply Long.

Syntax

`PMULL{2} Vd.Ta, Vn.Tb, Vm.Tb`

Where:

2
Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements. See <Q> in the Usage table.

Vd
Is the name of the SIMD and FP destination register.

Ta
Is an arrangement specifier, `8H`.

Vn
Is the name of the first SIMD and FP source register.

Tb
Is an arrangement specifier, and can be one of the values shown in Usage.

Vm
Is the name of the second SIMD and FP source register.

Usage

Polynomial Multiply Long. This instruction multiplies corresponding elements in the lower or upper half of the vectors of the two source SIMD and FP registers, places the results in a vector, and writes the vector to the destination SIMD and FP register. The destination vector elements are twice as long as the elements that are multiplied.

For information about multiplying polynomials see *Polynomial arithmetic over {0, 1}* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

The `PMULL` instruction extracts each source vector from the lower half of each source register, while the `PMULL2` instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-31 PMULL, PMULL2 (Vector) specifier combinations

<Q>	*Ta*	*Tb*
-	8H	8B
2	8H	16B

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.133 RADDHN, RADDHN2 (vector)

Rounding Add returning High Narrow.

Syntax

`RADDHN{2} Vd.Tb, Vn.Ta, Vm.Ta`

Where:

2
Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements. See <Q> in the Usage table.

Vd
Is the name of the SIMD and FP destination register.

Tb
Is an arrangement specifier, and can be one of the values shown in Usage.

Vn
Is the name of the first SIMD and FP source register.

Ta
Is an arrangement specifier, and can be one of the values shown in Usage.

Vm
Is the name of the second SIMD and FP source register.

Usage

Rounding Add returning High Narrow. This instruction adds each vector element in the first source SIMD and FP register to the corresponding vector element in the second source SIMD and FP register, places the most significant half of the result into a vector, and writes the vector to the lower or upper half of the destination SIMD and FP register.

The results are rounded. For truncated results, see *20.4 ADDHN, ADDHN2 (vector)* on page 20-1353.

The `RADDHN` instruction writes the vector to the lower half of the destination register and clears the upper half, while the `RADDHN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-32 RADDHN, RADDHN2 (Vector) specifier combinations

<Q>	*Tb*	*Ta*
-	8B	8H
2	16B	8H
-	4H	4S
2	8H	4S
-	2S	2D
2	4S	2D

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.134 RBIT (vector)

Reverse Bit order (vector).

Syntax

`RBIT` *`Vd.T, Vn.T`*

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be either `8B` or `16B`.

Vn
Is the name of the SIMD and FP source register.

Usage

Reverse Bit order (vector). This instruction reads each vector element from the source SIMD and FP register, reverses the bits of the element, places the results into a vector, and writes the vector to the destination SIMD and FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.135 REV16 (vector)

Reverse elements in 16-bit halfwords (vector).

Syntax

`REV16 Vd.T, Vn.T`

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be either `8B` or `16B`.

Vn
Is the name of the SIMD and FP source register.

Usage

Reverse elements in 16-bit halfwords (vector). This instruction reverses the order of 8-bit elements in each halfword of the vector in the source SIMD and FP register, places the results into a vector, and writes the vector to the destination SIMD and FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.136 REV32 (vector)

Reverse elements in 32-bit words (vector).

Syntax

`REV32 Vd.T, Vn.T`

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of `8B`, `16B`, `4H` or `8H`.

Vn
Is the name of the SIMD and FP source register.

Usage

Reverse elements in 32-bit words (vector). This instruction reverses the order of 8-bit or 16-bit elements in each word of the vector in the source SIMD and FP register, places the results into a vector, and writes the vector to the destination SIMD and FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.137 REV64 (vector)

Reverse elements in 64-bit doublewords (vector).

Syntax

`REV64 Vd.T, Vn.T`

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of `8B`, `16B`, `4H`, `8H`, `2S` or `4S`.

Vn
Is the name of the SIMD and FP source register.

Usage

Reverse elements in 64-bit doublewords (vector). This instruction reverses the order of 8-bit, 16-bit, or 32-bit elements in each doubleword of the vector in the source SIMD and FP register, places the results into a vector, and writes the vector to the destination SIMD and FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.138 RSHRN, RSHRN2 (vector)

Rounding Shift Right Narrow (immediate).

Syntax

`RSHRN{2} Vd.Tb, Vn.Ta, #shift`

Where:

2
Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements. See <Q> in the Usage table.

Vd
Is the name of the SIMD and FP destination register.

Tb
Is an arrangement specifier, and can be one of the values shown in Usage.

Vn
Is the name of the SIMD and FP source register.

Ta
Is an arrangement specifier, and can be one of the values shown in Usage.

shift
Is the right shift amount, in the range 1 to the destination element width in bits, and can be one of the values shown in Usage.

Usage

Rounding Shift Right Narrow (immediate). This instruction reads each unsigned integer value from the vector in the source SIMD and FP register, right shifts each result by an immediate value, writes the final result to a vector, and writes the vector to the lower or upper half of the destination SIMD and FP register. The destination vector elements are half as long as the source vector elements. The results are rounded. For truncated results, see *20.154 SHRN, SHRN2 (vector)* on page 20-1511.

The `RSHRN` instruction writes the vector to the lower half of the destination register and clears the upper half, while the `RSHRN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-33 RSHRN, RSHRN2 (Vector) specifier combinations

<Q>	*Tb*	*Ta*	*shift*
-	8B	8H	1 to 8
2	16B	8H	1 to 8
-	4H	4S	1 to 16
2	8H	4S	1 to 16
-	2S	2D	1 to 32
2	4S	2D	1 to 32

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.139 RSUBHN, RSUBHN2 (vector)

Rounding Subtract returning High Narrow.

Syntax

`RSUBHN{2} Vd.Tb, Vn.Ta, Vm.Ta`

Where:

2
Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements. See <Q> in the Usage table.

Vd
Is the name of the SIMD and FP destination register.

Tb
Is an arrangement specifier, and can be one of the values shown in Usage.

Vn
Is the name of the first SIMD and FP source register.

Ta
Is an arrangement specifier, and can be one of the values shown in Usage.

Vm
Is the name of the second SIMD and FP source register.

Usage

Rounding Subtract returning High Narrow. This instruction subtracts each vector element of the second source SIMD and FP register from the corresponding vector element of the first source SIMD and FP register, places the most significant half of the result into a vector, and writes the vector to the lower or upper half of the destination SIMD and FP register.

The results are rounded. For truncated results, see *20.218 SUBHN, SUBHN2 (vector)* on page 20-1581.

The `RSUBHN` instruction writes the vector to the lower half of the destination register and clears the upper half, while the `RSUBHN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-34 RSUBHN, RSUBHN2 (Vector) specifier combinations

<Q>	*Tb*	*Ta*
-	8B	8H
2	16B	8H
-	4H	4S
2	8H	4S
-	2S	2D
2	4S	2D

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.140 SABA (vector)

Signed Absolute difference and Accumulate.

Syntax

`SABA Vd.T, Vn.T, Vm.T`

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of `8B`, `16B`, `4H`, `8H`, `2S` or `4S`.

Vn
Is the name of the first SIMD and FP source register.

Vm
Is the name of the second SIMD and FP source register.

Usage

Signed Absolute difference and Accumulate. This instruction subtracts the elements of the vector of the second source SIMD and FP register from the corresponding elements of the first source SIMD and FP register, and accumulates the absolute values of the results into the elements of the vector of the destination SIMD and FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.141 SABAL, SABAL2 (vector)

Signed Absolute difference and Accumulate Long.

Syntax

`SABAL{2} Vd.Ta, Vn.Tb, Vm.Tb`

Where:

2
Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements. See <Q> in the Usage table.

Vd
Is the name of the SIMD and FP destination register.

Ta
Is an arrangement specifier, and can be one of the values shown in Usage.

Vn
Is the name of the first SIMD and FP source register.

Tb
Is an arrangement specifier, and can be one of the values shown in Usage.

Vm
Is the name of the second SIMD and FP source register.

Usage

Signed Absolute difference and Accumulate Long. This instruction subtracts the vector elements in the lower or upper half of the second source SIMD and FP register from the corresponding vector elements of the first source SIMD and FP register, and accumulates the absolute values of the results into the vector elements of the destination SIMD and FP register. The destination vector elements are twice as long as the source vector elements.

The `SABAL` instruction extracts each source vector from the lower half of each source register, while the `SABAL2` instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-35 SABAL, SABAL2 (Vector) specifier combinations

<Q>	*Ta*	*Tb*
-	8H	8B
2	8H	16B
-	4S	4H
2	4S	8H
-	2D	2S
2	2D	4S

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.142 SABD (vector)

Signed Absolute Difference.

Syntax

`SABD Vd.T, Vn.T, Vm.T`

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of `8B`, `16B`, `4H`, `8H`, `2S` or `4S`.

Vn
Is the name of the first SIMD and FP source register.

Vm
Is the name of the second SIMD and FP source register.

Usage

Signed Absolute Difference. This instruction subtracts the elements of the vector of the second source SIMD and FP register from the corresponding elements of the first source SIMD and FP register, places the absolute values of the results into a vector, and writes the vector to the destination SIMD and FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.143 SABDL, SABDL2 (vector)

Signed Absolute Difference Long.

Syntax

`SABDL{2} Vd.Ta, Vn.Tb, Vm.Tb`

Where:

2
Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements. See <Q> in the Usage table.

Vd
Is the name of the SIMD and FP destination register.

Ta
Is an arrangement specifier, and can be one of the values shown in Usage.

Vn
Is the name of the first SIMD and FP source register.

Tb
Is an arrangement specifier, and can be one of the values shown in Usage.

Vm
Is the name of the second SIMD and FP source register.

Usage

Signed Absolute Difference Long. This instruction subtracts the vector elements of the second source SIMD and FP register from the corresponding vector elements of the first source SIMD and FP register, places the absolute value of the results into a vector, and writes the vector to the lower or upper half of the destination SIMD and FP register. The destination vector elements are twice as long as the source vector elements.

The `SABDL` instruction writes the vector to the lower half of the destination register and clears the upper half, while the `SABDL2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-36 SABDL, SABDL2 (Vector) specifier combinations

<Q>	*Ta*	*Tb*
-	8H	8B
2	8H	16B
-	4S	4H
2	4S	8H
-	2D	2S
2	2D	4S

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.144 SADALP (vector)

Signed Add and Accumulate Long Pairwise.

Syntax

```
SADALP Vd.Ta, Vn.Tb
```

Where:

Vd
Is the name of the SIMD and FP destination register.

Ta
Is an arrangement specifier, and can be one of the values shown in Usage.

Vn
Is the name of the SIMD and FP source register.

Tb
Is an arrangement specifier, and can be one of the values shown in Usage.

Usage

Signed Add and Accumulate Long Pairwise. This instruction adds pairs of adjacent signed integer values from the vector in the source SIMD and FP register and accumulates the results into the vector elements of the destination SIMD and FP register. The destination vector elements are twice as long as the source vector elements.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-37 SADALP (Vector) specifier combinations

Ta	*Tb*
4H	8B
8H	16B
2S	4H
4S	8H
1D	2S
2D	4S

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.145 SADDL, SADDL2 (vector)

Signed Add Long (vector).

Syntax

`SADDL{2} Vd.Ta, Vn.Tb, Vm.Tb`

Where:

2
Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements. See <Q> in the Usage table.

Vd
Is the name of the SIMD and FP destination register.

Ta
Is an arrangement specifier, and can be one of the values shown in Usage.

Vn
Is the name of the first SIMD and FP source register.

Tb
Is an arrangement specifier, and can be one of the values shown in Usage.

Vm
Is the name of the second SIMD and FP source register.

Usage

Signed Add Long (vector). This instruction adds each vector element in the lower or upper half of the first source SIMD and FP register to the corresponding vector element of the second source SIMD and FP register, places the results into a vector, and writes the vector to the destination SIMD and FP register. The destination vector elements are twice as long as the source vector elements. All the values in this instruction are signed integer values.

The `SADDL` instruction extracts each source vector from the lower half of each source register, while the `SADDL2` instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-38 SADDL, SADDL2 (Vector) specifier combinations

<Q>	*Ta*	*Tb*
-	8H	8B
2	8H	16B
-	4S	4H
2	4S	8H
-	2D	2S
2	2D	4S

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.146 SADDLP (vector)

Signed Add Long Pairwise.

Syntax

```
SADDLP Vd.Ta, Vn.Tb
```

Where:

Vd
Is the name of the SIMD and FP destination register.

Ta
Is an arrangement specifier, and can be one of the values shown in Usage.

Vn
Is the name of the SIMD and FP source register.

Tb
Is an arrangement specifier, and can be one of the values shown in Usage.

Usage

Signed Add Long Pairwise. This instruction adds pairs of adjacent signed integer values from the vector in the source SIMD and FP register, places the result into a vector, and writes the vector to the destination SIMD and FP register. The destination vector elements are twice as long as the source vector elements.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-39 SADDLP (Vector) specifier combinations

Ta	*Tb*
4H	8B
8H	16B
2S	4H
4S	8H
1D	2S
2D	4S

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.147 SADDLV (vector)

Signed Add Long across Vector.

Syntax

```
SADDLV Vd, Vn.T
```

Where:

V
Is the destination width specifier, and can be one of the values shown in Usage.

d
Is the number of the SIMD and FP destination register.

Vn
Is the name of the SIMD and FP source register.

T
Is an arrangement specifier, and can be one of the values shown in Usage.

Usage

Signed Add Long across Vector. This instruction adds every vector element in the source SIMD and FP register together, and writes the scalar result to the destination SIMD and FP register. The destination scalar is twice as long as the source vector elements. All the values in this instruction are signed integer values.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-40 SADDLV (Vector) specifier combinations

V	*T*
H	8B
H	16B
S	4H
S	8H
D	4S

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.148 SADDW, SADDW2 (vector)

Signed Add Wide.

Syntax

`SADDW{2}` *`Vd.Ta, Vn.Ta, Vm.Tb`*

Where:

`2`
Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements. See <Q> in the Usage table.

`Vd`
Is the name of the SIMD and FP destination register.

`Ta`
Is an arrangement specifier, and can be one of the values shown in Usage.

`Vn`
Is the name of the first SIMD and FP source register.

`Vm`
Is the name of the second SIMD and FP source register.

`Tb`
Is an arrangement specifier, and can be one of the values shown in Usage.

Usage

Signed Add Wide. This instruction adds vector elements of the first source SIMD and FP register to the corresponding vector elements in the lower or upper half of the second source SIMD and FP register, places the results in a vector, and writes the vector to the SIMD and FP destination register.

The `SADDW` instruction extracts the second source vector from the lower half of the second source register, while the `SADDW2` instruction extracts the second source vector from the upper half of the second source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-41 SADDW, SADDW2 (Vector) specifier combinations

<Q>	*Ta*	*Tb*
-	8H	8B
2	8H	16B
-	4S	4H
2	4S	8H
-	2D	2S
2	2D	4S

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.149 SCVTF (vector, fixed-point)

Signed fixed-point Convert to Floating-point (vector).

Syntax

`SCVTF Vd.T, Vn.T, #fbits`

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of the values shown in Usage.

Vn
Is the name of the SIMD and FP source register.

fbits
Is the number of fractional bits, in the range 1 to the element width.

Usage

Signed fixed-point Convert to Floating-point (vector). This instruction converts each element in a vector from fixed-point to floating-point using the rounding mode that is specified by the FPCR, and writes the result to the SIMD and FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-42 SCVTF (Vector) specifier combinations

T	*fbits*
4H	
8H	
2S	1 to 32
4S	1 to 32
2D	1 to 64

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.150 SCVTF (vector, integer)

Signed integer Convert to Floating-point (vector).

Syntax

```
SCVTF Vd.T, Vn.T ; Vector half precision

SCVTF Vd.T, Vn.T ; Vector single-precision and double-precision
```

Where:

Vd
Is the name of the SIMD and FP destination register

T
Is an arrangement specifier:

Vector half precision
Can be one of `4H` or `8H`.

Vector single-precision and double-precision
Can be one of `2S`, `4S` or `2D`.

Vn
Is the name of the SIMD and FP source register

Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

Usage

Signed integer Convert to Floating-point (vector). This instruction converts each element in a vector from signed integer to floating-point using the rounding mode that is specified by the FPCR, and writes the result to the SIMD and FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.151 SHADD (vector)

Signed Halving Add.

Syntax

`SHADD Vd.T, Vn.T, Vm.T`

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of `8B`, `16B`, `4H`, `8H`, `2S` or `4S`.

Vn
Is the name of the first SIMD and FP source register.

Vm
Is the name of the second SIMD and FP source register.

Usage

Signed Halving Add. This instruction adds corresponding signed integer values from the two source SIMD and FP registers, shifts each result right one bit, places the results into a vector, and writes the vector to the destination SIMD and FP register.

The results are truncated. For rounded results, see *20.198 SRHADD (vector)* on page 20-1558.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.152 SHL (vector)

Shift Left (immediate).

Syntax

`SHL Vd.T, Vn.T, #shift`

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of the values shown in Usage.

Vn
Is the name of the SIMD and FP source register.

shift
Is the left shift amount, in the range 0 to the element width in bits minus 1, and can be one of the values shown in Usage.

Usage

Shift Left (immediate). This instruction reads each value from a vector, right shifts each result by an immediate value, writes the final result to a vector, and writes the vector to the destination SIMD and FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-43 SHL (Vector) specifier combinations

T	*shift*
8B	0 to 7
16B	0 to 7
4H	0 to 15
8H	0 to 15
2S	0 to 31
4S	0 to 31
2D	0 to 63

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.153 SHLL, SHLL2 (vector)

Shift Left Long (by element size).

Syntax

```
SHLL{2} Vd.Ta, Vn.Tb, #shift
```

Where:

2
Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements. See <Q> in the Usage table.

Vd
Is the name of the SIMD and FP destination register.

Ta
Is an arrangement specifier, and can be one of the values shown in Usage.

Vn
Is the name of the SIMD and FP source register.

Tb
Is an arrangement specifier, and can be one of the values shown in Usage.

shift
Is the left shift amount, which must be equal to the source element width in bits, and can be one of the values shown in Usage.

Usage

Shift Left Long (by element size). This instruction reads each vector element in the lower or upper half of the source SIMD and FP register, left shifts each result by the element size, writes the final result to a vector, and writes the vector to the destination SIMD and FP register. The destination vector elements are twice as long as the source vector elements.

The `SHLL` instruction extracts vector elements from the lower half of the source register, while the `SHLL2` instruction extracts vector elements from the upper half of the source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-44 SHLL, SHLL2 (Vector) specifier combinations

<Q>	*Ta*	*Tb*	*shift*
-	8H	8B	8
2	8H	16B	8
-	4S	4H	16
2	4S	8H	16
-	2D	2S	32
2	2D	4S	32

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.154 SHRN, SHRN2 (vector)

Shift Right Narrow (immediate).

Syntax

```
SHRN{2} Vd.Tb, Vn.Ta, #shift
```

Where:

2
Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements. See <Q> in the Usage table.

Vd
Is the name of the SIMD and FP destination register.

Tb
Is an arrangement specifier, and can be one of the values shown in Usage.

Vn
Is the name of the SIMD and FP source register.

Ta
Is an arrangement specifier, and can be one of the values shown in Usage.

shift
Is the right shift amount, in the range 1 to the destination element width in bits, and can be one of the values shown in Usage.

Usage

Shift Right Narrow (immediate). This instruction reads each unsigned integer value from the source SIMD and FP register, right shifts each result by an immediate value, puts the final result into a vector, and writes the vector to the lower or upper half of the destination SIMD and FP register. The destination vector elements are half as long as the source vector elements. The results are truncated. For rounded results, see *20.138 RSHRN, RSHRN2 (vector)* on page 20-1495.

The `RSHRN` instruction writes the vector to the lower half of the destination register and clears the upper half, while the `RSHRN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-45 SHRN, SHRN2 (Vector) specifier combinations

<Q>	*Tb*	*Ta*	*shift*
-	8B	8H	1 to 8
2	16B	8H	1 to 8
-	4H	4S	1 to 16
2	8H	4S	1 to 16
-	2S	2D	1 to 32
2	4S	2D	1 to 32

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.155 SHSUB (vector)

Signed Halving Subtract.

Syntax

`SHSUB Vd.T, Vn.T, Vm.T`

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of `8B`, `16B`, `4H`, `8H`, `2S` or `4S`.

Vn
Is the name of the first SIMD and FP source register.

Vm
Is the name of the second SIMD and FP source register.

Usage

Signed Halving Subtract. This instruction subtracts the elements in the vector in the second source SIMD and FP register from the corresponding elements in the vector in the first source SIMD and FP register, shifts each result right one bit, places each result into elements of a vector, and writes the vector to the destination SIMD and FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.156 SLI (vector)

Shift Left and Insert (immediate).

Syntax

```
SLI Vd.T, Vn.T, #shift
```

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of the values shown in Usage.

Vn
Is the name of the SIMD and FP source register.

shift
Is the left shift amount, in the range 0 to the element width in bits minus 1, and can be one of the values shown in Usage.

Usage

Shift Left and Insert (immediate). This instruction reads each vector element in the source SIMD and FP register, left shifts each vector element by an immediate value, and inserts the result into the corresponding vector element in the destination SIMD and FP register such that the new zero bits created by the shift are not inserted but retain their existing value. Bits shifted out of the left of each vector element in the source register are lost.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-46 SLI (Vector) specifier combinations

T	*shift*
8B	0 to 7
16B	0 to 7
4H	0 to 15
8H	0 to 15
2S	0 to 31
4S	0 to 31
2D	0 to 63

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.157 SMAX (vector)

Signed Maximum (vector).

Syntax

`SMAX` *`Vd.T, Vn.T, Vm.T`*

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of `8B`, `16B`, `4H`, `8H`, `2S` or `4S`.

Vn
Is the name of the first SIMD and FP source register.

Vm
Is the name of the second SIMD and FP source register.

Usage

Signed Maximum (vector). This instruction compares corresponding elements in the vectors in the two source SIMD and FP registers, places the larger of each pair of signed integer values into a vector, and writes the vector to the destination SIMD and FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.158 SMAXP (vector)

Signed Maximum Pairwise.

Syntax

`SMAXP` *`Vd.T, Vn.T, Vm.T`*

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of `8B`, `16B`, `4H`, `8H`, `2S` or `4S`.

Vn
Is the name of the first SIMD and FP source register.

Vm
Is the name of the second SIMD and FP source register.

Usage

Signed Maximum Pairwise. This instruction creates a vector by concatenating the vector elements of the first source SIMD and FP register after the vector elements of the second source SIMD and FP register, reads each pair of adjacent vector elements in the two source SIMD and FP registers, writes the largest of each pair of signed integer values into a vector, and writes the vector to the destination SIMD and FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.159 SMAXV (vector)

Signed Maximum across Vector.

Syntax

```
SMAXV Vd, Vn.T
```

Where:

V
Is the destination width specifier, and can be one of the values shown in Usage.

d
Is the number of the SIMD and FP destination register.

Vn
Is the name of the SIMD and FP source register.

T
Is an arrangement specifier, and can be one of the values shown in Usage.

Usage

Signed Maximum across Vector. This instruction compares all the vector elements in the source SIMD and FP register, and writes the largest of the values as a scalar to the destination SIMD and FP register. All the values in this instruction are signed integer values.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-47 SMAXV (Vector) specifier combinations

V	*T*
B	8B
B	16B
H	4H
H	8H
S	4S

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.160 SMIN (vector)

Signed Minimum (vector).

Syntax

`SMIN` *`Vd.T, Vn.T, Vm.T`*

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of `8B`, `16B`, `4H`, `8H`, `2S` or `4S`.

Vn
Is the name of the first SIMD and FP source register.

Vm
Is the name of the second SIMD and FP source register.

Usage

Signed Minimum (vector). This instruction compares corresponding elements in the vectors in the two source SIMD and FP registers, places the smaller of each of the two signed integer values into a vector, and writes the vector to the destination SIMD and FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.161 SMINP (vector)

Signed Minimum Pairwise.

Syntax

`SMINP` *`Vd.T, Vn.T, Vm.T`*

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of `8B`, `16B`, `4H`, `8H`, `2S` or `4S`.

Vn
Is the name of the first SIMD and FP source register.

Vm
Is the name of the second SIMD and FP source register.

Usage

Signed Minimum Pairwise. This instruction creates a vector by concatenating the vector elements of the first source SIMD and FP register after the vector elements of the second source SIMD and FP register, reads each pair of adjacent vector elements in the two source SIMD and FP registers, writes the smallest of each pair of signed integer values into a vector, and writes the vector to the destination SIMD and FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.162 SMINV (vector)

Signed Minimum across Vector.

Syntax

`SMINV Vd, Vn.T`

Where:

V
Is the destination width specifier, and can be one of the values shown in Usage.

d
Is the number of the SIMD and FP destination register.

Vn
Is the name of the SIMD and FP source register.

T
Is an arrangement specifier, and can be one of the values shown in Usage.

Usage

Signed Minimum across Vector. This instruction compares all the vector elements in the source SIMD and FP register, and writes the smallest of the values as a scalar to the destination SIMD and FP register. All the values in this instruction are signed integer values.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-48 SMINV (Vector) specifier combinations

V	*T*
B	8B
B	16B
H	4H
H	8H
S	4S

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.163 SMLAL, SMLAL2 (vector, by element)

Signed Multiply-Add Long (vector, by element).

Syntax

`SMLAL{2} Vd.Ta, Vn.Tb, Vm.Ts[index]`

Where:

2
Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements. See <Q> in the Usage table.

Vd
Is the name of the SIMD and FP destination register.

Ta
Is an arrangement specifier, and can be either `4S` or `2D`.

Vn
Is the name of the first SIMD and FP source register.

Tb
Is an arrangement specifier, and can be one of the values shown in Usage.

Vm
Is the name of the second SIMD and FP source register:
- If *Ts* is `H`, then *Vm* must be in the range V0 to V15.
- If *Ts* is `S`, then *Vm* must be in the range V0 to V31.

Ts
Is an element size specifier, and can be either `H` or `S`.

index
Is the element index, in the range shown in Usage.

Usage

Signed Multiply-Add Long (vector, by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD and FP register by the specified vector element in the second source SIMD and FP register, and accumulates the results with the vector elements of the destination SIMD and FP register. The destination vector elements are twice as long as the elements that are multiplied. All the values in this instruction are signed integer values.

The `SMLAL` instruction extracts vector elements from the lower half of the first source register, while the `SMLAL2` instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-49 SMLAL, SMLAL2 (Vector) specifier combinations

<Q>	*Ta*	*Tb*	*Ts*	*index*
-	4S	4H	H	0 to 7
2	4S	8H	H	0 to 7
-	2D	2S	S	0 to 3
2	2D	4S	S	0 to 3

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.164 SMLAL, SMLAL2 (vector)

Signed Multiply-Add Long (vector).

Syntax

`SMLAL{2} Vd.Ta, Vn.Tb, Vm.Tb`

Where:

2
Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements. See <Q> in the Usage table.

Vd
Is the name of the SIMD and FP destination register.

Ta
Is an arrangement specifier, and can be one of the values shown in Usage.

Vn
Is the name of the first SIMD and FP source register.

Tb
Is an arrangement specifier, and can be one of the values shown in Usage.

Vm
Is the name of the second SIMD and FP source register.

Usage

Signed Multiply-Add Long (vector). This instruction multiplies corresponding signed integer values in the lower or upper half of the vectors of the two source SIMD and FP registers, and accumulates the results with the vector elements of the destination SIMD and FP register. The destination vector elements are twice as long as the elements that are multiplied.

The `SMLAL` instruction extracts each source vector from the lower half of each source register, while the `SMLAL2` instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-50 SMLAL, SMLAL2 (Vector) specifier combinations

<Q>	*Ta*	*Tb*
-	8H	8B
2	8H	16B
-	4S	4H
2	4S	8H
-	2D	2S
2	2D	4S

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.165 SMLSL, SMLSL2 (vector, by element)

Signed Multiply-Subtract Long (vector, by element).

Syntax

`SMLSL{2} Vd.Ta, Vn.Tb, Vm.Ts[index]`

Where:

2
Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements. See <Q> in the Usage table.

Vd
Is the name of the SIMD and FP destination register.

Ta
Is an arrangement specifier, and can be either `4S` or `2D`.

Vn
Is the name of the first SIMD and FP source register.

Tb
Is an arrangement specifier, and can be one of the values shown in Usage.

Vm
Is the name of the second SIMD and FP source register:
- If *Ts* is `H`, then *Vm* must be in the range V0 to V15.
- If *Ts* is `S`, then *Vm* must be in the range V0 to V31.

Ts
Is an element size specifier, and can be either `H` or `S`.

index
Is the element index, in the range shown in Usage.

Usage

Signed Multiply-Subtract Long (vector, by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD and FP register by the specified vector element of the second source SIMD and FP register and subtracts the results from the vector elements of the destination SIMD and FP register. The destination vector elements are twice as long as the elements that are multiplied.

The `SMLSL` instruction extracts vector elements from the lower half of the first source register, while the `SMLSL2` instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-51 SMLSL, SMLSL2 (Vector) specifier combinations

<Q>	*Ta*	*Tb*	*Ts*	*index*
-	4S	4H	H	0 to 7
2	4S	8H	H	0 to 7
-	2D	2S	S	0 to 3
2	2D	4S	S	0 to 3

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.166 SMLSL, SMLSL2 (vector)

Signed Multiply-Subtract Long (vector).

Syntax

`SMLSL{2} Vd.Ta, Vn.Tb, Vm.Tb`

Where:

2
Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements. See <Q> in the Usage table.

Vd
Is the name of the SIMD and FP destination register.

Ta
Is an arrangement specifier, and can be one of the values shown in Usage.

Vn
Is the name of the first SIMD and FP source register.

Tb
Is an arrangement specifier, and can be one of the values shown in Usage.

Vm
Is the name of the second SIMD and FP source register.

Usage

Signed Multiply-Subtract Long (vector). This instruction multiplies corresponding signed integer values in the lower or upper half of the vectors of the two source SIMD and FP registers, and subtracts the results from the vector elements of the destination SIMD and FP register. The destination vector elements are twice as long as the elements that are multiplied.

The `SMLSL` instruction extracts each source vector from the lower half of each source register, while the `SMLSL2` instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-52 SMLSL, SMLSL2 (Vector) specifier combinations

<Q>	*Ta*	*Tb*
-	8H	8B
2	8H	16B
-	4S	4H
2	4S	8H
-	2D	2S
2	2D	4S

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.167 SMOV (vector)

Signed Move vector element to general-purpose register.

Syntax

```
SMOV Wd, Vn.Ts[index] ; 32-bit
SMOV Xd, Vn.Ts[index] ; 64-bit
```

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

Ts
Is an element size specifier:

32-bit
Can be one of B or H.

64-bit
Can be one of B, H or S.

index
Is the element index, in the range shown in Usage.

Xd
Is the 64-bit name of the general-purpose destination register.

Vn
Is the name of the SIMD and FP source register.

Usage

Signed Move vector element to general-purpose register. This instruction reads the signed integer from the source SIMD and FP register, sign-extends it to form a 32-bit or 64-bit value, and writes the result to destination general-purpose register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following tables show valid specifier combinations:

Table 20-53 SMOV (32-bit) specifier combinations

Ts	*index*
B	0 to 15
H	0 to 7

Table 20-54 SMOV (64-bit) specifier combinations

Ts	*index*
B	0 to 15
H	0 to 7
S	0 to 3

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.168 SMULL, SMULL2 (vector, by element)

Signed Multiply Long (vector, by element).

Syntax

`SMULL{2} Vd.Ta, Vn.Tb, Vm.Ts[index]`

Where:

2
Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements. See <Q> in the Usage table.

Vd
Is the name of the SIMD and FP destination register.

Ta
Is an arrangement specifier, and can be either `4S` or `2D`.

Vn
Is the name of the first SIMD and FP source register.

Tb
Is an arrangement specifier, and can be one of the values shown in Usage.

Vm
Is the name of the second SIMD and FP source register:
- If *Ts* is `H`, then *Vm* must be in the range V0 to V15.
- If *Ts* is `S`, then *Vm* must be in the range V0 to V31.

Ts
Is an element size specifier, and can be either `H` or `S`.

index
Is the element index, in the range shown in Usage.

Usage

Signed Multiply Long (vector, by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD and FP register by the specified vector element of the second source SIMD and FP register, places the result in a vector, and writes the vector to the destination SIMD and FP register. The destination vector elements are twice as long as the elements that are multiplied.

The `SMULL` instruction extracts vector elements from the lower half of the first source register, while the `SMULL2` instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-55 SMULL, SMULL2 (Vector) specifier combinations

<Q>	*Ta*	*Tb*	*Ts*	*index*
-	4S	4H	H	0 to 7
2	4S	8H	H	0 to 7
-	2D	2S	S	0 to 3
2	2D	4S	S	0 to 3

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.169 SMULL, SMULL2 (vector)

Signed Multiply Long (vector).

Syntax

`SMULL{2} Vd.Ta, Vn.Tb, Vm.Tb`

Where:

2
Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements. See <Q> in the Usage table.

Vd
Is the name of the SIMD and FP destination register.

Ta
Is an arrangement specifier, and can be one of the values shown in Usage.

Vn
Is the name of the first SIMD and FP source register.

Tb
Is an arrangement specifier, and can be one of the values shown in Usage.

Vm
Is the name of the second SIMD and FP source register.

Usage

Signed Multiply Long (vector). This instruction multiplies corresponding signed integer values in the lower or upper half of the vectors of the two source SIMD and FP registers, places the results in a vector, and writes the vector to the destination SIMD and FP register.

The destination vector elements are twice as long as the elements that are multiplied.

The `SMULL` instruction extracts each source vector from the lower half of each source register, while the `SMULL2` instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-56 SMULL, SMULL2 (Vector) specifier combinations

<Q>	*Ta*	*Tb*
-	8H	8B
2	8H	16B
-	4S	4H
2	4S	8H
-	2D	2S
2	2D	4S

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.170 SQABS (vector)

Signed saturating Absolute value.

Syntax

`SQABS Vd.T, Vn.T`

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of `8B`, `16B`, `4H`, `8H`, `2S`, `4S` or `2D`.

Vn
Is the name of the SIMD and FP source register.

Usage

Signed saturating Absolute value. This instruction reads each vector element from the source SIMD and FP register, puts the absolute value of the result into a vector, and writes the vector to the destination SIMD and FP register. All the values in this instruction are signed integer values.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.171 SQADD (vector)

Signed saturating Add.

Syntax

`SQADD Vd.T, Vn.T, Vm.T`

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of `8B`, `16B`, `4H`, `8H`, `2S`, `4S` or `2D`.

Vn
Is the name of the first SIMD and FP source register.

Vm
Is the name of the second SIMD and FP source register.

Usage

Signed saturating Add. This instruction adds the values of corresponding elements of the two source SIMD and FP registers, places the results into a vector, and writes the vector to the destination SIMD and FP register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.172 SQDMLAL, SQDMLAL2 (vector, by element)

Signed saturating Doubling Multiply-Add Long (by element).

Syntax

`SQDMLAL{2}` *`Vd.Ta, Vn.Tb, Vm.Ts`*`[`*`index`*`]`

Where:

`2`
Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements. See <Q> in the Usage table.

`Vd`
Is the name of the SIMD and FP destination register.

`Ta`
Is an arrangement specifier, and can be either `4S` or `2D`.

`Vn`
Is the name of the first SIMD and FP source register.

`Tb`
Is an arrangement specifier, and can be one of the values shown in Usage.

`Vm`
Is the name of the second SIMD and FP source register:
- If *`Ts`* is `H`, then *`Vm`* must be in the range V0 to V15.
- If *`Ts`* is `S`, then *`Vm`* must be in the range V0 to V31.

`Ts`
Is an element size specifier, and can be either `H` or `S`.

`index`
Is the element index, in the range shown in Usage.

Usage

Signed saturating Doubling Multiply-Add Long (by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD and FP register by the specified vector element of the second source SIMD and FP register, doubles the results, and accumulates the final results with the vector elements of the destination SIMD and FP register. The destination vector elements are twice as long as the elements that are multiplied.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

The `SQDMLAL` instruction extracts vector elements from the lower half of the first source register, while the `SQDMLAL2` instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-57 SQDMLAL{2} (Vector) specifier combinations

<Q>	*`Ta`*	*`Tb`*	*`Ts`*	*`index`*
-	4S	4H	H	0 to 7
2	4S	8H	H	0 to 7
-	2D	2S	S	0 to 3
2	2D	4S	S	0 to 3

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.173 SQDMLAL, SQDMLAL2 (vector)

Signed saturating Doubling Multiply-Add Long.

Syntax

`SQDMLAL{2} Vd.Ta, Vn.Tb, Vm.Tb`

Where:

2
Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements. See <Q> in the Usage table.

Vd
Is the name of the SIMD and FP destination register.

Ta
Is an arrangement specifier, and can be either `4S` or `2D`.

Vn
Is the name of the first SIMD and FP source register.

Tb
Is an arrangement specifier, and can be one of the values shown in Usage.

Vm
Is the name of the second SIMD and FP source register.

Usage

Signed saturating Doubling Multiply-Add Long. This instruction multiplies corresponding signed integer values in the lower or upper half of the vectors of the two source SIMD and FP registers, doubles the results, and accumulates the final results with the vector elements of the destination SIMD and FP register. The destination vector elements are twice as long as the elements that are multiplied.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

The `SQDMLAL` instruction extracts each source vector from the lower half of each source register, while the `SQDMLAL2` instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-58 SQDMLAL{2} (Vector) specifier combinations

<Q>	*Ta*	*Tb*
-	4S	4H
2	4S	8H
-	2D	2S
2	2D	4S

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.174 SQDMLSL, SQDMLSL2 (vector, by element)

Signed saturating Doubling Multiply-Subtract Long (by element).

Syntax

`SQDMLSL{2}` *`Vd.Ta, Vn.Tb, Vm.Ts`*`[`*`index`*`]`

Where:

`2`
Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements. See <Q> in the Usage table.

`Vd`
Is the name of the SIMD and FP destination register.

`Ta`
Is an arrangement specifier, and can be either `4S` or `2D`.

`Vn`
Is the name of the first SIMD and FP source register.

`Tb`
Is an arrangement specifier, and can be one of the values shown in Usage.

`Vm`
Is the name of the second SIMD and FP source register:
- If *`Ts`* is `H`, then *`Vm`* must be in the range V0 to V15.
- If *`Ts`* is `S`, then *`Vm`* must be in the range V0 to V31.

`Ts`
Is an element size specifier, and can be either `H` or `S`.

`index`
Is the element index, in the range shown in Usage.

Usage

Signed saturating Doubling Multiply-Subtract Long (by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD and FP register by the specified vector element of the second source SIMD and FP register, doubles the results, and subtracts the final results from the vector elements of the destination SIMD and FP register. The destination vector elements are twice as long as the elements that are multiplied. All the values in this instruction are signed integer values.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

The `SQDMLSL` instruction extracts vector elements from the lower half of the first source register, while the `SQDMLSL2` instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-59 SQDMLSL{2} (Vector) specifier combinations

<Q>	*Ta*	*Tb*	*Ts*	*index*
-	4S	4H	H	0 to 7
2	4S	8H	H	0 to 7
-	2D	2S	S	0 to 3
2	2D	4S	S	0 to 3

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.175 SQDMLSL, SQDMLSL2 (vector)

Signed saturating Doubling Multiply-Subtract Long.

Syntax

`SQDMLSL{2} Vd.Ta, Vn.Tb, Vm.Tb`

Where:

2
Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements. See <Q> in the Usage table.

Vd
Is the name of the SIMD and FP destination register.

Ta
Is an arrangement specifier, and can be either `4S` or `2D`.

Vn
Is the name of the first SIMD and FP source register.

Tb
Is an arrangement specifier, and can be one of the values shown in Usage.

Vm
Is the name of the second SIMD and FP source register.

Usage

Signed saturating Doubling Multiply-Subtract Long. This instruction multiplies corresponding signed integer values in the lower or upper half of the vectors of the two source SIMD and FP registers, doubles the results, and subtracts the final results from the vector elements of the destination SIMD and FP register. The destination vector elements are twice as long as the elements that are multiplied.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

The `SQDMLSL` instruction extracts each source vector from the lower half of each source register, while the `SQDMLSL2` instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-60 SQDMLSL{2} (Vector) specifier combinations

<Q>	*Ta*	*Tb*
-	4S	4H
2	4S	8H
-	2D	2S
2	2D	4S

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.176 SQDMULH (vector, by element)

Signed saturating Doubling Multiply returning High half (by element).

Syntax

```
SQDMULH Vd.T, Vn.T, Vm.Ts[index]
```

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of the values shown in Usage.

Vn
Is the name of the first SIMD and FP source register.

Vm
Is the name of the second SIMD and FP source register:

- If *Ts* is H, then *Vm* must be in the range V0 to V15.
- If *Ts* is S, then *Vm* must be in the range V0 to V31.

Ts
Is an element size specifier, and can be either H or S.

index
Is the element index, in the range shown in Usage.

Usage

Signed saturating Doubling Multiply returning High half (by element). This instruction multiplies each vector element in the first source SIMD and FP register by the specified vector element of the second source SIMD and FP register, doubles the results, places the most significant half of the final results into a vector, and writes the vector to the destination SIMD and FP register.

The results are truncated. For rounded results, see *20.185 SQRDMULH (vector, by element)* on page 20-1545.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-61 SQDMULH (Vector) specifier combinations

T	*Ts*	*index*
4H	H	0 to 7
8H	H	0 to 7
2S	S	0 to 3
4S	S	0 to 3

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.177 SQDMULH (vector)

Signed saturating Doubling Multiply returning High half.

Syntax

`SQDMULH` *`Vd.T, Vn.T, Vm.T`*

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of `4H`, `8H`, `2S` or `4S`.

Vn
Is the name of the first SIMD and FP source register.

Vm
Is the name of the second SIMD and FP source register.

Usage

Signed saturating Doubling Multiply returning High half. This instruction multiplies the values of corresponding elements of the two source SIMD and FP registers, doubles the results, places the most significant half of the final results into a vector, and writes the vector to the destination SIMD and FP register.

The results are truncated. For rounded results, see *20.186 SQRDMULH (vector)* on page 20-1546.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.178 SQDMULL, SQDMULL2 (vector, by element)

Signed saturating Doubling Multiply Long (by element).

Syntax

`SQDMULL{2}` *`Vd.Ta, Vn.Tb, Vm.Ts`*`[`*`index`*`]`

Where:

`2`
Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements. See <Q> in the Usage table.

`Vd`
Is the name of the SIMD and FP destination register.

`Ta`
Is an arrangement specifier, and can be either `4S` or `2D`.

`Vn`
Is the name of the first SIMD and FP source register.

`Tb`
Is an arrangement specifier, and can be one of the values shown in Usage.

`Vm`
Is the name of the second SIMD and FP source register:
- If *`Ts`* is `H`, then *`Vm`* must be in the range V0 to V15.
- If *`Ts`* is `S`, then *`Vm`* must be in the range V0 to V31.

`Ts`
Is an element size specifier, and can be either `H` or `S`.

`index`
Is the element index, in the range shown in Usage.

Usage

Signed saturating Doubling Multiply Long (by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD and FP register by the specified vector element of the second source SIMD and FP register, doubles the results, places the final results in a vector, and writes the vector to the destination SIMD and FP register. All the values in this instruction are signed integer values.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

The `SQDMULL` instruction extracts the first source vector from the lower half of the first source register, while the `SQDMULL2` instruction extracts the first source vector from the upper half of the first source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-62 SQDMULL{2} (Vector) specifier combinations

<Q>	*Ta*	*Tb*	*Ts*	*index*
-	4S	4H	H	0 to 7
2	4S	8H	H	0 to 7
-	2D	2S	S	0 to 3
2	2D	4S	S	0 to 3

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.179 SQDMULL, SQDMULL2 (vector)

Signed saturating Doubling Multiply Long.

Syntax

`SQDMULL{2} Vd.Ta, Vn.Tb, Vm.Tb`

Where:

2
Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements. See <Q> in the Usage table.

Vd
Is the name of the SIMD and FP destination register.

Ta
Is an arrangement specifier, and can be either `4S` or `2D`.

Vn
Is the name of the first SIMD and FP source register.

Tb
Is an arrangement specifier, and can be one of the values shown in Usage.

Vm
Is the name of the second SIMD and FP source register.

Usage

Signed saturating Doubling Multiply Long. This instruction multiplies corresponding vector elements in the lower or upper half of the two source SIMD and FP registers, doubles the results, places the final results in a vector, and writes the vector to the destination SIMD and FP register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

The `SQDMULL` instruction extracts each source vector from the lower half of each source register, while the `SQDMULL2` instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-63 SQDMULL{2} (Vector) specifier combinations

<Q>	*Ta*	*Tb*
-	4S	4H
2	4S	8H
-	2D	2S
2	2D	4S

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.180 SQNEG (vector)

Signed saturating Negate.

Syntax

`SQNEG` *`Vd.T, Vn.T`*

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of `8B`, `16B`, `4H`, `8H`, `2S`, `4S` or `2D`.

Vn
Is the name of the SIMD and FP source register.

Usage

Signed saturating Negate. This instruction reads each vector element from the source SIMD and FP register, negates each value, places the result into a vector, and writes the vector to the destination SIMD and FP register. All the values in this instruction are signed integer values.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.181 SQRDMLAH (vector, by element)

Signed Saturating Rounding Doubling Multiply Accumulate returning High Half (by element).

Syntax

```
SQRDMLAH Vd.T, Vn.T, Vm.Ts[index]
```

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of the values shown in Usage.

Vn
Is the name of the first SIMD and FP source register.

Vm
Is the name of the second SIMD and FP source register:
- If *Ts* is H, then *Vm* must be in the range V0 to V15.
- If *Ts* is S, then *Vm* must be in the range V0 to V31.

Ts
Is an element size specifier, and can be either H or S.

index
Is the element index, in the range shown in Usage.

Architectures supported (vector)

Supported in the Armv8.1 architecture and later.

Usage

Signed Saturating Rounding Doubling Multiply Accumulate returning High Half (by element). This instruction multiplies the vector elements of the first source SIMD and FP register with the value of a vector element of the second source SIMD and FP register without saturating the multiply results, doubles the results, and accumulates the most significant half of the final results with the vector elements of the destination SIMD and FP register. The results are rounded.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSR.QC, is set if saturation occurs.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-64 SQRDMLAH (Vector) specifier combinations

T	*Ts*	*index*
4H	H	0 to 7
8H	H	0 to 7
2S	S	0 to 3
4S	S	0 to 3

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.182 SQRDMLAH (vector)

Signed Saturating Rounding Doubling Multiply Accumulate returning High Half (vector).

Syntax

`SQRDMLAH` *`Vd.T, Vn.T, Vm.T`*

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of `4H`, `8H`, `2S` or `4S`.

Vn
Is the name of the first SIMD and FP source register.

Vm
Is the name of the second SIMD and FP source register.

Architectures supported (vector)

Supported in the Armv8.1 architecture and later.

Usage

Signed Saturating Rounding Doubling Multiply Accumulate returning High Half (vector). This instruction multiplies the vector elements of the first source SIMD and FP register with the corresponding vector elements of the second source SIMD and FP register without saturating the multiply results, doubles the results, and accumulates the most significant half of the final results with the vector elements of the destination SIMD and FP register. The results are rounded.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSR.QC, is set if saturation occurs.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.183 SQRDMLSH (vector, by element)

Signed Saturating Rounding Doubling Multiply Subtract returning High Half (by element).

Syntax

```
SQRDMLSH Vd.T, Vn.T, Vm.Ts[index]
```

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of the values shown in Usage.

Vn
Is the name of the first SIMD and FP source register.

Vm
Is the name of the second SIMD and FP source register:

- If *Ts* is H, then *Vm* must be in the range V0 to V15.
- If *Ts* is S, then *Vm* must be in the range V0 to V31.

Ts
Is an element size specifier, and can be either H or S.

index
Is the element index, in the range shown in Usage.

Architectures supported (vector)

Supported in the Armv8.1 architecture and later.

Usage

Signed Saturating Rounding Doubling Multiply Subtract returning High Half (by element). This instruction multiplies the vector elements of the first source SIMD and FP register with the value of a vector element of the second source SIMD and FP register without saturating the multiply results, doubles the results, and subtracts the most significant half of the final results from the vector elements of the destination SIMD and FP register. The results are rounded.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSR.QC, is set if saturation occurs.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-65 SQRDMLSH (Vector) specifier combinations

T	*Ts*	*index*
4H	H	0 to 7
8H	H	0 to 7
2S	S	0 to 3
4S	S	0 to 3

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.184 SQRDMLSH (vector)

Signed Saturating Rounding Doubling Multiply Subtract returning High Half (vector).

Syntax

`SQRDMLSH` *`Vd.T, Vn.T, Vm.T`*

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of `4H`, `8H`, `2S` or `4S`.

Vn
Is the name of the first SIMD and FP source register.

Vm
Is the name of the second SIMD and FP source register.

Architectures supported (vector)

Supported in the Armv8.1 architecture and later.

Usage

Signed Saturating Rounding Doubling Multiply Subtract returning High Half (vector). This instruction multiplies the vector elements of the first source SIMD and FP register with the corresponding vector elements of the second source SIMD and FP register without saturating the multiply results, doubles the results, and subtracts the most significant half of the final results from the vector elements of the destination SIMD and FP register. The results are rounded.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSR.QC, is set if saturation occurs.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.185 SQRDMULH (vector, by element)

Signed saturating Rounding Doubling Multiply returning High half (by element).

Syntax

```
SQRDMULH Vd.T, Vn.T, Vm.Ts[index]
```

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of the values shown in Usage.

Vn
Is the name of the first SIMD and FP source register.

Vm
Is the name of the second SIMD and FP source register:

- If *Ts* is H, then *Vm* must be in the range V0 to V15.
- If *Ts* is S, then *Vm* must be in the range V0 to V31.

Ts
Is an element size specifier, and can be either H or S.

index
Is the element index, in the range shown in Usage.

Usage

Signed saturating Rounding Doubling Multiply returning High half (by element). This instruction multiplies each vector element in the first source SIMD and FP register by the specified vector element of the second source SIMD and FP register, doubles the results, places the most significant half of the final results into a vector, and writes the vector to the destination SIMD and FP register.

The results are rounded. For truncated results, see *20.176 SQDMULH (vector, by element)* on page 20-1535.

If any of the results overflows, they are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-66 SQRDMULH (Vector) specifier combinations

T	*Ts*	*index*
4H	H	0 to 7
8H	H	0 to 7
2S	S	0 to 3
4S	S	0 to 3

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.186 SQRDMULH (vector)

Signed saturating Rounding Doubling Multiply returning High half.

Syntax

`SQRDMULH Vd.T, Vn.T, Vm.T`

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of `4H`, `8H`, `2S` or `4S`.

Vn
Is the name of the first SIMD and FP source register.

Vm
Is the name of the second SIMD and FP source register.

Usage

Signed saturating Rounding Doubling Multiply returning High half. This instruction multiplies the values of corresponding elements of the two source SIMD and FP registers, doubles the results, places the most significant half of the final results into a vector, and writes the vector to the destination SIMD and FP register.

The results are rounded. For truncated results, see *20.177 SQDMULH (vector)* on page 20-1536.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.187 SQRSHL (vector)

Signed saturating Rounding Shift Left (register).

Syntax

`SQRSHL Vd.T, Vn.T, Vm.T`

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of `8B`, `16B`, `4H`, `8H`, `2S`, `4S` or `2D`.

Vn
Is the name of the first SIMD and FP source register.

Vm
Is the name of the second SIMD and FP source register.

Usage

Signed saturating Rounding Shift Left (register). This instruction takes each vector element in the first source SIMD and FP register, shifts it by a value from the least significant byte of the corresponding vector element of the second source SIMD and FP register, places the results into a vector, and writes the vector to the destination SIMD and FP register.

If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift. The results are rounded. For truncated results, see *20.191 SQSHL (vector, register)* on page 20-1551.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.188 SQRSHRN, SQRSHRN2 (vector)

Signed saturating Rounded Shift Right Narrow (immediate).

Syntax

`SQRSHRN{2} Vd.Tb, Vn.Ta, #shift`

Where:

2
Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements. See <Q> in the Usage table.

Vd
Is the name of the SIMD and FP destination register.

Tb
Is an arrangement specifier, and can be one of the values shown in Usage.

Vn
Is the name of the SIMD and FP source register.

Ta
Is an arrangement specifier, and can be one of the values shown in Usage.

shift
Is the right shift amount, in the range 1 to the destination element width in bits, and can be one of the values shown in Usage.

Usage

Signed saturating Rounded Shift Right Narrow (immediate). This instruction reads each vector element in the source SIMD and FP register, right shifts each result by an immediate value, saturates each shifted result to a value that is half the original width, puts the final result into a vector, and writes the vector to the lower or upper half of the destination SIMD and FP register. All the values in this instruction are signed integer values. The destination vector elements are half as long as the source vector elements. The results are rounded. For truncated results, see *20.193 SQSHRN, SQSHRN2 (vector)* on page 20-1553.

The `SQRSHRN` instruction writes the vector to the lower half of the destination register and clears the upper half, while the `SQRSHRN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-67 SQRSHRN{2} (Vector) specifier combinations

<Q>	*Tb*	*Ta*	*shift*
-	8B	8H	1 to 8
2	16B	8H	1 to 8
-	4H	4S	1 to 16
2	8H	4S	1 to 16
-	2S	2D	1 to 32
2	4S	2D	1 to 32

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.189 SQRSHRUN, SQRSHRUN2 (vector)

Signed saturating Rounded Shift Right Unsigned Narrow (immediate).

Syntax

`SQRSHRUN{2} Vd.Tb, Vn.Ta, #shift`

Where:

2
Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements. See <Q> in the Usage table.

Vd
Is the name of the SIMD and FP destination register.

Tb
Is an arrangement specifier, and can be one of the values shown in Usage.

Vn
Is the name of the SIMD and FP source register.

Ta
Is an arrangement specifier, and can be one of the values shown in Usage.

shift
Is the right shift amount, in the range 1 to the destination element width in bits, and can be one of the values shown in Usage.

Usage

Signed saturating Rounded Shift Right Unsigned Narrow (immediate). This instruction reads each signed integer value in the vector of the source SIMD and FP register, right shifts each value by an immediate value, saturates the result to an unsigned integer value that is half the original width, places the final result into a vector, and writes the vector to the destination SIMD and FP register. The results are rounded. For truncated results, see *20.194 SQSHRUN, SQSHRUN2 (vector)* on page 20-1554.

The `SQRSHRUN` instruction writes the vector to the lower half of the destination register and clears the upper half, while the `SQRSHRUN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-68 SQRSHRUN{2} (Vector) specifier combinations

<Q>	*Tb*	*Ta*	*shift*
-	8B	8H	1 to 8
2	16B	8H	1 to 8
-	4H	4S	1 to 16
2	8H	4S	1 to 16
-	2S	2D	1 to 32
2	4S	2D	1 to 32

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.190 SQSHL (vector, immediate)

Signed saturating Shift Left (immediate).

Syntax

```
SQSHL Vd.T, Vn.T, #shift
```

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of the values shown in Usage.

Vn
Is the name of the SIMD and FP source register.

shift
Is the left shift amount, in the range 0 to the element width in bits minus 1, and can be one of the values shown in Usage.

Usage

Signed saturating Shift Left (immediate). This instruction reads each vector element in the source SIMD and FP register, shifts each result by an immediate value, places the final result in a vector, and writes the vector to the destination SIMD and FP register. The results are truncated. For rounded results, see *20.252 UQRSHL (vector)* on page 20-1615.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-69 SQSHL (Vector) specifier combinations

T	*shift*
8B	0 to 7
16B	0 to 7
4H	0 to 15
8H	0 to 15
2S	0 to 31
4S	0 to 31
2D	0 to 63

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.191 SQSHL (vector, register)

Signed saturating Shift Left (register).

Syntax

`SQSHL` *`Vd.T, Vn.T, Vm.T`*

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of `8B`, `16B`, `4H`, `8H`, `2S`, `4S` or `2D`.

Vn
Is the name of the first SIMD and FP source register.

Vm
Is the name of the second SIMD and FP source register.

Usage

Signed saturating Shift Left (register). This instruction takes each element in the vector of the first source SIMD and FP register, shifts each element by a value from the least significant byte of the corresponding element of the second source SIMD and FP register, places the results in a vector, and writes the vector to the destination SIMD and FP register.

If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift. The results are truncated. For rounded results, see *20.187 SQRSHL (vector)* on page 20-1547.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.192 SQSHLU (vector)

Signed saturating Shift Left Unsigned (immediate).

Syntax

```
SQSHLU Vd.T, Vn.T, #shift
```

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of the values shown in Usage.

Vn
Is the name of the SIMD and FP source register.

shift
Is the left shift amount, in the range 0 to the element width in bits minus 1, and can be one of the values shown in Usage.

Usage

Signed saturating Shift Left Unsigned (immediate). This instruction reads each signed integer value in the vector of the source SIMD and FP register, shifts each value by an immediate value, saturates the shifted result to an unsigned integer value, places the result in a vector, and writes the vector to the destination SIMD and FP register. The results are truncated. For rounded results, see *20.252 UQRSHL (vector)* on page 20-1615.

If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-70 SQSHLU (Vector) specifier combinations

T	*shift*
8B	0 to 7
16B	0 to 7
4H	0 to 15
8H	0 to 15
2S	0 to 31
4S	0 to 31
2D	0 to 63

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.193 SQSHRN, SQSHRN2 (vector)

Signed saturating Shift Right Narrow (immediate).

Syntax

`SQSHRN{2} Vd.Tb, Vn.Ta, #shift`

Where:

2
Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements. See <Q> in the Usage table.

Vd
Is the name of the SIMD and FP destination register.

Tb
Is an arrangement specifier, and can be one of the values shown in Usage.

Vn
Is the name of the SIMD and FP source register.

Ta
Is an arrangement specifier, and can be one of the values shown in Usage.

shift
Is the right shift amount, in the range 1 to the destination element width in bits, and can be one of the values shown in Usage.

Usage

Signed saturating Shift Right Narrow (immediate). This instruction reads each vector element in the source SIMD and FP register, right shifts and truncates each result by an immediate value, saturates each shifted result to a value that is half the original width, puts the final result into a vector, and writes the vector to the lower or upper half of the destination SIMD and FP register. All the values in this instruction are signed integer values. The destination vector elements are half as long as the source vector elements. For rounded results, see *20.188 SQRSHRN, SQRSHRN2 (vector)* on page 20-1548.

The `SQSHRN` instruction writes the vector to the lower half of the destination register and clears the upper half, while the `SQSHRN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-71 SQSHRN{2} (Vector) specifier combinations

<Q>	*Tb*	*Ta*	*shift*
-	8B	8H	1 to 8
2	16B	8H	1 to 8
-	4H	4S	1 to 16
2	8H	4S	1 to 16
-	2S	2D	1 to 32
2	4S	2D	1 to 32

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.194 SQSHRUN, SQSHRUN2 (vector)

Signed saturating Shift Right Unsigned Narrow (immediate).

Syntax

```
SQSHRUN{2} Vd.Tb, Vn.Ta, #shift
```

Where:

2
Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements. See <Q> in the Usage table.

Vd
Is the name of the SIMD and FP destination register.

Tb
Is an arrangement specifier, and can be one of the values shown in Usage.

Vn
Is the name of the SIMD and FP source register.

Ta
Is an arrangement specifier, and can be one of the values shown in Usage.

shift
Is the right shift amount, in the range 1 to the destination element width in bits, and can be one of the values shown in Usage.

Usage

Signed saturating Shift Right Unsigned Narrow (immediate). This instruction reads each signed integer value in the vector of the source SIMD and FP register, right shifts each value by an immediate value, saturates the result to an unsigned integer value that is half the original width, places the final result into a vector, and writes the vector to the destination SIMD and FP register. The results are truncated. For rounded results, see *20.189 SQRSHRUN, SQRSHRUN2 (vector)* on page 20-1549.

The `SQSHRUN` instruction writes the vector to the lower half of the destination register and clears the upper half, while the `SQSHRUN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-72 SQSHRUN{2} (Vector) specifier combinations

<Q>	*Tb*	*Ta*	*shift*
-	8B	8H	1 to 8
2	16B	8H	1 to 8
-	4H	4S	1 to 16
2	8H	4S	1 to 16
-	2S	2D	1 to 32
2	4S	2D	1 to 32

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.195 SQSUB (vector)

Signed saturating Subtract.

Syntax

`SQSUB Vd.T, Vn.T, Vm.T`

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of `8B`, `16B`, `4H`, `8H`, `2S`, `4S` or `2D`.

Vn
Is the name of the first SIMD and FP source register.

Vm
Is the name of the second SIMD and FP source register.

Usage

Signed saturating Subtract. This instruction subtracts the element values of the second source SIMD and FP register from the corresponding element values of the first source SIMD and FP register, places the results into a vector, and writes the vector to the destination SIMD and FP register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.196 SQXTN, SQXTN2 (vector)

Signed saturating extract Narrow.

Syntax

`SQXTN{2} Vd.Tb, Vn.Ta`

Where:

2
Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements. See <Q> in the Usage table.

Vd
Is the name of the SIMD and FP destination register.

Tb
Is an arrangement specifier, and can be one of the values shown in Usage.

Vn
Is the name of the SIMD and FP source register.

Ta
Is an arrangement specifier, and can be one of the values shown in Usage.

Usage

Signed saturating extract Narrow. This instruction reads each vector element from the source SIMD and FP register, saturates the value to half the original width, places the result into a vector, and writes the vector to the lower or upper half of the destination SIMD and FP register. The destination vector elements are half as long as the source vector elements. All the values in this instruction are signed integer values.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

The `SQXTN` instruction writes the vector to the lower half of the destination register and clears the upper half, while the `SQXTN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-73 SQXTN{2} (Vector) specifier combinations

<Q>	*Tb*	*Ta*
-	8B	8H
2	16B	8H
-	4H	4S
2	8H	4S
-	2S	2D
2	4S	2D

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.197 SQXTUN, SQXTUN2 (vector)

Signed saturating extract Unsigned Narrow.

Syntax

`SQXTUN{2} Vd.Tb, Vn.Ta`

Where:

2
Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements. See <Q> in the Usage table.

Vd
Is the name of the SIMD and FP destination register.

Tb
Is an arrangement specifier, and can be one of the values shown in Usage.

Vn
Is the name of the SIMD and FP source register.

Ta
Is an arrangement specifier, and can be one of the values shown in Usage.

Usage

Signed saturating extract Unsigned Narrow. This instruction reads each signed integer value in the vector of the source SIMD and FP register, saturates the value to an unsigned integer value that is half the original width, places the result into a vector, and writes the vector to the lower or upper half of the destination SIMD and FP register. The destination vector elements are half as long as the source vector elements.

If saturation occurs, the cumulative saturation bit FPSR.QC is set.

The `SQXTUN` instruction writes the vector to the lower half of the destination register and clears the upper half, while the `SQXTUN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-74 SQXTUN{2} (Vector) specifier combinations

<Q>	*Tb*	*Ta*
-	8B	8H
2	16B	8H
-	4H	4S
2	8H	4S
-	2S	2D
2	4S	2D

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.198 SRHADD (vector)

Signed Rounding Halving Add.

Syntax

`SRHADD Vd.T, Vn.T, Vm.T`

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of `8B`, `16B`, `4H`, `8H`, `2S` or `4S`.

Vn
Is the name of the first SIMD and FP source register.

Vm
Is the name of the second SIMD and FP source register.

Usage

Signed Rounding Halving Add. This instruction adds corresponding signed integer values from the two source SIMD and FP registers, shifts each result right one bit, places the results into a vector, and writes the vector to the destination SIMD and FP register.

The results are rounded. For truncated results, see *20.151 SHADD (vector)* on page 20-1508.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.199 SRI (vector)

Shift Right and Insert (immediate).

Syntax

```
SRI Vd.T, Vn.T, #shift
```

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of the values shown in Usage.

Vn
Is the name of the SIMD and FP source register.

shift
Is the right shift amount, in the range 1 to the element width in bits, and can be one of the values shown in Usage.

Usage

Shift Right and Insert (immediate). This instruction reads each vector element in the source SIMD and FP register, right shifts each vector element by an immediate value, and inserts the result into the corresponding vector element in the destination SIMD and FP register such that the new zero bits created by the shift are not inserted but retain their existing value. Bits shifted out of the right of each vector element of the source register are lost.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-75 SRI (Vector) specifier combinations

T	*shift*
8B	1 to 8
16B	1 to 8
4H	1 to 16
8H	1 to 16
2S	1 to 32
4S	1 to 32
2D	1 to 64

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.200 SRSHL (vector)

Signed Rounding Shift Left (register).

Syntax

`SRSHL` *`Vd.T, Vn.T, Vm.T`*

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of `8B`, `16B`, `4H`, `8H`, `2S`, `4S` or `2D`.

Vn
Is the name of the first SIMD and FP source register.

Vm
Is the name of the second SIMD and FP source register.

Usage

Signed Rounding Shift Left (register). This instruction takes each signed integer value in the vector of the first source SIMD and FP register, shifts it by a value from the least significant byte of the corresponding element of the second source SIMD and FP register, places the results in a vector, and writes the vector to the destination SIMD and FP register.

If the shift value is positive, the operation is a left shift. If the shift value is negative, it is a rounding right shift. For a truncating shift, see *20.203 SSHL (vector)* on page 20-1563.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.201 SRSHR (vector)

Signed Rounding Shift Right (immediate).

Syntax

```
SRSHR Vd.T, Vn.T, #shift
```

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of the values shown in Usage.

Vn
Is the name of the SIMD and FP source register.

shift
Is the right shift amount, in the range 1 to the element width in bits, and can be one of the values shown in Usage.

Usage

Signed Rounding Shift Right (immediate). This instruction reads each vector element in the source SIMD and FP register, right shifts each result by an immediate value, places the final result into a vector, and writes the vector to the destination SIMD and FP register. All the values in this instruction are signed integer values. The results are rounded. For truncated results, see *20.205 SSHR (vector)* on page 20-1565.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-76 SRSHR (Vector) specifier combinations

T	*shift*
8B	1 to 8
16B	1 to 8
4H	1 to 16
8H	1 to 16
2S	1 to 32
4S	1 to 32
2D	1 to 64

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.202 SRSRA (vector)

Signed Rounding Shift Right and Accumulate (immediate).

Syntax

```
SRSRA Vd.T, Vn.T, #shift
```

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of the values shown in Usage.

Vn
Is the name of the SIMD and FP source register.

shift
Is the right shift amount, in the range 1 to the element width in bits, and can be one of the values shown in Usage.

Usage

Signed Rounding Shift Right and Accumulate (immediate). This instruction reads each vector element in the source SIMD and FP register, right shifts each result by an immediate value, and accumulates the final results with the vector elements of the destination SIMD and FP register. All the values in this instruction are signed integer values. The results are rounded. For truncated results, see *20.206 SSRA (vector)* on page 20-1566.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-77 SRSRA (Vector) specifier combinations

T	*shift*
8B	1 to 8
16B	1 to 8
4H	1 to 16
8H	1 to 16
2S	1 to 32
4S	1 to 32
2D	1 to 64

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.203 SSHL (vector)

Signed Shift Left (register).

Syntax

`SSHL Vd.T, Vn.T, Vm.T`

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of `8B`, `16B`, `4H`, `8H`, `2S`, `4S` or `2D`.

Vn
Is the name of the first SIMD and FP source register.

Vm
Is the name of the second SIMD and FP source register.

Usage

Signed Shift Left (register). This instruction takes each signed integer value in the vector of the first source SIMD and FP register, shifts each value by a value from the least significant byte of the corresponding element of the second source SIMD and FP register, places the results in a vector, and writes the vector to the destination SIMD and FP register.

If the shift value is positive, the operation is a left shift. If the shift value is negative, it is a truncating right shift. For a rounding shift, see *20.200 SRSHL (vector)* on page 20-1560.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.204 SSHLL, SSHLL2 (vector)

Signed Shift Left Long (immediate).

This instruction is used by the alias `SXTL, SXTL2`, `SXTL, SXTL22`.

Syntax

`SSHLL{2} Vd.Ta, Vn.Tb, #shift`

Where:

2
Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements. See <Q> in the Usage table.

Vd
Is the name of the SIMD and FP destination register.

Ta
Is an arrangement specifier, and can be one of the values shown in Usage.

Vn
Is the name of the SIMD and FP source register.

Tb
Is an arrangement specifier, and can be one of the values shown in Usage.

shift
Is the left shift amount, in the range 0 to the source element width in bits minus 1, and can be one of the values shown in Usage.

Usage

Signed Shift Left Long (immediate). This instruction reads each vector element from the source SIMD and FP register, left shifts each vector element by the specified shift amount, places the result into a vector, and writes the vector to the destination SIMD and FP register. The destination vector elements are twice as long as the source vector elements. All the values in this instruction are signed integer values.

The `SSHLL` instruction extracts vector elements from the lower half of the source register, while the `SSHLL2` instruction extracts vector elements from the upper half of the source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-78 SSHLL, SSHLL2 (Vector) specifier combinations

<Q>	*Ta*	*Tb*	*shift*
-	8H	8B	0 to 7
2	8H	16B	0 to 7
-	4S	4H	0 to 15
2	4S	8H	0 to 15
-	2D	2S	0 to 31
2	2D	4S	0 to 31

Related references

20.220 SXTL, SXTL2 (vector) on page 20-1583.
20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.205 SSHR (vector)

Signed Shift Right (immediate).

Syntax

```
SSHR Vd.T, Vn.T, #shift
```

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of the values shown in Usage.

Vn
Is the name of the SIMD and FP source register.

shift
Is the right shift amount, in the range 1 to the element width in bits, and can be one of the values shown in Usage.

Usage

Signed Shift Right (immediate). This instruction reads each vector element in the source SIMD and FP register, right shifts each result by an immediate value, places the final result into a vector, and writes the vector to the destination SIMD and FP register. All the values in this instruction are signed integer values. The results are truncated. For rounded results, see *20.201 SRSHR (vector)* on page 20-1561.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-79 SSHR (Vector) specifier combinations

T	*shift*
8B	1 to 8
16B	1 to 8
4H	1 to 16
8H	1 to 16
2S	1 to 32
4S	1 to 32
2D	1 to 64

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.206 SSRA (vector)

Signed Shift Right and Accumulate (immediate).

Syntax

```
SSRA Vd.T, Vn.T, #shift
```

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of the values shown in Usage.

Vn
Is the name of the SIMD and FP source register.

shift
Is the right shift amount, in the range 1 to the element width in bits, and can be one of the values shown in Usage.

Usage

Signed Shift Right and Accumulate (immediate). This instruction reads each vector element in the source SIMD and FP register, right shifts each result by an immediate value, and accumulates the final results with the vector elements of the destination SIMD and FP register. All the values in this instruction are signed integer values. The results are truncated. For rounded results, see *20.202 SRSRA (vector)* on page 20-1562.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-80 SSRA (Vector) specifier combinations

T	*shift*
8B	1 to 8
16B	1 to 8
4H	1 to 16
8H	1 to 16
2S	1 to 32
4S	1 to 32
2D	1 to 64

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.207 SSUBL, SSUBL2 (vector)

Signed Subtract Long.

Syntax

`SSUBL{2} Vd.Ta, Vn.Tb, Vm.Tb`

Where:

2
Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements. See <Q> in the Usage table.

Vd
Is the name of the SIMD and FP destination register.

Ta
Is an arrangement specifier, and can be one of the values shown in Usage.

Vn
Is the name of the first SIMD and FP source register.

Tb
Is an arrangement specifier, and can be one of the values shown in Usage.

Vm
Is the name of the second SIMD and FP source register.

Usage

Signed Subtract Long. This instruction subtracts each vector element in the lower or upper half of the second source SIMD and FP register from the corresponding vector element of the first source SIMD and FP register, places the results into a vector, and writes the vector to the destination SIMD and FP register. All the values in this instruction are signed integer values. The destination vector elements are twice as long as the source vector elements.

The `SSUBL` instruction extracts each source vector from the lower half of each source register, while the `SSUBL2` instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-81 SSUBL, SSUBL2 (Vector) specifier combinations

<Q>	*Ta*	*Tb*
-	8H	8B
2	8H	16B
-	4S	4H
2	4S	8H
-	2D	2S
2	2D	4S

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.208 SSUBW, SSUBW2 (vector)

Signed Subtract Wide.

Syntax

`SSUBW{2} Vd.Ta, Vn.Ta, Vm.Tb`

Where:

2
Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements. See <Q> in the Usage table.

Vd
Is the name of the SIMD and FP destination register.

Ta
Is an arrangement specifier, and can be one of the values shown in Usage.

Vn
Is the name of the first SIMD and FP source register.

Vm
Is the name of the second SIMD and FP source register.

Tb
Is an arrangement specifier, and can be one of the values shown in Usage.

Usage

Signed Subtract Wide. This instruction subtracts each vector element in the lower or upper half of the second source SIMD and FP register from the corresponding vector element in the first source SIMD and FP register, places the result in a vector, and writes the vector to the SIMD and FP destination register. All the values in this instruction are signed integer values.

The `SSUBW` instruction extracts the second source vector from the lower half of the second source register, while the `SSUBW2` instruction extracts the second source vector from the upper half of the second source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-82 SSUBW, SSUBW2 (Vector) specifier combinations

<Q>	*Ta*	*Tb*
-	8H	8B
2	8H	16B
-	4S	4H
2	4S	8H
-	2D	2S
2	2D	4S

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.209 ST1 (vector, multiple structures)

Store multiple single-element structures from one, two, three, or four registers.

Syntax

```
ST1 { Vt.T }, [Xn|SP] ; T1 One register

ST1 { Vt.T, Vt2.T }, [Xn|SP] ; T1 Two registers

ST1 { Vt.T, Vt2.T, Vt3.T }, [Xn|SP] ; T1 Three registers

ST1 { Vt.T, Vt2.T, Vt3.T, Vt4.T }, [Xn|SP] ; T1 Four registers

ST1 { Vt.T }, [Xn|SP], imm ; T1 One register, immediate offset, Post-index

ST1 { Vt.T }, [Xn|SP], Xm ; T1 One register, register offset, Post-index

ST1 { Vt.T, Vt2.T }, [Xn|SP], imm ; T1 Two registers, immediate offset, Post-index

ST1 { Vt.T, Vt2.T }, [Xn|SP], Xm ; T1 Two registers, register offset, Post-index

ST1 { Vt.T, Vt2.T, Vt3.T }, [Xn|SP], imm ; T1 Three registers, immediate offset, Post-index

ST1 { Vt.T, Vt2.T, Vt3.T }, [Xn|SP], Xm ; T1 Three registers, register offset, Post-index

ST1 { Vt.T, Vt2.T, Vt3.T, Vt4.T }, [Xn|SP], imm ; T1 Four registers, immediate offset, Post-
index

ST1 { Vt.T, Vt2.T, Vt3.T, Vt4.T }, [Xn|SP], Xm ; T1 Four registers, register offset, Post-index
```

Where:

Vt2
Is the name of the second SIMD and FP register to be transferred.

Vt3
Is the name of the third SIMD and FP register to be transferred.

Vt4
Is the name of the fourth SIMD and FP register to be transferred.

imm
Is the post-index immediate offset:

One register, immediate offset
Can be one of `#8` or `#16`.

Two registers, immediate offset
Can be one of `#16` or `#32`.

Three registers, immediate offset
Can be one of `#24` or `#48`.

Four registers, immediate offset
Can be one of `#32` or `#64`.

Xm
Is the 64-bit name of the general-purpose post-index register, excluding XZR.

Vt
Is the name of the first or only SIMD and FP register to be transferred.

T
Is an arrangement specifier, and can be one of the values shown in Usage.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Usage

Store multiple single-element structures from one, two, three, or four registers. This instruction stores elements to memory from one, two, three, or four SIMD and FP registers, without interleaving. Every element of each register is stored.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following tables show valid specifier combinations:

Table 20-83 ST1 (One register, immediate offset) specifier combinations

T	*imm*
8B	#8
16B	#16
4H	#8
8H	#16
2S	#8
4S	#16
1D	#8
2D	#16

Table 20-84 ST1 (Two registers, immediate offset) specifier combinations

T	*imm*
8B	#16
16B	#32
4H	#16
8H	#32
2S	#16
4S	#32
1D	#16
2D	#32

Table 20-85 ST1 (Three registers, immediate offset) specifier combinations

T	*imm*
8B	#24
16B	#48
4H	#24
8H	#48
2S	#24
4S	#48
1D	#24
2D	#48

Table 20-86 ST1 (Four registers, immediate offset) specifier combinations

T	*imm*
8B	#32
16B	#64
4H	#32
8H	#64
2S	#32
4S	#64
1D	#32
2D	#64

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.210 ST1 (vector, single structure)

Store a single-element structure from one lane of one register.

Syntax

```
ST1 { Vt.B }[index], [Xn|SP] ; T1 8-bit
ST1 { Vt.H }[index], [Xn|SP] ; T1 16-bit
ST1 { Vt.S }[index], [Xn|SP] ; T1 32-bit
ST1 { Vt.D }[index], [Xn|SP] ; T1 64-bit
ST1 { Vt.B }[index], [Xn|SP], #1 ; T1 8-bit, immediate offset, Post-index
ST1 { Vt.B }[index], [Xn|SP], Xm ; T1 8-bit, register offset, Post-index
ST1 { Vt.H }[index], [Xn|SP], #2 ; T1 16-bit, immediate offset, Post-index
ST1 { Vt.H }[index], [Xn|SP], Xm ; T1 16-bit, register offset, Post-index
ST1 { Vt.S }[index], [Xn|SP], #4 ; T1 32-bit, immediate offset, Post-index
ST1 { Vt.S }[index], [Xn|SP], Xm ; T1 32-bit, register offset, Post-index
ST1 { Vt.D }[index], [Xn|SP], #8 ; T1 64-bit, immediate offset, Post-index
ST1 { Vt.D }[index], [Xn|SP], Xm ; T1 64-bit, register offset, Post-index
```

Where:

Vt
Is the name of the first or only SIMD and FP register to be transferred.

index
The value depends on the instruction variant:

8-bit
Is the element index, in the range 0 to 15.

16-bit
Is the element index, in the range 0 to 7.

32-bit
Is the element index, in the range 0 to 3.

64-bit
Is the element index, and can be either 0 or 1.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Xm
Is the 64-bit name of the general-purpose post-index register, excluding XZR.

Usage

Store a single-element structure from one lane of one register. This instruction stores the specified element of a SIMD and FP register to memory.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.211 ST2 (vector, multiple structures)

Store multiple 2-element structures from two registers.

Syntax

```
ST2 { Vt.T, Vt2.T }, [Xn|SP] ; T2
ST2 { Vt.T, Vt2.T }, [Xn|SP], imm ; T2
ST2 { Vt.T, Vt2.T }, [Xn|SP], Xm ; T2
```

Where:

Vt
Is the name of the first or only SIMD and FP register to be transferred.

Vt2
Is the name of the second SIMD and FP register to be transferred.

imm
Is the post-index immediate offset, and can be either `#16` or `#32`.

Xm
Is the 64-bit name of the general-purpose post-index register, excluding XZR.

T
Is an arrangement specifier, and can be one of `8B`, `16B`, `4H`, `8H`, `2S`, `4S` or `2D`.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Usage

Store multiple 2-element structures from two registers. This instruction stores multiple 2-element structures from two SIMD and FP registers to memory, with interleaving. Every element of each register is stored.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.212 ST2 (vector, single structure)

Store single 2-element structure from one lane of two registers.

Syntax

```
ST2 { Vt.B, Vt2.B }[index], [Xn|SP] ; T2
ST2 { Vt.H, Vt2.H }[index], [Xn|SP] ; T2
ST2 { Vt.S, Vt2.S }[index], [Xn|SP] ; T2
ST2 { Vt.D, Vt2.D }[index], [Xn|SP] ; T2
ST2 { Vt.B, Vt2.B }[index], [Xn|SP], #2 ; T2
ST2 { Vt.B, Vt2.B }[index], [Xn|SP], Xm ; T2
ST2 { Vt.H, Vt2.H }[index], [Xn|SP], #4 ; T2
ST2 { Vt.H, Vt2.H }[index], [Xn|SP], Xm ; T2
ST2 { Vt.S, Vt2.S }[index], [Xn|SP], #8 ; T2
ST2 { Vt.S, Vt2.S }[index], [Xn|SP], Xm ; T2
ST2 { Vt.D, Vt2.D }[index], [Xn|SP], #16 ; T2
ST2 { Vt.D, Vt2.D }[index], [Xn|SP], Xm ; T2
```

Where:

Vt
Is the name of the first or only SIMD and FP register to be transferred.

Vt2
Is the name of the second SIMD and FP register to be transferred.

index
The value depends on the instruction variant:

8-bit
Is the element index, in the range 0 to 15.

16-bit
Is the element index, in the range 0 to 7.

32-bit
Is the element index, in the range 0 to 3.

64-bit
Is the element index, and can be either 0 or 1.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Xm
Is the 64-bit name of the general-purpose post-index register, excluding XZR.

Usage

Store single 2-element structure from one lane of two registers. This instruction stores a 2-element structure to memory from corresponding elements of two SIMD and FP registers.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.213 ST3 (vector, multiple structures)

Store multiple 3-element structures from three registers.

Syntax

```
ST3 { Vt.T, Vt2.T, Vt3.T }, [Xn|SP] ; T3
ST3 { Vt.T, Vt2.T, Vt3.T }, [Xn|SP], imm ; T3
ST3 { Vt.T, Vt2.T, Vt3.T }, [Xn|SP], Xm ; T3
```

Where:

Vt
Is the name of the first or only SIMD and FP register to be transferred.

Vt2
Is the name of the second SIMD and FP register to be transferred.

Vt3
Is the name of the third SIMD and FP register to be transferred.

imm
Is the post-index immediate offset, and can be either `#24` or `#48`.

Xm
Is the 64-bit name of the general-purpose post-index register, excluding XZR.

T
Is an arrangement specifier, and can be one of `8B`, `16B`, `4H`, `8H`, `2S`, `4S` or `2D`.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Usage

Store multiple 3-element structures from three registers. This instruction stores multiple 3-element structures to memory from three SIMD and FP registers, with interleaving. Every element of each register is stored.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.214 ST3 (vector, single structure)

Store single 3-element structure from one lane of three registers.

Syntax

```
ST3 { Vt.B, Vt2.B, Vt3.B }[index], [Xn|SP] ; T3
ST3 { Vt.H, Vt2.H, Vt3.H }[index], [Xn|SP] ; T3
ST3 { Vt.S, Vt2.S, Vt3.S }[index], [Xn|SP] ; T3
ST3 { Vt.D, Vt2.D, Vt3.D }[index], [Xn|SP] ; T3
ST3 { Vt.B, Vt2.B, Vt3.B }[index], [Xn|SP], #3 ; T3
ST3 { Vt.B, Vt2.B, Vt3.B }[index], [Xn|SP], Xm ; T3
ST3 { Vt.H, Vt2.H, Vt3.H }[index], [Xn|SP], #6 ; T3
ST3 { Vt.H, Vt2.H, Vt3.H }[index], [Xn|SP], Xm ; T3
ST3 { Vt.S, Vt2.S, Vt3.S }[index], [Xn|SP], #12 ; T3
ST3 { Vt.S, Vt2.S, Vt3.S }[index], [Xn|SP], Xm ; T3
ST3 { Vt.D, Vt2.D, Vt3.D }[index], [Xn|SP], #24 ; T3
ST3 { Vt.D, Vt2.D, Vt3.D }[index], [Xn|SP], Xm ; T3
```

Where:

Vt
Is the name of the first or only SIMD and FP register to be transferred.

Vt2
Is the name of the second SIMD and FP register to be transferred.

Vt3
Is the name of the third SIMD and FP register to be transferred.

index
The value depends on the instruction variant:

8-bit
Is the element index, in the range 0 to 15.

16-bit
Is the element index, in the range 0 to 7.

32-bit
Is the element index, in the range 0 to 3.

64-bit
Is the element index, and can be either 0 or 1.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Xm
Is the 64-bit name of the general-purpose post-index register, excluding XZR.

Usage

Store single 3-element structure from one lane of three registers. This instruction stores a 3-element structure to memory from corresponding elements of three SIMD and FP registers.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.215 ST4 (vector, multiple structures)

Store multiple 4-element structures from four registers.

Syntax

```
ST4 { Vt.T, Vt2.T, Vt3.T, Vt4.T }, [Xn|SP] ;
ST4 { Vt.T, Vt2.T, Vt3.T, Vt4.T }, [Xn|SP], imm ;
ST4 { Vt.T, Vt2.T, Vt3.T, Vt4.T }, [Xn|SP], Xm ;
```

Where:

Vt
Is the name of the first or only SIMD and FP register to be transferred.

Vt2
Is the name of the second SIMD and FP register to be transferred.

Vt3
Is the name of the third SIMD and FP register to be transferred.

Vt4
Is the name of the fourth SIMD and FP register to be transferred.

imm
Is the post-index immediate offset, and can be either `#32` or `#64`.

Xm
Is the 64-bit name of the general-purpose post-index register, excluding XZR.

T
Is an arrangement specifier, and can be one of `8B`, `16B`, `4H`, `8H`, `2S`, `4S` or `2D`.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Usage

Store multiple 4-element structures from four registers. This instruction stores multiple 4-element structures to memory from four SIMD and FP registers, with interleaving. Every element of each register is stored.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.216 ST4 (vector, single structure)

Store single 4-element structure from one lane of four registers.

Syntax

```
ST4 { Vt.B, Vt2.B, Vt3.B, Vt4.B }[index], [Xn|SP] ;
ST4 { Vt.H, Vt2.H, Vt3.H, Vt4.H }[index], [Xn|SP] ;
ST4 { Vt.S, Vt2.S, Vt3.S, Vt4.S }[index], [Xn|SP] ;
ST4 { Vt.D, Vt2.D, Vt3.D, Vt4.D }[index], [Xn|SP] ;
ST4 { Vt.B, Vt2.B, Vt3.B, Vt4.B }[index], [Xn|SP], #4 ;
ST4 { Vt.B, Vt2.B, Vt3.B, Vt4.B }[index], [Xn|SP], Xm ;
ST4 { Vt.H, Vt2.H, Vt3.H, Vt4.H }[index], [Xn|SP], #8 ;
ST4 { Vt.H, Vt2.H, Vt3.H, Vt4.H }[index], [Xn|SP], Xm ;
ST4 { Vt.S, Vt2.S, Vt3.S, Vt4.S }[index], [Xn|SP], #16 ;
ST4 { Vt.S, Vt2.S, Vt3.S, Vt4.S }[index], [Xn|SP], Xm ;
ST4 { Vt.D, Vt2.D, Vt3.D, Vt4.D }[index], [Xn|SP], #32 ;
ST4 { Vt.D, Vt2.D, Vt3.D, Vt4.D }[index], [Xn|SP], Xm ;
```

Where:

Vt
Is the name of the first or only SIMD and FP register to be transferred.

Vt2
Is the name of the second SIMD and FP register to be transferred.

Vt3
Is the name of the third SIMD and FP register to be transferred.

Vt4
Is the name of the fourth SIMD and FP register to be transferred.

index
The value depends on the instruction variant:

8-bit
Is the element index, in the range 0 to 15.

16-bit
Is the element index, in the range 0 to 7.

32-bit
Is the element index, in the range 0 to 3.

64-bit
Is the element index, and can be either 0 or 1.

Xn|SP
Is the 64-bit name of the general-purpose base register or stack pointer.

Xm
Is the 64-bit name of the general-purpose post-index register, excluding XZR.

Usage

Store single 4-element structure from one lane of four registers. This instruction stores a 4-element structure to memory from corresponding elements of four SIMD and FP registers.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.217 SUB (vector)

Subtract (vector).

Syntax

`SUB` *`Vd.T, Vn.T, Vm.T`*

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of `8B`, `16B`, `4H`, `8H`, `2S`, `4S` or `2D`.

Vn
Is the name of the first SIMD and FP source register.

Vm
Is the name of the second SIMD and FP source register.

Usage

Subtract (vector). This instruction subtracts each vector element in the second source SIMD and FP register from the corresponding vector element in the first source SIMD and FP register, places the result into a vector, and writes the vector to the destination SIMD and FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.218 SUBHN, SUBHN2 (vector)

Subtract returning High Narrow.

Syntax

`SUBHN{2} Vd.Tb, Vn.Ta, Vm.Ta`

Where:

2
Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements. See <Q> in the Usage table.

Vd
Is the name of the SIMD and FP destination register.

Tb
Is an arrangement specifier, and can be one of the values shown in Usage.

Vn
Is the name of the first SIMD and FP source register.

Ta
Is an arrangement specifier, and can be one of the values shown in Usage.

Vm
Is the name of the second SIMD and FP source register.

Usage

Subtract returning High Narrow. This instruction subtracts each vector element in the second source SIMD and FP register from the corresponding vector element in the first source SIMD and FP register, places the most significant half of the result into a vector, and writes the vector to the lower or upper half of the destination SIMD and FP register. All the values in this instruction are signed integer values.

The results are truncated. For rounded results, see *20.139 RSUBHN, RSUBHN2 (vector)* on page 20-1496.

The `SUBHN` instruction writes the vector to the lower half of the destination register and clears the upper half, while the `SUBHN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-87 SUBHN, SUBHN2 (Vector) specifier combinations

<Q>	*Tb*	*Ta*
-	8B	8H
2	16B	8H
-	4H	4S
2	8H	4S
-	2S	2D
2	4S	2D

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.219 SUQADD (vector)

Signed saturating Accumulate of Unsigned value.

Syntax

`SUQADD` *`Vd.T, Vn.T`*

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of `8B`, `16B`, `4H`, `8H`, `2S`, `4S` or `2D`.

Vn
Is the name of the SIMD and FP source register.

Usage

Signed saturating Accumulate of Unsigned value. This instruction adds the unsigned integer values of the vector elements in the source SIMD and FP register to corresponding signed integer values of the vector elements in the destination SIMD and FP register, and writes the resulting signed integer values to the destination SIMD and FP register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.220 SXTL, SXTL2 (vector)

Signed extend Long.

This instruction is an alias of `SSHLL, SSHLL2`.

The equivalent instruction is `SSHLL{2} Vd.Ta, Vn.Tb, #0`.

Syntax

`SXTL{2} Vd.Ta, Vn.Tb`

Where:

2
Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements. See <Q> in the Usage table.

Vd
Is the name of the SIMD and FP destination register.

Ta
Is an arrangement specifier, and can be one of the values shown in Usage.

Vn
Is the name of the SIMD and FP source register.

Tb
Is an arrangement specifier, and can be one of the values shown in Usage.

Usage

Signed extend Long. This instruction duplicates each vector element in the lower or upper half of the source SIMD and FP register into a vector, and writes the vector to the destination SIMD and FP register. The destination vector elements are twice as long as the source vector elements. All the values in this instruction are signed integer values.

The `SXTL` instruction extracts the source vector from the lower half of the source register, while the `SXTL2` instruction extracts the source vector from the upper half of the source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-88 SXTL, SXTL2 (Vector) specifier combinations

<Q>	*Ta*	*Tb*
-	8H	8B
2	8H	16B
-	4S	4H
2	4S	8H
-	2D	2S
2	2D	4S

Related references

20.204 SSHLL, SSHLL2 (vector) on page 20-1564.
20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.221 TBL (vector)

Table vector Lookup.

Syntax

```
TBL Vd.Ta, { Vn.16B }, Vm.Ta ; Single register table
TBL Vd.Ta, { Vn.16B, <Vn+1>.16B }, Vm.Ta ; Two register table
TBL Vd.Ta, { Vn.16B, <Vn+1>.16B, <Vn+2>.16B }, Vm.Ta ; Three register table
TBL Vd.Ta, { Vn.16B, <Vn+1>.16B, <Vn+2>.16B, <Vn+3>.16B }, Vm.Ta ; Four register table
```

Where:

Vn
The value depends on the instruction variant:

Single register table
Is the name of the SIMD and FP table register

Two, Three, or Four register table
Is the name of the first SIMD and FP table register

<Vn+1>
Is the name of the second SIMD and FP table register.

<Vn+2>
Is the name of the third SIMD and FP table register.

<Vn+3>
Is the name of the fourth SIMD and FP table register.

Vd
Is the name of the SIMD and FP destination register.

Ta
Is an arrangement specifier, and can be either `8B` or `16B`.

Vm
Is the name of the SIMD and FP index register.

Usage

Table vector Lookup. This instruction reads each value from the vector elements in the index source SIMD and FP register, uses each result as an index to perform a lookup in a table of bytes that is described by one to four source table SIMD and FP registers, places the lookup result in a vector, and writes the vector to the destination SIMD and FP register. If an index is out of range for the table, the result for that lookup is 0. If more than one source register is used to describe the table, the first source register describes the lowest bytes of the table.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.222 TBX (vector)

Table vector lookup extension.

Syntax

```
TBX Vd.Ta, { Vn.16B }, Vm.Ta ; Single register table
TBX Vd.Ta, { Vn.16B, <Vn+1>.16B }, Vm.Ta ; Two register table
TBX Vd.Ta, { Vn.16B, <Vn+1>.16B, <Vn+2>.16B }, Vm.Ta ; Three register table
TBX Vd.Ta, { Vn.16B, <Vn+1>.16B, <Vn+2>.16B, <Vn+3>.16B }, Vm.Ta ; Four register table
```

Where:

Vn
The value depends on the instruction variant:

Single register table
Is the name of the SIMD and FP table register

Two, Three, or Four register table
Is the name of the first SIMD and FP table register

<Vn+1>
Is the name of the second SIMD and FP table register.

<Vn+2>
Is the name of the third SIMD and FP table register.

<Vn+3>
Is the name of the fourth SIMD and FP table register.

Vd
Is the name of the SIMD and FP destination register.

Ta
Is an arrangement specifier, and can be either `8B` or `16B`.

Vm
Is the name of the SIMD and FP index register.

Usage

Table vector lookup extension. This instruction reads each value from the vector elements in the index source SIMD and FP register, uses each result as an index to perform a lookup in a table of bytes that is described by one to four source table SIMD and FP registers, places the lookup result in a vector, and writes the vector to the destination SIMD and FP register. If an index is out of range for the table, the existing value in the vector element of the destination register is left unchanged. If more than one source register is used to describe the table, the first source register describes the lowest bytes of the table.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.223 TRN1 (vector)

Transpose vectors (primary).

Syntax

`TRN1` *`Vd.T, Vn.T, Vm.T`*

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of `8B`, `16B`, `4H`, `8H`, `2S`, `4S` or `2D`.

Vn
Is the name of the first SIMD and FP source register.

Vm
Is the name of the second SIMD and FP source register.

Usage

Transpose vectors (primary). This instruction reads corresponding even-numbered vector elements from the two source SIMD and FP registers, starting at zero, places each result into consecutive elements of a vector, and writes the vector to the destination SIMD and FP register. Vector elements from the first source register are placed into even-numbered elements of the destination vector, starting at zero, while vector elements from the second source register are placed into odd-numbered elements of the destination vector.

Note

By using this instruction with `TRN2`, a 2 x 2 matrix can be transposed.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.224 TRN2 (vector)

Transpose vectors (secondary).

Syntax

`TRN2 Vd.T, Vn.T, Vm.T`

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of `8B`, `16B`, `4H`, `8H`, `2S`, `4S` or `2D`.

Vn
Is the name of the first SIMD and FP source register.

Vm
Is the name of the second SIMD and FP source register.

Usage

Transpose vectors (secondary). This instruction reads corresponding odd-numbered vector elements from the two source SIMD and FP registers, places each result into consecutive elements of a vector, and writes the vector to the destination SIMD and FP register. Vector elements from the first source register are placed into even-numbered elements of the destination vector, starting at zero, while vector elements from the second source register are placed into odd-numbered elements of the destination vector.

Note

By using this instruction with `TRN1`, a 2 x 2 matrix can be transposed.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.225 UABA (vector)

Unsigned Absolute difference and Accumulate.

Syntax

`UABA` *`Vd.T, Vn.T, Vm.T`*

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of `8B`, `16B`, `4H`, `8H`, `2S` or `4S`.

Vn
Is the name of the first SIMD and FP source register.

Vm
Is the name of the second SIMD and FP source register.

Usage

Unsigned Absolute difference and Accumulate. This instruction subtracts the elements of the vector of the second source SIMD and FP register from the corresponding elements of the first source SIMD and FP register, and accumulates the absolute values of the results into the elements of the vector of the destination SIMD and FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.226 UABAL, UABAL2 (vector)

Unsigned Absolute difference and Accumulate Long.

Syntax

`UABAL{2} Vd.Ta, Vn.Tb, Vm.Tb`

Where:

2
Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements. See <Q> in the Usage table.

Vd
Is the name of the SIMD and FP destination register.

Ta
Is an arrangement specifier, and can be one of the values shown in Usage.

Vn
Is the name of the first SIMD and FP source register.

Tb
Is an arrangement specifier, and can be one of the values shown in Usage.

Vm
Is the name of the second SIMD and FP source register.

Usage

Unsigned Absolute difference and Accumulate Long. This instruction subtracts the vector elements in the lower or upper half of the second source SIMD and FP register from the corresponding vector elements of the first source SIMD and FP register, and accumulates the absolute values of the results into the vector elements of the destination SIMD and FP register. The destination vector elements are twice as long as the source vector elements. All the values in this instruction are unsigned integer values.

The `UABAL` instruction extracts each source vector from the lower half of each source register, while the `UABAL2` instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-89 UABAL, UABAL2 (Vector) specifier combinations

<Q>	*Ta*	*Tb*
-	8H	8B
2	8H	16B
-	4S	4H
2	4S	8H
-	2D	2S
2	2D	4S

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.227 UABD (vector)

Unsigned Absolute Difference (vector).

Syntax

`UABD Vd.T, Vn.T, Vm.T`

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of `8B`, `16B`, `4H`, `8H`, `2S` or `4S`.

Vn
Is the name of the first SIMD and FP source register.

Vm
Is the name of the second SIMD and FP source register.

Usage

Unsigned Absolute Difference (vector). This instruction subtracts the elements of the vector of the second source SIMD and FP register from the corresponding elements of the first source SIMD and FP register, places the absolute values of the results into a vector, and writes the vector to the destination SIMD and FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.228 UABDL, UABDL2 (vector)

Unsigned Absolute Difference Long.

Syntax

`UABDL{2} Vd.Ta, Vn.Tb, Vm.Tb`

Where:

2
Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements. See <Q> in the Usage table.

Vd
Is the name of the SIMD and FP destination register.

Ta
Is an arrangement specifier, and can be one of the values shown in Usage.

Vn
Is the name of the first SIMD and FP source register.

Tb
Is an arrangement specifier, and can be one of the values shown in Usage.

Vm
Is the name of the second SIMD and FP source register.

Usage

Unsigned Absolute Difference Long. This instruction subtracts the vector elements in the lower or upper half of the second source SIMD and FP register from the corresponding vector elements of the first source SIMD and FP register, places the absolute value of the result into a vector, and writes the vector to the destination SIMD and FP register. The destination vector elements are twice as long as the source vector elements. All the values in this instruction are unsigned integer values.

The `UABDL` instruction extracts each source vector from the lower half of each source register, while the `UABDL2` instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-90 UABDL, UABDL2 (Vector) specifier combinations

<Q>	*Ta*	*Tb*
-	8H	8B
2	8H	16B
-	4S	4H
2	4S	8H
-	2D	2S
2	2D	4S

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.229 UADALP (vector)

Unsigned Add and Accumulate Long Pairwise.

Syntax

```
UADALP Vd.Ta, Vn.Tb
```

Where:

Vd
Is the name of the SIMD and FP destination register.

Ta
Is an arrangement specifier, and can be one of the values shown in Usage.

Vn
Is the name of the SIMD and FP source register.

Tb
Is an arrangement specifier, and can be one of the values shown in Usage.

Usage

Unsigned Add and Accumulate Long Pairwise. This instruction adds pairs of adjacent unsigned integer values from the vector in the source SIMD and FP register and accumulates the results with the vector elements of the destination SIMD and FP register. The destination vector elements are twice as long as the source vector elements.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-91 UADALP (Vector) specifier combinations

Ta	*Tb*
4H	8B
8H	16B
2S	4H
4S	8H
1D	2S
2D	4S

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.230 UADDL, UADDL2 (vector)

Unsigned Add Long (vector).

Syntax

`UADDL{2} Vd.Ta, Vn.Tb, Vm.Tb`

Where:

2
Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements. See <Q> in the Usage table.

Vd
Is the name of the SIMD and FP destination register.

Ta
Is an arrangement specifier, and can be one of the values shown in Usage.

Vn
Is the name of the first SIMD and FP source register.

Tb
Is an arrangement specifier, and can be one of the values shown in Usage.

Vm
Is the name of the second SIMD and FP source register.

Usage

Unsigned Add Long (vector). This instruction adds each vector element in the lower or upper half of the first source SIMD and FP register to the corresponding vector element of the second source SIMD and FP register, places the result into a vector, and writes the vector to the destination SIMD and FP register. The destination vector elements are twice as long as the source vector elements. All the values in this instruction are unsigned integer values.

The `UADDL` instruction extracts each source vector from the lower half of each source register, while the `UADDL2` instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-92 UADDL, UADDL2 (Vector) specifier combinations

<Q>	*Ta*	*Tb*
-	8H	8B
2	8H	16B
-	4S	4H
2	4S	8H
-	2D	2S
2	2D	4S

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.231 UADDLP (vector)

Unsigned Add Long Pairwise.

Syntax

`UADDLP` *`Vd.Ta, Vn.Tb`*

Where:

Vd
Is the name of the SIMD and FP destination register.

Ta
Is an arrangement specifier, and can be one of the values shown in Usage.

Vn
Is the name of the SIMD and FP source register.

Tb
Is an arrangement specifier, and can be one of the values shown in Usage.

Usage

Unsigned Add Long Pairwise. This instruction adds pairs of adjacent unsigned integer values from the vector in the source SIMD and FP register, places the result into a vector, and writes the vector to the destination SIMD and FP register. The destination vector elements are twice as long as the source vector elements.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-93 UADDLP (Vector) specifier combinations

Ta	*Tb*
4H	8B
8H	16B
2S	4H
4S	8H
1D	2S
2D	4S

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.232 UADDLV (vector)

Unsigned sum Long across Vector.

Syntax

`UADDLV Vd, Vn.T`

Where:

V
Is the destination width specifier, and can be one of the values shown in Usage.

d
Is the number of the SIMD and FP destination register.

Vn
Is the name of the SIMD and FP source register.

T
Is an arrangement specifier, and can be one of the values shown in Usage.

Usage

Unsigned sum Long across Vector. This instruction adds every vector element in the source SIMD and FP register together, and writes the scalar result to the destination SIMD and FP register. The destination scalar is twice as long as the source vector elements. All the values in this instruction are unsigned integer values.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-94 UADDLV (Vector) specifier combinations

V	*T*
H	8B
H	16B
S	4H
S	8H
D	4S

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.233 UADDW, UADDW2 (vector)

Unsigned Add Wide.

Syntax

`UADDW{2} Vd.Ta, Vn.Ta, Vm.Tb`

Where:

2
Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements. See <Q> in the Usage table.

Vd
Is the name of the SIMD and FP destination register.

Ta
Is an arrangement specifier, and can be one of the values shown in Usage.

Vn
Is the name of the first SIMD and FP source register.

Vm
Is the name of the second SIMD and FP source register.

Tb
Is an arrangement specifier, and can be one of the values shown in Usage.

Usage

Unsigned Add Wide. This instruction adds the vector elements of the first source SIMD and FP register to the corresponding vector elements in the lower or upper half of the second source SIMD and FP register, places the result in a vector, and writes the vector to the SIMD and FP destination register. The vector elements of the destination register and the first source register are twice as long as the vector elements of the second source register. All the values in this instruction are unsigned integer values.

The `UADDW` instruction extracts vector elements from the lower half of the second source register, while the `UADDW2` instruction extracts vector elements from the upper half of the second source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-95 UADDW, UADDW2 (Vector) specifier combinations

<Q>	*Ta*	*Tb*
-	8H	8B
2	8H	16B
-	4S	4H
2	4S	8H
-	2D	2S
2	2D	4S

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.234 UCVTF (vector, fixed-point)

Unsigned fixed-point Convert to Floating-point (vector).

Syntax

```
UCVTF Vd.T, Vn.T, #fbits
```

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of the values shown in Usage.

Vn
Is the name of the SIMD and FP source register.

fbits
Is the number of fractional bits, in the range 1 to the element width.

Usage

Unsigned fixed-point Convert to Floating-point (vector). This instruction converts each element in a vector from fixed-point to floating-point using the rounding mode that is specified by the FPCR, and writes the result to the SIMD and FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-96 UCVTF (Vector) specifier combinations

T	*fbits*
4H	
8H	
2S	1 to 32
4S	1 to 32
2D	1 to 64

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.235 UCVTF (vector, integer)

Unsigned integer Convert to Floating-point (vector).

Syntax

```
UCVTF Vd.T, Vn.T ; Vector half precision
```

```
UCVTF Vd.T, Vn.T ; Vector single-precision and double-precision
```

Where:

Vd
Is the name of the SIMD and FP destination register

T
Is an arrangement specifier:

Vector half precision
Can be one of `4H` or `8H`.

Vector single-precision and double-precision
Can be one of `2S`, `4S` or `2D`.

Vn
Is the name of the SIMD and FP source register

Architectures supported (vector)

Supported in the Armv8.2 architecture and later.

Usage

Unsigned integer Convert to Floating-point (vector). This instruction converts each element in a vector from an unsigned integer value to a floating-point value using the rounding mode that is specified by the FPCR, and writes the result to the SIMD and FP destination register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps* in the *Arm® Architecture Reference Manual Arm®v8, for Arm®v8-A architecture profile*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.236 UHADD (vector)

Unsigned Halving Add.

Syntax

`UHADD Vd.T, Vn.T, Vm.T`

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of `8B`, `16B`, `4H`, `8H`, `2S` or `4S`.

Vn
Is the name of the first SIMD and FP source register.

Vm
Is the name of the second SIMD and FP source register.

Usage

Unsigned Halving Add. This instruction adds corresponding unsigned integer values from the two source SIMD and FP registers, shifts each result right one bit, places the results into a vector, and writes the vector to the destination SIMD and FP register.

The results are truncated. For rounded results, see *20.260 URHADD (vector)* on page 20-1624.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.237 UHSUB (vector)

Unsigned Halving Subtract.

Syntax

`UHSUB` *`Vd.T, Vn.T, Vm.T`*

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of `8B`, `16B`, `4H`, `8H`, `2S` or `4S`.

Vn
Is the name of the first SIMD and FP source register.

Vm
Is the name of the second SIMD and FP source register.

Usage

Unsigned Halving Subtract. This instruction subtracts the vector elements in the second source SIMD and FP register from the corresponding vector elements in the first source SIMD and FP register, shifts each result right one bit, places each result into a vector, and writes the vector to the destination SIMD and FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.238 UMAX (vector)

Unsigned Maximum (vector).

Syntax

`UMAX Vd.T, Vn.T, Vm.T`

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of `8B`, `16B`, `4H`, `8H`, `2S` or `4S`.

Vn
Is the name of the first SIMD and FP source register.

Vm
Is the name of the second SIMD and FP source register.

Usage

Unsigned Maximum (vector). This instruction compares corresponding elements in the vectors in the two source SIMD and FP registers, places the larger of each pair of unsigned integer values into a vector, and writes the vector to the destination SIMD and FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.239 UMAXP (vector)

Unsigned Maximum Pairwise.

Syntax

`UMAXP Vd.T, Vn.T, Vm.T`

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of `8B`, `16B`, `4H`, `8H`, `2S` or `4S`.

Vn
Is the name of the first SIMD and FP source register.

Vm
Is the name of the second SIMD and FP source register.

Usage

Unsigned Maximum Pairwise. This instruction creates a vector by concatenating the vector elements of the first source SIMD and FP register after the vector elements of the second source SIMD and FP register, reads each pair of adjacent vector elements in the two source SIMD and FP registers, writes the largest of each pair of unsigned integer values into a vector, and writes the vector to the destination SIMD and FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.240 UMAXV (vector)

Unsigned Maximum across Vector.

Syntax

`UMAXV Vd, Vn.T`

Where:

V
Is the destination width specifier, and can be one of the values shown in Usage.

d
Is the number of the SIMD and FP destination register.

Vn
Is the name of the SIMD and FP source register.

T
Is an arrangement specifier, and can be one of the values shown in Usage.

Usage

Unsigned Maximum across Vector. This instruction compares all the vector elements in the source SIMD and FP register, and writes the largest of the values as a scalar to the destination SIMD and FP register. All the values in this instruction are unsigned integer values.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-97 UMAXV (Vector) specifier combinations

V	*T*
B	8B
B	16B
H	4H
H	8H
S	4S

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.241 UMIN (vector)

Unsigned Minimum (vector).

Syntax

`UMIN Vd.T, Vn.T, Vm.T`

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of `8B`, `16B`, `4H`, `8H`, `2S` or `4S`.

Vn
Is the name of the first SIMD and FP source register.

Vm
Is the name of the second SIMD and FP source register.

Usage

Unsigned Minimum (vector). This instruction compares corresponding vector elements in the two source SIMD and FP registers, places the smaller of each of the two unsigned integer values into a vector, and writes the vector to the destination SIMD and FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.242 UMINP (vector)

Unsigned Minimum Pairwise.

Syntax

`UMINP` *`Vd.T, Vn.T, Vm.T`*

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of `8B`, `16B`, `4H`, `8H`, `2S` or `4S`.

Vn
Is the name of the first SIMD and FP source register.

Vm
Is the name of the second SIMD and FP source register.

Usage

Unsigned Minimum Pairwise. This instruction creates a vector by concatenating the vector elements of the first source SIMD and FP register after the vector elements of the second source SIMD and FP register, reads each pair of adjacent vector elements in the two source SIMD and FP registers, writes the smallest of each pair of unsigned integer values into a vector, and writes the vector to the destination SIMD and FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.243 UMINV (vector)

Unsigned Minimum across Vector.

Syntax

`UMINV Vd, Vn.T`

Where:

V
Is the destination width specifier, and can be one of the values shown in Usage.

d
Is the number of the SIMD and FP destination register.

Vn
Is the name of the SIMD and FP source register.

T
Is an arrangement specifier, and can be one of the values shown in Usage.

Usage

Unsigned Minimum across Vector. This instruction compares all the vector elements in the source SIMD and FP register, and writes the smallest of the values as a scalar to the destination SIMD and FP register. All the values in this instruction are unsigned integer values.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-98 UMINV (Vector) specifier combinations

V	*T*
B	8B
B	16B
H	4H
H	8H
S	4S

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.244 UMLAL, UMLAL2 (vector, by element)

Unsigned Multiply-Add Long (vector, by element).

Syntax

`UMLAL{2} Vd.Ta, Vn.Tb, Vm.Ts[index]`

Where:

2
Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements. See <Q> in the Usage table.

Vd
Is the name of the SIMD and FP destination register.

Ta
Is an arrangement specifier, and can be either `4S` or `2D`.

Vn
Is the name of the first SIMD and FP source register.

Tb
Is an arrangement specifier, and can be one of the values shown in Usage.

Vm
Is the name of the second SIMD and FP source register:
- If *Ts* is `H`, then *Vm* must be in the range V0 to V15.
- If *Ts* is `S`, then *Vm* must be in the range V0 to V31.

Ts
Is an element size specifier, and can be either `H` or `S`.

index
Is the element index, in the range shown in Usage.

Usage

Unsigned Multiply-Add Long (vector, by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD and FP register by the specified vector element of the second source SIMD and FP register and accumulates the results with the vector elements of the destination SIMD and FP register. The destination vector elements are twice as long as the elements that are multiplied.

The `UMLAL` instruction extracts vector elements from the lower half of the first source register, while the `UMLAL2` instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-99 UMLAL, UMLAL2 (Vector) specifier combinations

<Q>	*Ta*	*Tb*	*Ts*	*index*
-	4S	4H	H	0 to 7
2	4S	8H	H	0 to 7
-	2D	2S	S	0 to 3
2	2D	4S	S	0 to 3

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.245 UMLAL, UMLAL2 (vector)

Unsigned Multiply-Add Long (vector).

Syntax

`UMLAL{2} Vd.Ta, Vn.Tb, Vm.Tb`

Where:

2
Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements. See <Q> in the Usage table.

Vd
Is the name of the SIMD and FP destination register.

Ta
Is an arrangement specifier, and can be one of the values shown in Usage.

Vn
Is the name of the first SIMD and FP source register.

Tb
Is an arrangement specifier, and can be one of the values shown in Usage.

Vm
Is the name of the second SIMD and FP source register.

Usage

Unsigned Multiply-Add Long (vector). This instruction multiplies the vector elements in the lower or upper half of the first source SIMD and FP register by the corresponding vector elements of the second source SIMD and FP register, and accumulates the results with the vector elements of the destination SIMD and FP register. The destination vector elements are twice as long as the elements that are multiplied.

The `UMLAL` instruction extracts vector elements from the lower half of the first source register, while the `UMLAL2` instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-100 UMLAL, UMLAL2 (Vector) specifier combinations

<Q>	*Ta*	*Tb*
-	8H	8B
2	8H	16B
-	4S	4H
2	4S	8H
-	2D	2S
2	2D	4S

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.246 UMLSL, UMLSL2 (vector, by element)

Unsigned Multiply-Subtract Long (vector, by element).

Syntax

`UMLSL{2} Vd.Ta, Vn.Tb, Vm.Ts[index]`

Where:

2
Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements. See <Q> in the Usage table.

Vd
Is the name of the SIMD and FP destination register.

Ta
Is an arrangement specifier, and can be either `4S` or `2D`.

Vn
Is the name of the first SIMD and FP source register.

Tb
Is an arrangement specifier, and can be one of the values shown in Usage.

Vm
Is the name of the second SIMD and FP source register:
- If *Ts* is `H`, then *Vm* must be in the range V0 to V15.
- If *Ts* is `S`, then *Vm* must be in the range V0 to V31.

Ts
Is an element size specifier, and can be either `H` or `S`.

index
Is the element index, in the range shown in Usage.

Usage

Unsigned Multiply-Subtract Long (vector, by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD and FP register by the specified vector element of the second source SIMD and FP register and subtracts the results from the vector elements of the destination SIMD and FP register. The destination vector elements are twice as long as the elements that are multiplied.

The `UMLSL` instruction extracts vector elements from the lower half of the first source register, while the `UMLSL2` instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-101 UMLSL, UMLSL2 (Vector) specifier combinations

<Q>	*Ta*	*Tb*	*Ts*	*index*
-	4S	4H	H	0 to 7
2	4S	8H	H	0 to 7
-	2D	2S	S	0 to 3
2	2D	4S	S	0 to 3

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.247 UMLSL, UMLSL2 (vector)

Unsigned Multiply-Subtract Long (vector).

Syntax

`UMLSL{2} Vd.Ta, Vn.Tb, Vm.Tb`

Where:

2
Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements. See <Q> in the Usage table.

Vd
Is the name of the SIMD and FP destination register.

Ta
Is an arrangement specifier, and can be one of the values shown in Usage.

Vn
Is the name of the first SIMD and FP source register.

Tb
Is an arrangement specifier, and can be one of the values shown in Usage.

Vm
Is the name of the second SIMD and FP source register.

Usage

Unsigned Multiply-Subtract Long (vector). This instruction multiplies corresponding vector elements in the lower or upper half of the two source SIMD and FP registers, and subtracts the results from the vector elements of the destination SIMD and FP register. The destination vector elements are twice as long as the elements that are multiplied. All the values in this instruction are unsigned integer values.

The `UMLSL` instruction extracts each source vector from the lower half of each source register, while the `UMLSL2` instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-102 UMLSL, UMLSL2 (Vector) specifier combinations

<Q>	*Ta*	*Tb*
-	8H	8B
2	8H	16B
-	4S	4H
2	4S	8H
-	2D	2S
2	2D	4S

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.248 UMOV (vector)

Unsigned Move vector element to general-purpose register.

This instruction is used by the alias `MOV` (to general).

Syntax

```
UMOV Wd, Vn.Ts[index] ; 32-bit
UMOV Xd, Vn.Ts[index] ; 64-bit
```

Where:

Wd
Is the 32-bit name of the general-purpose destination register.

Ts
Is an element size specifier:

32-bit
Can be one of `B`, `H` or `S`.

64-bit
Must be `D`.

index
The value depends on the instruction variant:

32-bit
Is the element index, in the range shown in Usage.

64-bit
Is the element index and can be either 0 or 1.

Xd
Is the 64-bit name of the general-purpose destination register.

Vn
Is the name of the SIMD and FP source register.

Usage

Unsigned Move vector element to general-purpose register. This instruction reads the unsigned integer from the source SIMD and FP register, zero-extends it to form a 32-bit or 64-bit value, and writes the result to the destination general-purpose register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Table 20-103 UMOV (32-bit) specifier combinations

Ts	*index*
B	0 to 15
H	0 to 7
S	0 to 3

Related references

20.120 MOV (vector, to general) on page 20-1477.
20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.249 UMULL, UMULL2 (vector, by element)

Unsigned Multiply Long (vector, by element).

Syntax

`UMULL{2} Vd.Ta, Vn.Tb, Vm.Ts[index]`

Where:

2
Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements. See <Q> in the Usage table.

Vd
Is the name of the SIMD and FP destination register.

Ta
Is an arrangement specifier, and can be either `4S` or `2D`.

Vn
Is the name of the first SIMD and FP source register.

Tb
Is an arrangement specifier, and can be one of the values shown in Usage.

Vm
Is the name of the second SIMD and FP source register:
- If *Ts* is `H`, then *Vm* must be in the range V0 to V15.
- If *Ts* is `S`, then *Vm* must be in the range V0 to V31.

Ts
Is an element size specifier, and can be either `H` or `S`.

index
Is the element index, in the range shown in Usage.

Usage

Unsigned Multiply Long (vector, by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD and FP register by the specified vector element of the second source SIMD and FP register, places the results in a vector, and writes the vector to the destination SIMD and FP register. The destination vector elements are twice as long as the elements that are multiplied.

The `UMULL` instruction extracts vector elements from the lower half of the first source register, while the `UMULL2` instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-104 UMULL, UMULL2 (Vector) specifier combinations

<Q>	*Ta*	*Tb*	*Ts*	*index*
-	4S	4H	H	0 to 7
2	4S	8H	H	0 to 7
-	2D	2S	S	0 to 3
2	2D	4S	S	0 to 3

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.250 UMULL, UMULL2 (vector)

Unsigned Multiply long (vector).

Syntax

`UMULL{2} Vd.Ta, Vn.Tb, Vm.Tb`

Where:

2
Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements. See <Q> in the Usage table.

Vd
Is the name of the SIMD and FP destination register.

Ta
Is an arrangement specifier, and can be one of the values shown in Usage.

Vn
Is the name of the first SIMD and FP source register.

Tb
Is an arrangement specifier, and can be one of the values shown in Usage.

Vm
Is the name of the second SIMD and FP source register.

Usage

Unsigned Multiply long (vector). This instruction multiplies corresponding vector elements in the lower or upper half of the two source SIMD and FP registers, places the result in a vector, and writes the vector to the destination SIMD and FP register. The destination vector elements are twice as long as the elements that are multiplied. All the values in this instruction are unsigned integer values.

The `UMULL` instruction extracts each source vector from the lower half of each source register, while the `UMULL2` instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-105 UMULL, UMULL2 (Vector) specifier combinations

<Q>	*Ta*	*Tb*
-	8H	8B
2	8H	16B
-	4S	4H
2	4S	8H
-	2D	2S
2	2D	4S

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.251 UQADD (vector)

Unsigned saturating Add.

Syntax

`UQADD Vd.T, Vn.T, Vm.T`

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of `8B`, `16B`, `4H`, `8H`, `2S`, `4S` or `2D`.

Vn
Is the name of the first SIMD and FP source register.

Vm
Is the name of the second SIMD and FP source register.

Usage

Unsigned saturating Add. This instruction adds the values of corresponding elements of the two source SIMD and FP registers, places the results into a vector, and writes the vector to the destination SIMD and FP register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.252 UQRSHL (vector)

Unsigned saturating Rounding Shift Left (register).

Syntax

`UQRSHL Vd.T, Vn.T, Vm.T`

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of `8B`, `16B`, `4H`, `8H`, `2S`, `4S` or `2D`.

Vn
Is the name of the first SIMD and FP source register.

Vm
Is the name of the second SIMD and FP source register.

Usage

Unsigned saturating Rounding Shift Left (register). This instruction takes each vector element of the first source SIMD and FP register, shifts the vector element by a value from the least significant byte of the corresponding vector element of the second source SIMD and FP register, places the results into a vector, and writes the vector to the destination SIMD and FP register.

If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift. The results are rounded. For truncated results, see *20.254 UQSHL (vector, immediate)* on page 20-1617.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.253 UQRSHRN, UQRSHRN2 (vector)

Unsigned saturating Rounded Shift Right Narrow (immediate).

Syntax

`UQRSHRN{2} Vd.Tb, Vn.Ta, #shift`

Where:

2
Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements. See <Q> in the Usage table.

Vd
Is the name of the SIMD and FP destination register.

Tb
Is an arrangement specifier, and can be one of the values shown in Usage.

Vn
Is the name of the SIMD and FP source register.

Ta
Is an arrangement specifier, and can be one of the values shown in Usage.

shift
Is the right shift amount, in the range 1 to the destination element width in bits, and can be one of the values shown in Usage.

Usage

Unsigned saturating Rounded Shift Right Narrow (immediate). This instruction reads each vector element in the source SIMD and FP register, right shifts each result by an immediate value, puts the final result into a vector, and writes the vector to the lower or upper half of the destination SIMD and FP register. All the values in this instruction are unsigned integer values. The results are rounded. For truncated results, see *20.256 UQSHRN, UQSHRN2 (vector)* on page 20-1619.

The `UQRSHRN` instruction writes the vector to the lower half of the destination register and clears the upper half, while the `UQRSHRN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-106 UQRSHRN{2} (Vector) specifier combinations

<Q>	*Tb*	*Ta*	*shift*
-	8B	8H	1 to 8
2	16B	8H	1 to 8
-	4H	4S	1 to 16
2	8H	4S	1 to 16
-	2S	2D	1 to 32
2	4S	2D	1 to 32

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.254 UQSHL (vector, immediate)

Unsigned saturating Shift Left (immediate).

Syntax

```
UQSHL Vd.T, Vn.T, #shift
```

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of the values shown in Usage.

Vn
Is the name of the SIMD and FP source register.

shift
Is the left shift amount, in the range 0 to the element width in bits minus 1, and can be one of the values shown in Usage.

Usage

Unsigned saturating Shift Left (immediate). This instruction takes each vector element in the source SIMD and FP register, shifts it by an immediate value, places the results in a vector, and writes the vector to the destination SIMD and FP register. The results are truncated. For rounded results, see *20.252 UQRSHL (vector)* on page 20-1615.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-107 UQSHL (Vector) specifier combinations

T	*shift*
8B	0 to 7
16B	0 to 7
4H	0 to 15
8H	0 to 15
2S	0 to 31
4S	0 to 31
2D	0 to 63

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.255 UQSHL (vector, register)

Unsigned saturating Shift Left (register).

Syntax

`UQSHL` *`Vd.T, Vn.T, Vm.T`*

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of `8B`, `16B`, `4H`, `8H`, `2S`, `4S` or `2D`.

Vn
Is the name of the first SIMD and FP source register.

Vm
Is the name of the second SIMD and FP source register.

Usage

Unsigned saturating Shift Left (register). This instruction takes each element in the vector of the first source SIMD and FP register, shifts the element by a value from the least significant byte of the corresponding element of the second source SIMD and FP register, places the results in a vector, and writes the vector to the destination SIMD and FP register.

If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift. The results are truncated. For rounded results, see *20.252 UQRSHL (vector)* on page 20-1615.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.256 UQSHRN, UQSHRN2 (vector)

Unsigned saturating Shift Right Narrow (immediate).

Syntax

`UQSHRN{2} Vd.Tb, Vn.Ta, #shift`

Where:

2
Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements. See <Q> in the Usage table.

Vd
Is the name of the SIMD and FP destination register.

Tb
Is an arrangement specifier, and can be one of the values shown in Usage.

Vn
Is the name of the SIMD and FP source register.

Ta
Is an arrangement specifier, and can be one of the values shown in Usage.

shift
Is the right shift amount, in the range 1 to the destination element width in bits, and can be one of the values shown in Usage.

Usage

Unsigned saturating Shift Right Narrow (immediate). This instruction reads each vector element in the source SIMD and FP register, right shifts each result by an immediate value, saturates each shifted result to a value that is half the original width, puts the final result into a vector, and writes the vector to the lower or upper half of the destination SIMD and FP register. All the values in this instruction are unsigned integer values. The results are truncated. For rounded results, see *20.253 UQRSHRN, UQRSHRN2 (vector)* on page 20-1616.

The `UQSHRN` instruction writes the vector to the lower half of the destination register and clears the upper half, while the `UQSHRN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-108 UQSHRN{2} (Vector) specifier combinations

<Q>	*Tb*	*Ta*	*shift*
-	8B	8H	1 to 8
2	16B	8H	1 to 8
-	4H	4S	1 to 16
2	8H	4S	1 to 16
-	2S	2D	1 to 32
2	4S	2D	1 to 32

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.257 UQSUB (vector)

Unsigned saturating Subtract.

Syntax

`UQSUB` *`Vd.T, Vn.T, Vm.T`*

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of `8B`, `16B`, `4H`, `8H`, `2S`, `4S` or `2D`.

Vn
Is the name of the first SIMD and FP source register.

Vm
Is the name of the second SIMD and FP source register.

Usage

Unsigned saturating Subtract. This instruction subtracts the element values of the second source SIMD and FP register from the corresponding element values of the first source SIMD and FP register, places the results into a vector, and writes the vector to the destination SIMD and FP register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.258 UQXTN, UQXTN2 (vector)

Unsigned saturating extract Narrow.

Syntax

`UQXTN{2} Vd.Tb, Vn.Ta`

Where:

2
Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements. See <Q> in the Usage table.

Vd
Is the name of the SIMD and FP destination register.

Tb
Is an arrangement specifier, and can be one of the values shown in Usage.

Vn
Is the name of the SIMD and FP source register.

Ta
Is an arrangement specifier, and can be one of the values shown in Usage.

Usage

Unsigned saturating extract Narrow. This instruction reads each vector element from the source SIMD and FP register, saturates each value to half the original width, places the result into a vector, and writes the vector to the destination SIMD and FP register. All the values in this instruction are unsigned integer values.

If saturation occurs, the cumulative saturation bit FPSR.QC is set.

The `UQXTN` instruction writes the vector to the lower half of the destination register and clears the upper half, while the `UQXTN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-109 UQXTN{2} (Vector) specifier combinations

<Q>	*Tb*	*Ta*
-	8B	8H
2	16B	8H
-	4H	4S
2	8H	4S
-	2S	2D
2	4S	2D

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.259 URECPE (vector)

Unsigned Reciprocal Estimate.

Syntax

`URECPE Vd.T, Vn.T`

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be either `2S` or `4S`.

Vn
Is the name of the SIMD and FP source register.

Usage

Unsigned Reciprocal Estimate. This instruction reads each vector element from the source SIMD and FP register, calculates an approximate inverse for the unsigned integer value, places the result into a vector, and writes the vector to the destination SIMD and FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.260 URHADD (vector)

Unsigned Rounding Halving Add.

Syntax

URHADD *Vd.T, Vn.T, Vm.T*

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of `8B`, `16B`, `4H`, `8H`, `2S` or `4S`.

Vn
Is the name of the first SIMD and FP source register.

Vm
Is the name of the second SIMD and FP source register.

Usage

Unsigned Rounding Halving Add. This instruction adds corresponding unsigned integer values from the two source SIMD and FP registers, shifts each result right one bit, places the results into a vector, and writes the vector to the destination SIMD and FP register.

The results are rounded. For truncated results, see *20.236 UHADD (vector)* on page 20-1599.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.261 URSHL (vector)

Unsigned Rounding Shift Left (register).

Syntax

`URSHL` *`Vd.T, Vn.T, Vm.T`*

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of `8B`, `16B`, `4H`, `8H`, `2S`, `4S` or `2D`.

Vn
Is the name of the first SIMD and FP source register.

Vm
Is the name of the second SIMD and FP source register.

Usage

Unsigned Rounding Shift Left (register). This instruction takes each element in the vector of the first source SIMD and FP register, shifts the vector element by a value from the least significant byte of the corresponding element of the second source SIMD and FP register, places the results in a vector, and writes the vector to the destination SIMD and FP register.

If the shift value is positive, the operation is a left shift. If the shift value is negative, it is a rounding right shift.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.262 URSHR (vector)

Unsigned Rounding Shift Right (immediate).

Syntax

```
URSHR Vd.T, Vn.T, #shift
```

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of the values shown in Usage.

Vn
Is the name of the SIMD and FP source register.

shift
Is the right shift amount, in the range 1 to the element width in bits, and can be one of the values shown in Usage.

Usage

Unsigned Rounding Shift Right (immediate). This instruction reads each vector element in the source SIMD and FP register, right shifts each result by an immediate value, writes the final result to a vector, and writes the vector to the destination SIMD and FP register. All the values in this instruction are unsigned integer values. The results are rounded. For truncated results, see *20.267 USHR (vector)* on page 20-1631.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-110 URSHR (Vector) specifier combinations

T	*shift*
8B	1 to 8
16B	1 to 8
4H	1 to 16
8H	1 to 16
2S	1 to 32
4S	1 to 32
2D	1 to 64

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.263 URSQRTE (vector)

Unsigned Reciprocal Square Root Estimate.

Syntax

`URSQRTE` *`Vd.T, Vn.T`*

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be either `2S` or `4S`.

Vn
Is the name of the SIMD and FP source register.

Usage

Unsigned Reciprocal Square Root Estimate. This instruction reads each vector element from the source SIMD and FP register, calculates an approximate inverse square root for each value, places the result into a vector, and writes the vector to the destination SIMD and FP register. All the values in this instruction are unsigned integer values.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.264 URSRA (vector)

Unsigned Rounding Shift Right and Accumulate (immediate).

Syntax

`URSRA Vd.T, Vn.T, #shift`

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of the values shown in Usage.

Vn
Is the name of the SIMD and FP source register.

shift
Is the right shift amount, in the range 1 to the element width in bits, and can be one of the values shown in Usage.

Usage

Unsigned Rounding Shift Right and Accumulate (immediate). This instruction reads each vector element in the source SIMD and FP register, right shifts each result by an immediate value, and accumulates the final results with the vector elements of the destination SIMD and FP register. All the values in this instruction are unsigned integer values. The results are rounded. For truncated results, see *20.269 USRA (vector)* on page 20-1633.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-111 URSRA (Vector) specifier combinations

T	*shift*
8B	1 to 8
16B	1 to 8
4H	1 to 16
8H	1 to 16
2S	1 to 32
4S	1 to 32
2D	1 to 64

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.265 USHL (vector)

Unsigned Shift Left (register).

Syntax

`USHL` *`Vd.T, Vn.T, Vm.T`*

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of `8B`, `16B`, `4H`, `8H`, `2S`, `4S` or `2D`.

Vn
Is the name of the first SIMD and FP source register.

Vm
Is the name of the second SIMD and FP source register.

Usage

Unsigned Shift Left (register). This instruction takes each element in the vector of the first source SIMD and FP register, shifts each element by a value from the least significant byte of the corresponding element of the second source SIMD and FP register, places the results in a vector, and writes the vector to the destination SIMD and FP register.

If the shift value is positive, the operation is a left shift. If the shift value is negative, it is a truncating right shift. For a rounding shift, see *20.261 URSHL (vector)* on page 20-1625.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.266 USHLL, USHLL2 (vector)

Unsigned Shift Left Long (immediate).

This instruction is used by the alias `UXTL, UXTL2`, `UXTL, UXTL22`.

Syntax

`USHLL{2} Vd.Ta, Vn.Tb, #shift`

Where:

2
Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements. See <Q> in the Usage table.

Vd
Is the name of the SIMD and FP destination register.

Ta
Is an arrangement specifier, and can be one of the values shown in Usage.

Vn
Is the name of the SIMD and FP source register.

Tb
Is an arrangement specifier, and can be one of the values shown in Usage.

shift
Is the left shift amount, in the range 0 to the source element width in bits minus 1, and can be one of the values shown in Usage.

Usage

Unsigned Shift Left Long (immediate). This instruction reads each vector element in the lower or upper half of the source SIMD and FP register, shifts the unsigned integer value left by the specified number of bits, places the result into a vector, and writes the vector to the destination SIMD and FP register. The destination vector elements are twice as long as the source vector elements.

The `USHLL` instruction extracts vector elements from the lower half of the source register, while the `USHLL2` instruction extracts vector elements from the upper half of the source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-112 USHLL, USHLL2 (Vector) specifier combinations

<Q>	*Ta*	*Tb*	*shift*
-	8H	8B	0 to 7
2	8H	16B	0 to 7
-	4S	4H	0 to 15
2	4S	8H	0 to 15
-	2D	2S	0 to 31
2	2D	4S	0 to 31

Related references

20.272 UXTL, UXTL2 (vector) on page 20-1636.
20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.267 USHR (vector)

Unsigned Shift Right (immediate).

Syntax

`USHR Vd.T, Vn.T, #shift`

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of the values shown in Usage.

Vn
Is the name of the SIMD and FP source register.

shift
Is the right shift amount, in the range 1 to the element width in bits, and can be one of the values shown in Usage.

Usage

Unsigned Shift Right (immediate). This instruction reads each vector element in the source SIMD and FP register, right shifts each result by an immediate value, writes the final result to a vector, and writes the vector to the destination SIMD and FP register. All the values in this instruction are unsigned integer values. The results are truncated. For rounded results, see *20.262 URSHR (vector)* on page 20-1626.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-113 USHR (Vector) specifier combinations

T	*shift*
8B	1 to 8
16B	1 to 8
4H	1 to 16
8H	1 to 16
2S	1 to 32
4S	1 to 32
2D	1 to 64

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.268 USQADD (vector)

Unsigned saturating Accumulate of Signed value.

Syntax

`USQADD Vd.T, Vn.T`

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of `8B`, `16B`, `4H`, `8H`, `2S`, `4S` or `2D`.

Vn
Is the name of the SIMD and FP source register.

Usage

Unsigned saturating Accumulate of Signed value. This instruction adds the signed integer values of the vector elements in the source SIMD and FP register to corresponding unsigned integer values of the vector elements in the destination SIMD and FP register, and accumulates the resulting unsigned integer values with the vector elements of the destination SIMD and FP register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.269 USRA (vector)

Unsigned Shift Right and Accumulate (immediate).

Syntax

`USRA Vd.T, Vn.T, #shift`

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of the values shown in Usage.

Vn
Is the name of the SIMD and FP source register.

shift
Is the right shift amount, in the range 1 to the element width in bits, and can be one of the values shown in Usage.

Usage

Unsigned Shift Right and Accumulate (immediate). This instruction reads each vector element in the source SIMD and FP register, right shifts each result by an immediate value, and accumulates the final results with the vector elements of the destination SIMD and FP register. All the values in this instruction are unsigned integer values. The results are truncated. For rounded results, see *20.264 URSRA (vector)* on page 20-1628.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-114 USRA (Vector) specifier combinations

T	*shift*
8B	1 to 8
16B	1 to 8
4H	1 to 16
8H	1 to 16
2S	1 to 32
4S	1 to 32
2D	1 to 64

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.270 USUBL, USUBL2 (vector)

Unsigned Subtract Long.

Syntax

`USUBL{2} Vd.Ta, Vn.Tb, Vm.Tb`

Where:

2
Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements. See <Q> in the Usage table.

Vd
Is the name of the SIMD and FP destination register.

Ta
Is an arrangement specifier, and can be one of the values shown in Usage.

Vn
Is the name of the first SIMD and FP source register.

Tb
Is an arrangement specifier, and can be one of the values shown in Usage.

Vm
Is the name of the second SIMD and FP source register.

Usage

Unsigned Subtract Long. This instruction subtracts each vector element in the lower or upper half of the second source SIMD and FP register from the corresponding vector element of the first source SIMD and FP register, places the result into a vector, and writes the vector to the destination SIMD and FP register. All the values in this instruction are unsigned integer values. The destination vector elements are twice as long as the source vector elements.

The `USUBL` instruction extracts each source vector from the lower half of each source register, while the `USUBL2` instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-115 USUBL, USUBL2 (Vector) specifier combinations

<Q>	*Ta*	*Tb*
-	8H	8B
2	8H	16B
-	4S	4H
2	4S	8H
-	2D	2S
2	2D	4S

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.271 USUBW, USUBW2 (vector)

Unsigned Subtract Wide.

Syntax

`USUBW{2} Vd.Ta, Vn.Ta, Vm.Tb`

Where:

2
Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements. See <Q> in the Usage table.

Vd
Is the name of the SIMD and FP destination register.

Ta
Is an arrangement specifier, and can be one of the values shown in Usage.

Vn
Is the name of the first SIMD and FP source register.

Vm
Is the name of the second SIMD and FP source register.

Tb
Is an arrangement specifier, and can be one of the values shown in Usage.

Usage

Unsigned Subtract Wide. This instruction subtracts each vector element of the second source SIMD and FP register from the corresponding vector element in the lower or upper half of the first source SIMD and FP register, places the result in a vector, and writes the vector to the SIMD and FP destination register. All the values in this instruction are signed integer values.

The vector elements of the destination register and the first source register are twice as long as the vector elements of the second source register.

The `USUBW` instruction extracts vector elements from the lower half of the first source register, while the `USUBW2` instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-116 USUBW, USUBW2 (Vector) specifier combinations

<Q>	*Ta*	*Tb*
-	8H	8B
2	8H	16B
-	4S	4H
2	4S	8H
-	2D	2S
2	2D	4S

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.272 UXTL, UXTL2 (vector)

Unsigned extend Long.

This instruction is an alias of `USHLL, USHLL2`.

The equivalent instruction is `USHLL{2} Vd.Ta, Vn.Tb, #0`.

Syntax

`UXTL{2} Vd.Ta, Vn.Tb`

Where:

2
Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements. See <Q> in the Usage table.

Vd
Is the name of the SIMD and FP destination register.

Ta
Is an arrangement specifier, and can be one of the values shown in Usage.

Vn
Is the name of the SIMD and FP source register.

Tb
Is an arrangement specifier, and can be one of the values shown in Usage.

Usage

Unsigned extend Long. This instruction copies each vector element from the lower or upper half of the source SIMD and FP register into a vector, and writes the vector to the destination SIMD and FP register. The destination vector elements are twice as long as the source vector elements.

The `UXTL` instruction extracts vector elements from the lower half of the source register, while the `UXTL2` instruction extracts vector elements from the upper half of the source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-117 UXTL, UXTL2 (Vector) specifier combinations

<Q>	*Ta*	*Tb*
-	8H	8B
2	8H	16B
-	4S	4H
2	4S	8H
-	2D	2S
2	2D	4S

Related references

20.266 USHLL, USHLL2 (vector) on page 20-1630.
20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.273 UZP1 (vector)

Unzip vectors (primary).

Syntax

`UZP1 Vd.T, Vn.T, Vm.T`

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of `8B`, `16B`, `4H`, `8H`, `2S`, `4S` or `2D`.

Vn
Is the name of the first SIMD and FP source register.

Vm
Is the name of the second SIMD and FP source register.

Usage

Unzip vectors (primary). This instruction reads corresponding even-numbered vector elements from the two source SIMD and FP registers, starting at zero, places the result from the first source register into consecutive elements in the lower half of a vector, and the result from the second source register into consecutive elements in the upper half of a vector, and writes the vector to the destination SIMD and FP register.

Note

This instruction can be used with `UZP2` to de-interleave two vectors.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.274 UZP2 (vector)

Unzip vectors (secondary).

Syntax

`UZP2 Vd.T, Vn.T, Vm.T`

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of `8B`, `16B`, `4H`, `8H`, `2S`, `4S` or `2D`.

Vn
Is the name of the first SIMD and FP source register.

Vm
Is the name of the second SIMD and FP source register.

Usage

Unzip vectors (secondary). This instruction reads corresponding odd-numbered vector elements from the two source SIMD and FP registers, places the result from the first source register into consecutive elements in the lower half of a vector, and the result from the second source register into consecutive elements in the upper half of a vector, and writes the vector to the destination SIMD and FP register.

Note

This instruction can be used with `UZP1` to de-interleave two vectors.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.275 XTN, XTN2 (vector)

Extract Narrow.

Syntax

`XTN{2} Vd.Tb, Vn.Ta`

Where:

2
Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements. See <Q> in the Usage table.

Vd
Is the name of the SIMD and FP destination register.

Tb
Is an arrangement specifier, and can be one of the values shown in Usage.

Vn
Is the name of the SIMD and FP source register.

Ta
Is an arrangement specifier, and can be one of the values shown in Usage.

Usage

Extract Narrow. This instruction reads each vector element from the source SIMD and FP register, narrows each value to half the original width, places the result into a vector, and writes the vector to the lower or upper half of the destination SIMD and FP register. The destination vector elements are half as long as the source vector elements.

The `XTN` instruction writes the vector to the lower half of the destination register and clears the upper half, while the `XTN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The following table shows the valid specifier combinations:

Table 20-118 XTN, XTN2 (Vector) specifier combinations

<Q>	*Tb*	*Ta*
-	8B	8H
2	16B	8H
-	4H	4S
2	8H	4S
-	2S	2D
2	4S	2D

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.276 ZIP1 (vector)

Zip vectors (primary).

Syntax

`ZIP1` *`Vd.T, Vn.T, Vm.T`*

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of `8B`, `16B`, `4H`, `8H`, `2S`, `4S` or `2D`.

Vn
Is the name of the first SIMD and FP source register.

Vm
Is the name of the second SIMD and FP source register.

Usage

Zip vectors (primary). This instruction reads adjacent vector elements from the upper half of two source SIMD and FP registers as pairs, interleaves the pairs and places them into a vector, and writes the vector to the destination SIMD and FP register. The first pair from the first source register is placed into the two lowest vector elements, with subsequent pairs taken alternately from each source register.

Note

This instruction can be used with `ZIP2` to interleave two vectors.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

20.277 ZIP2 (vector)

Zip vectors (secondary).

Syntax

`ZIP2 Vd.T, Vn.T, Vm.T`

Where:

Vd
Is the name of the SIMD and FP destination register.

T
Is an arrangement specifier, and can be one of `8B`, `16B`, `4H`, `8H`, `2S`, `4S` or `2D`.

Vn
Is the name of the first SIMD and FP source register.

Vm
Is the name of the second SIMD and FP source register.

Usage

Zip vectors (secondary). This instruction reads adjacent vector elements from the lower half of two source SIMD and FP registers as pairs, interleaves the pairs and places them into a vector, and writes the vector to the destination SIMD and FP register. The first pair from the first source register is placed into the two lowest vector elements, with subsequent pairs taken alternately from each source register.

Note

This instruction can be used with `ZIP1` to interleave two vectors.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Related references

20.1 A64 SIMD Vector instructions in alphabetical order on page 20-1341.

Chapter 21
Directives Reference

Describes the directives that are provided by the Arm assembler, `armasm`.

It contains the following sections:

21.1 Alphabetical list of directives

The Arm assembler, armasm, provides various directives.

The following table lists them:

Table 21-1 List of directives

Directive	Directive	Directive
ALIAS	EQU	LTORG
ALIGN	EXPORT or GLOBAL	MACRO and MEND
ARM or CODE32	EXPORTAS	MAP
AREA	EXTERN	MEND (see MACRO)
ASSERT	FIELD	MEXIT
ATTR	FRAME ADDRESS	NOFP
CN	FRAME POP	OPT
CODE16	FRAME PUSH	PRESERVE8 (see REQUIRE8)
COMMON	FRAME REGISTER	PROC see FUNCTION
CP	FRAME RESTORE	-
DATA	FRAME SAVE	RELOC
DCB	FRAME STATE REMEMBER	REQUIRE
DCD and DCDU	FRAME STATE RESTORE	REQUIRE8 and PRESERVE8
DCDO	FRAME UNWIND ON or OFF	RLIST
DCFD and DCFDU	FUNCTION or PROC	RN
DCFS and DCFSU	GBLA, GBLL, and GBLS	ROUT
DCI	GET or INCLUDE	SETA, SETL, and SETS
DCQ and DCQU	GLOBAL (see EXPORT)	SN
DCW and DCWU	IF, ELSE, ENDIF, and ELIF	SPACE or FILL
DN	IMPORT	SUBT
ELIF, ELSE (see IF)	INCBIN	THUMB
END	INCLUDE see GET	TTL
ENDFUNC or ENDP	INFO	WHILE and WEND
ENDIF (see IF)	KEEP	WN and XN
ENTRY	LCLA, LCLL, and LCLS	-

21.2 About assembly control directives

Some assembler directives control conditional assembly, looping, inclusions, and macros.

These directives are as follows:

- `MACRO` and `MEND`.
- `MEXIT`.
- `IF`, `ELSE`, `ENDIF`, and `ELIF`.
- `WHILE` and `WEND`.

Nesting directives

The following structures can be nested to a total depth of 256:

- `MACRO` definitions.
- `WHILE...WEND` loops.
- `IF...ELSE...ENDIF` conditional structures.
- `INCLUDE` file inclusions.

The limit applies to all structures taken together, regardless of how they are nested. The limit is not 256 of each type of structure.

Related references

21.51 MACRO and MEND on page 21-1700.
21.53 MEXIT on page 21-1704.
21.44 IF, ELSE, ENDIF, and ELIF on page 21-1691.
21.67 WHILE and WEND on page 21-1721.

21.3 About frame directives

Frame directives enable debugging and profiling of assembly language functions. They also enable the stack usage of functions to be calculated.

Correct use of these directives:

- Enables the `armlink --callgraph` option to calculate stack usage of assembler functions.
  The following are the rules that determine stack usage:
  - If a function is not marked with `PROC` or `ENDP`, stack usage is unknown.
  - If a function is marked with `PROC` or `ENDP` but with no `FRAME PUSH` or `FRAME POP`, stack usage is assumed to be zero. This means that there is no requirement to manually add `FRAME PUSH 0` or `FRAME POP 0`.
  - If a function is marked with `PROC` or `ENDP` and with `FRAME PUSH n` or `FRAME POP n`, stack usage is assumed to be `n` bytes.
- Helps you to avoid errors in function construction, particularly when you are modifying existing code.
- Enables the assembler to alert you to errors in function construction.
- Enables backtracing of function calls during debugging.
- Enables the debugger to profile assembler functions.

If you require profiling of assembler functions, but do not want frame description directives for other purposes:

- You must use the `FUNCTION` and `ENDFUNC`, or `PROC` and `ENDP`, directives.
- You can omit the other `FRAME` directives.
- You only have to use the `FUNCTION` and `ENDFUNC` directives for the functions you want to profile.

In DWARF, the canonical frame address is an address on the stack specifying where the call frame of an interrupted function is located.

Related references

21.30 FRAME ADDRESS on page 21-1677.
21.31 FRAME POP on page 21-1678.
21.32 FRAME PUSH on page 21-1679.
21.33 FRAME REGISTER on page 21-1680.
21.34 FRAME RESTORE on page 21-1681.
21.35 FRAME RETURN ADDRESS on page 21-1682.
21.36 FRAME SAVE on page 21-1683.
21.37 FRAME STATE REMEMBER on page 21-1684.
21.38 FRAME STATE RESTORE on page 21-1685.
21.39 FRAME UNWIND ON on page 21-1686.
21.40 FRAME UNWIND OFF on page 21-1687.
21.41 FUNCTION or PROC on page 21-1688.
21.24 ENDFUNC or ENDP on page 21-1670.

21.4 ALIAS

The `ALIAS` directive creates an alias for a symbol.

Syntax

`ALIAS` *`name`*`,` *`aliasname`*

where:

`name`
is the name of the symbol to create an alias for.

`aliasname`
is the name of the alias to be created.

Usage

The symbol *`name`* must already be defined in the source file before creating an alias for it. Properties of *`name`* set by the `EXPORT` directive are not inherited by *`aliasname`*, so you must use `EXPORT` on *`aliasname`* if you want to make the alias available outside the current source file. Apart from the properties set by the `EXPORT` directive, *`name`* and *`aliasname`* are identical.

Correct example

```
baz
bar PROC
    BX lr
    ENDP
    ALIAS  bar,foo    ; foo is an alias for bar
    EXPORT bar
    EXPORT foo        ; foo and bar have identical properties
                      ; because foo was created using ALIAS
    EXPORT baz        ; baz and bar are not identical
                      ; because the size field of baz is not set
```

Incorrect example

```
    EXPORT bar
    IMPORT car
    ALIAS  bar,foo ; ERROR - bar is not defined yet
    ALIAS  car,boo ; ERROR - car is external
bar PROC
    BX lr
    ENDP
```

Related references

21.27 EXPORT or GLOBAL on page 21-1673.

21.5 ALIGN

The ALIGN directive aligns the current location to a specified boundary by padding with zeros or NOP instructions.

Syntax

```
ALIGN {expr{,offset{,pad{,padsize}}}}
```

where:

expr
: is a numeric expression evaluating to any power of 2 from 2^0 to 2^{31}

offset
: can be any numeric expression

pad
: can be any numeric expression

padsize
: can be 1, 2 or 4.

Operation

The current location is aligned to the next lowest address of the form:

```
offset + n * expr
```

n is any integer which the assembler selects to minimise padding.

If *expr* is not specified, ALIGN sets the current location to the next word (four byte) boundary. The unused space between the previous and the new current location are filled with:

- Copies of *pad*, if *pad* is specified.
- NOP instructions, if all the following conditions are satisfied:
  - *pad* is not specified.
  - The ALIGN directive follows A32 or T32 instructions.
  - The current section has the CODEALIGN attribute set on the AREA directive.
- Zeros otherwise.

pad is treated as a byte, halfword, or word, according to the value of *padsize*. If *padsize* is not specified, *pad* defaults to bytes in data sections, halfwords in T32 code, or words in A32 code.

Usage

Use ALIGN to ensure that your data and code is aligned to appropriate boundaries. This is typically required in the following circumstances:

- The ADR T32 pseudo-instruction can only load addresses that are word aligned, but a label within T32 code might not be word aligned. Use ALIGN 4 to ensure four-byte alignment of an address within T32 code.
- Use ALIGN to take advantage of caches on some Arm processors. For example, the Arm940T™ has a cache with 16-byte lines. Use ALIGN 16 to align function entries on 16-byte boundaries and maximize the efficiency of the cache.
- A label on a line by itself can be arbitrarily aligned. Following A32 code is word-aligned (T32 code is halfword aligned). The label therefore does not address the code correctly. Use ALIGN 4 (or ALIGN 2 for T32) before the label.

Alignment is relative to the start of the ELF section where the routine is located. The section must be aligned to the same, or coarser, boundaries. The ALIGN attribute on the AREA directive is specified differently.

Examples

```
        AREA    cacheable, CODE, ALIGN=3
rout1   ; code          ; aligned on 8-byte boundary
        ; code
```

```
        MOV     pc,lr  ; aligned only on 4-byte boundary
        ALIGN   8      ; now aligned on 8-byte boundary
rout2   ; code
```

In the following example, the ALIGN directive tells the assembler that the next instruction is word aligned and offset by 3 bytes. The 3 byte offset is counted from the previous word aligned address, resulting in the second DCB placed in the last byte of the same word and 2 bytes of padding are to be added.

```
        AREA    OffsetExample, CODE
        DCB     1      ; This example places the two bytes in the first
        ALIGN   4,3    ; and fourth bytes of the same word.
        DCB     1      ; The second DCB is offset by 3 bytes from the
                       ; first DCB.
```

In the following example, the ALIGN directive tells the assembler that the next instruction is word aligned and offset by 2 bytes. Here, the 2 byte offset is counted from the next word aligned address, so the value *n* is set to 1 (*n*=0 clashes with the third DCB). This time three bytes of padding are to be added.

```
        AREA    OffsetExample1, CODE
        DCB     1      ; In this example, n cannot be 0 because it
        DCB     1      ; clashes with the 3rd DCB. The assembler
        DCB     1      ; sets n to 1.
        ALIGN   4,2    ; The next instruction is word aligned and
        DCB     2      ; offset by 2.
```

In the following example, the DCB directive makes the PC misaligned. The ALIGN directive ensures that the label subroutine1 and the following instruction are word aligned.

```
        AREA    Example, CODE, READONLY
start   LDR     r6,=label1
        ; code
        MOV     pc,lr
label1  DCB     1      ; PC now misaligned
        ALIGN          ; ensures that subroutine1 addresses
subroutine1            ; the following instruction.
        MOV r5,#0x5
```

Related references

21.6 AREA on page 21-1650.

21.6 AREA

The AREA directive instructs the assembler to assemble a new code or data section.

Syntax

`AREA sectionname{,attr}{,attr}...`

where:

sectionname
is the name to give to the section. Sections are independent, named, indivisible chunks of code or data that are manipulated by the linker.

You can choose any name for your sections. However, names starting with a non-alphabetic character must be enclosed in bars or a missing section name error is generated. For example, `|1_DataArea|`.

Certain names are conventional. For example, `|.text|` is used for code sections produced by the C compiler, or for code sections otherwise associated with the C library.

attr
are one or more comma-delimited section attributes. Valid attributes are:

ALIGN=*expression*
By default, ELF sections are aligned on a four-byte boundary. *expression* can have any integer value from 0 to 31. The section is aligned on a $2^{expression}$-byte boundary. For example, if *expression* is 10, the section is aligned on a 1KB boundary.
This is not the same as the way that the ALIGN directive is specified.

Note

Do not use ALIGN=0 or ALIGN=1 for A32 code sections.

Do not use ALIGN=0 for T32 code sections.

ASSOC=*section*
section specifies an associated ELF section. *sectionname* must be included in any link that includes *section*

CODE
Contains machine instructions. READONLY is the default.

CODEALIGN
Causes armasm to insert NOP instructions when the ALIGN directive is used after A32 or T32 instructions within the section, unless the ALIGN directive specifies a different padding. CODEALIGN is the default for execute-only sections.

COMDEF
Is a common section definition. This ELF section can contain code or data. It must be identical to any other section of the same name in other source files.

Identical ELF sections with the same name are overlaid in the same section of memory by the linker. If any are different, the linker generates a warning and does not overlay the sections.

COMGROUP=*symbol_name*
Is the signature that makes the AREA part of the named ELF section group. See the GROUP=*symbol_name* for more information. The COMGROUP attribute marks the ELF section group with the GRP_COMDAT flag.

COMMON
Is a common data section. You must not define any code or data in it. It is initialized to zeros by the linker. All common sections with the same name are overlaid in the same section of memory by the linker. They do not all have to be the same size. The linker allocates as much space as is required by the largest common section of each name.

DATA
: Contains data, not instructions. READWRITE is the default.

EXECONLY
: Indicates that the section is execute-only. Execute-only sections must also have the CODE attribute, and must not have any of the following attributes:
  - READONLY.
  - READWRITE.
  - DATA.
  - ZEROALIGN.

  armasm faults if any of the following occur in an execute-only section:
  - Explicit data definitions, for example DCD and DCB.
  - Implicit data definitions, for example LDR r0, =0xaabbccdd.
  - Literal pool directives, for example LTORG, if there is literal data to be emitted.
  - INCBIN or SPACE directives.
  - ALIGN directives, if the required alignment cannot be accomplished by padding with NOP instructions. armasm implicitly applies the CODEALIGN attribute to sections with the EXECONLY attribute.

FINI_ARRAY
: Sets the ELF type of the current area to SHT_FINI_ARRAY.

GROUP=*symbol_name*
: Is the signature that makes the AREA part of the named ELF section group. It must be defined by the source file, or a file included by the source file. All AREAS with the same *symbol_name* signature are part of the same group. Sections within a group are kept or discarded together.

INIT_ARRAY
: Sets the ELF type of the current area to SHT_INIT_ARRAY.

LINKORDER=*section*
: Specifies a relative location for the current section in the image. It ensures that the order of all the sections with the LINKORDER attribute, with respect to each other, is the same as the order of the corresponding named *sections* in the image.

MERGE=*n*
: Indicates that the linker can merge the current section with other sections with the MERGE=*n* attribute. *n* is the size of the elements in the section, for example *n* is 1 for characters. You must not assume that the section is merged, because the attribute does not force the linker to merge the sections.

NOALLOC
: Indicates that no memory on the target system is allocated to this area.

NOINIT
: Indicates that the data section is uninitialized, or initialized to zero. It contains only space reservation directives SPACE or DCB, DCD, DCDU, DCQ, DCQU, DCW, or DCWU with initialized values of zero. You can decide at link time whether an area is uninitialized or zero-initialized.

  Note

  Arm Compiler does not support systems with ECC or parity protection where the memory is not initialized.

PREINIT_ARRAY
: Sets the ELF type of the current area to SHT_PREINIT_ARRAY.

READONLY
: Indicates that this section must not be written to. This is the default for Code areas.

READWRITE
: Indicates that this section can be read from and written to. This is the default for Data areas.

`SECFLAGS=n`
: Adds one or more ELF flags, denoted by *n*, to the current section.

`SECTYPE=n`
: Sets the ELF type of the current section to *n*.

`STRINGS`
: Adds the `SHF_STRINGS` flag to the current section. To use the `STRINGS` attribute, you must also use the `MERGE=1` attribute. The contents of the section must be strings that are nul-terminated using the `DCB` directive.

`ZEROALIGN`
: Causes `armasm` to insert zeros when the `ALIGN` directive is used after A32 or T32 instructions within the section, unless the `ALIGN` directive specifies a different padding. `ZEROALIGN` is the default for sections that are not execute-only.

Usage

Use the `AREA` directive to subdivide your source file into ELF sections. You can use the same name in more than one `AREA` directive. All areas with the same name are placed in the same ELF section. Only the attributes of the first `AREA` directive of a particular name are applied.

In general, Arm recommends that you use separate ELF sections for code and data. However, you can put data in code sections. Large programs can usually be conveniently divided into several code sections. Large independent data sets are also usually best placed in separate sections.

The scope of numeric local labels is defined by `AREA` directives, optionally subdivided by `ROUT` directives.

There must be at least one `AREA` directive for an assembly.

Note

`armasm` emits `R_ARM_TARGET1` relocations for the `DCD` and `DCDU` directives if the directive uses PC-relative expressions and is in any of the `PREINIT_ARRAY`, `FINI_ARRAY`, or `INIT_ARRAY` ELF sections. You can override the relocation using the `RELOC` directive after each `DCD` or `DCDU` directive. If this relocation is used, read-write sections might become read-only sections at link time if the platform ABI permits this.

Example

The following example defines a read-only code section named `Example`:

```
    AREA    Example,CODE,READONLY   ; An example code section.
            ; code
```

Related concepts

5.3 ELF sections and the AREA directive on page 5-97.

Related references

21.5 ALIGN on page 21-1648.
21.57 RELOC on page 21-1710.
21.16 DCD and DCDU on page 21-1662.

Related information

Information about image structure and generation.

21.7 ARM or CODE32 directive

The `ARM` directive instructs the assembler to interpret subsequent instructions as A32 instructions, using either the UAL or the pre-UAL Arm assembler language syntax. `CODE32` is a synonym for `ARM`.

Note

Not supported for AArch64 state.

Syntax

```
ARM
```

Usage

In files that contain code using different instruction sets, the `ARM` directive must precede any A32 code.

If necessary, this directive also inserts up to three bytes of padding to align to the next word boundary.

This directive does not assemble to any instructions. It also does not change the state. It only instructs `armasm` to assemble A32 instructions as appropriate, and inserts padding if necessary.

Example

This example shows how you can use `ARM` and `THUMB` directives to switch state and assemble both A32 and T32 instructions in a single area.

```
        AREA ToT32, CODE, READONLY      ; Name this block of code
        ENTRY                           ; Mark first instruction to execute
        ARM                             ; Subsequent instructions are A32
start
        ADR     r0, into_t32 + 1        ; Processor starts in A32 state
        BX      r0                      ; Inline switch to T32 state
        THUMB                           ; Subsequent instructions are T32
into_t32
        MOVS    r0, #10                 ; New-style T32 instructions
```

Related references

21.11 CODE16 directive on page 21-1657.
21.65 THUMB directive on page 21-1719.

Related information

Arm Architecture Reference Manual.

21.8 ASSERT

The ASSERT directive generates an error message during assembly if a given assertion is false.

Syntax

`ASSERT logical-expression`

where:

logical-expression
is an assertion that can evaluate to either {TRUE} or {FALSE}.

Usage

Use ASSERT to ensure that any necessary condition is met during assembly.

If the assertion is false an error message is generated and assembly fails.

Example

```
        ASSERT  label1 <= label2    ; Tests if the address
                                    ; represented by label1
                                    ; is <= the address
                                    ; represented by label2.
```

Related references

21.47 INFO on page 21-1696.

21.9 ATTR

The ATTR set directives set values for the ABI build attributes. The ATTR scope directives specify the scope for which the set value applies to.

Syntax

```
ATTR FILESCOPE

ATTR SCOPE name

ATTR settype tagid, value
```

where:

name
: is a section name or symbol name.

settype
: can be any of:
- SETVALUE.
- SETSTRING.
- SETCOMPATWITHVALUE.
- SETCOMPATWITHSTRING.

tagid
: is an attribute tag name (or its numerical value) defined in the ABI for the Arm Architecture.

value
: depends on *settype*:
- is a 32-bit integer value when *settype* is SETVALUE or SETCOMPATWITHVALUE.
- is a nul-terminated string when *settype* is SETSTRING or SETCOMPATWITHSTRING.

Usage

The ATTR set directives following the ATTR FILESCOPE directive apply to the entire object file. The ATTR set directives following the ATTR SCOPE *name* directive apply only to the named section or symbol.

For tags that expect an integer, you must use SETVALUE or SETCOMPATWITHVALUE. For tags that expect a string, you must use SETSTRING or SETCOMPATWITHSTRING.

Use SETCOMPATWITHVALUE and SETCOMPATWITHSTRING to set tag values which the object file is also compatible with.

Examples

```
ATTR  SETSTRING Tag_CPU_raw_name, "Cortex-A8"
ATTR  SETVALUE  Tag_VFP_arch, 3   ; VFPv3 instructions permitted.
ATTR  SETVALUE  10, 3             ; 10 is the numerical value of
                                  ; Tag_VFP_arch.
```

Related information

Addenda to, and Errata in, the ABI for the Arm Architecture.

21.10 CN

The CN directive defines a name for a coprocessor register.

Syntax

name CN *expr*

where:

name
is the name to be defined for the coprocessor register. *name* cannot be the same as any of the predefined names.

expr
evaluates to a coprocessor register number from 0 to 15.

Usage

Use CN to allocate convenient names to registers, to help you remember what you use each register for.

Note

Avoid conflicting uses of the same register under different names.

The names c0 to c15 are predefined.

Example

```
power    CN  6        ; defines power as a symbol for
                      ; coprocessor register 6
```

Related references

3.7 Predeclared core register names in AArch32 state on page 3-71.
3.8 Predeclared extension register names in AArch32 state on page 3-72.

21.11 CODE16 directive

The `CODE16` directive instructs the assembler to interpret subsequent instructions as T32 instructions, using the UAL syntax.

Note

Not supported for AArch64 state.

Syntax

```
CODE16
```

Usage

In files that contain code using different instruction sets,`CODE16` must precede T32 code written in pre-UAL syntax.

If necessary, this directive also inserts one byte of padding to align to the next halfword boundary.

This directive does not assemble to any instructions. It also does not change the state. It only instructs `armasm` to assemble T32 instructions as appropriate, and inserts padding if necessary.

Related references

21.7 ARM or CODE32 directive on page 21-1653.
21.65 THUMB directive on page 21-1719.

21.12 COMMON

The COMMON directive allocates a block of memory of the defined size, at the specified symbol.

Syntax

```
COMMON symbol{,size{,alignment}} {[attr]}
```

where:

symbol
: is the symbol name. The symbol name is case-sensitive.

size
: is the number of bytes to reserve.

alignment
: is the alignment.

attr
: can be any one of:

DYNAMIC
: sets the ELF symbol visibility to STV_DEFAULT.

PROTECTED
: sets the ELF symbol visibility to STV_PROTECTED.

HIDDEN
: sets the ELF symbol visibility to STV_HIDDEN.

INTERNAL
: sets the ELF symbol visibility to STV_INTERNAL.

Usage

You specify how the memory is aligned. If the alignment is omitted, the default alignment is four. If the size is omitted, the default size is zero.

You can access this memory as you would any other memory, but no space is allocated by the assembler in object files. The linker allocates the required space as zero-initialized memory during the link stage.

You cannot define, IMPORT or EXTERN a symbol that has already been created by the COMMON directive. In the same way, if a symbol has already been defined or used with the IMPORT or EXTERN directive, you cannot use the same symbol for the COMMON directive.

Correct example

```
    LDR       r0, =xyz
    COMMON    xyz,255,4   ; defines 255 bytes of ZI store, word-aligned
```

Incorrect example

```
    COMMON    foo,4,4
    COMMON    bar,4,4
foo DCD       0         ; cannot define label with same name as COMMON
    IMPORT    bar       ; cannot import label with same name as COMMON
```

21.13 CP

The `CP` directive defines a name for a specified coprocessor.

Syntax

`name` `CP` *`expr`*

where:

`name`
is the name to be assigned to the coprocessor. *`name`* cannot be the same as any of the predefined names.

`expr`
evaluates to a coprocessor number within the range 0 to 15.

Usage

Use `CP` to allocate convenient names to coprocessors, to help you to remember what you use each one for.

Note

Avoid conflicting uses of the same coprocessor under different names.

The names p0 to p15 are predefined for coprocessors 0 to 15.

Example

```
dmu    CP  6       ; defines dmu as a symbol for
                   ; coprocessor 6
```

Related references

3.7 Predeclared core register names in AArch32 state on page 3-71.
3.8 Predeclared extension register names in AArch32 state on page 3-72.

21.14 DATA

The DATA directive is no longer required. It is ignored by the assembler.

21.15 DCB

The DCB directive allocates one or more bytes of memory, and defines the initial runtime contents of the memory.

Syntax

```
{label} DCB expr{,expr}...
```

where:

expr
is either:

- A numeric expression that evaluates to an integer in the range -128 to 255.
- A quoted string. The characters of the string are loaded into consecutive bytes of store.

Usage

If DCB is followed by an instruction, use an ALIGN directive to ensure that the instruction is aligned.

= is a synonym for DCB.

Example

Unlike C strings, armasm syntax assembler strings are not nul-terminated. You can construct a nul-terminated C string using DCB as follows:

```
C_string   DCB  "C_string",0
```

Related concepts

12.14 Numeric expressions on page 12-311.

Related references

21.16 DCD and DCDU on page 21-1662.
21.21 DCQ and DCQU on page 21-1667.
21.22 DCW and DCWU on page 21-1668.
21.64 SPACE or FILL on page 21-1718.
21.5 ALIGN on page 21-1648.

21.16 DCD and DCDU

The DCD directive allocates one or more words of memory, aligned on four-byte boundaries, and defines the initial runtime contents of the memory. DCDU is the same, except that the memory alignment is arbitrary.

Syntax

```
{label} DCD{U} expr{,expr}
```

where:

expr
is either:

- A numeric expression.
- A PC-relative expression.

Usage

DCD inserts up to three bytes of padding before the first defined word, if necessary, to achieve four-byte alignment.

Use DCDU if you do not require alignment.

& is a synonym for DCD.

Examples

```
data1   DCD     1,5,20      ; Defines 3 words containing
                            ; decimal values 1, 5, and 20
data2   DCD     mem06 + 4   ; Defines 1 word containing 4 +
                            ; the address of the label mem06
        AREA    MyData, DATA, READWRITE
        DCB     255         ; Now misaligned ...
data3   DCDU    1,5,20      ; Defines 3 words containing
                            ; 1, 5 and 20, not word aligned
```

Related concepts

12.14 Numeric expressions on page 12-311.

Related references

21.15 DCB on page 21-1661.
21.21 DCQ and DCQU on page 21-1667.
21.22 DCW and DCWU on page 21-1668.
21.64 SPACE or FILL on page 21-1718.
21.20 DCI on page 21-1666.

21.17 DCDO

The DCDO directive allocates one or more words of memory, aligned on four-byte boundaries, and defines the initial runtime contents of the memory as an offset from the *static base register*, sb (R9).

Syntax

{*label*} DCDO *expr*{,*expr*}...

where:

expr
is a register-relative expression or label. The base register must be sb.

Usage

Use DCDO to allocate space in memory for static base register relative relocatable addresses.

Example

```
        IMPORT  externsym
        DCDO    externsym   ; 32-bit word relocated by offset of
                            ; externsym from base of SB section.
```

21.18 DCFD and DCFDU

The DCFD directive allocates memory for word-aligned double-precision floating-point numbers, and defines the initial runtime contents of the memory. DCFDU is the same, except that the memory alignment is arbitrary.

Syntax

{*label*} DCFD{U} *fpliteral*{,*fpliteral*}...

where:

fpliteral
is a double-precision floating-point literal.

Usage

Double-precision numbers occupy two words and must be word aligned to be used in arithmetic operations. The assembler inserts up to three bytes of padding before the first defined number, if necessary, to achieve four-byte alignment.

Use DCFDU if you do not require alignment.

The word order used when converting *fpliteral* to internal form is controlled by the floating-point architecture selected. You cannot use DCFD or DCFDU if you select the `--fpu none` option.

The range for double-precision numbers is:
- Maximum 1.79769313486231571e+308.
- Minimum 2.22507385850720138e-308.

Examples

```
        DCFD    1E308,-4E-100
        DCFDU   10000,-.1,3.1E26
```

Related references

21.19 DCFS and DCFSU on page 21-1665.
12.16 Syntax of floating-point literals on page 12-313.

21.19 DCFS and DCFSU

The DCFS directive allocates memory for word-aligned single-precision floating-point numbers, and defines the initial runtime contents of the memory. DCFSU is the same, except that the memory alignment is arbitrary.

Syntax

{*label*} DCFS{U} *fpliteral*{,*fpliteral*}...

where:

fpliteral
is a single-precision floating-point literal.

Usage

Single-precision numbers occupy one word and must be word aligned to be used in arithmetic operations. DCFS inserts up to three bytes of padding before the first defined number, if necessary to achieve four-byte alignment.

Use DCFSU if you do not require alignment.

The range for single-precision values is:

- Maximum 3.40282347e+38.
- Minimum 1.17549435e-38.

Examples

```
        DCFS    1E3,-4E-9
        DCFSU   1.0,-.1,3.1E6
```

Related references

21.18 DCFD and DCFDU on page 21-1664.
12.16 Syntax of floating-point literals on page 12-313.

21.20 DCI

The DCI directive allocates memory that is aligned and defines the initial runtime contents of the memory.

In A32 code, it allocates one or more words of memory, aligned on four-byte boundaries.

In T32 code, it allocates one or more halfwords of memory, aligned on two-byte boundaries.

Syntax

```
{label} DCI{.W} expr{,expr}
```

where:

expr
: is a numeric expression.

.W
: if present, indicates that four bytes must be inserted in T32 code.

Usage

The DCI directive is very like the DCD or DCW directives, but the location is marked as code instead of data. Use DCI when writing macros for new instructions not supported by the version of the assembler you are using.

In A32 code, DCI inserts up to three bytes of padding before the first defined word, if necessary, to achieve four-byte alignment. In T32 code, DCI inserts an initial byte of padding, if necessary, to achieve two-byte alignment.

You can use DCI to insert a bit pattern into the instruction stream. For example, use:

```
DCI 0x46c0
```

to insert the T32 operation MOV r8,r8.

Example macro

```
    MACRO           ; this macro translates newinstr Rd,Rm
                    ; to the appropriate machine code
    newinst     $Rd,$Rm
    DCI         0xe16f0f10 :OR: ($Rd:SHL:12) :OR: $Rm
    MEND
```

32-bit T32 example

```
    DCI.W   0xf3af8000   ; inserts 32-bit NOP, 2-byte aligned.
```

Related concepts

12.14 Numeric expressions on page 12-311.

Related references

21.16 DCD and DCDU on page 21-1662.
21.22 DCW and DCWU on page 21-1668.

21.21 DCQ and DCQU

The DCQ directive allocates one or more eight-byte blocks of memory, aligned on four-byte boundaries, and defines the initial runtime contents of the memory. DCQU is the same, except that the memory alignment is arbitrary.

Syntax

```
{label} DCQ{U} {-}literal{,{-}literal…}
```

```
{label} DCQ{U} expr{,expr…}
```

where:

literal
is a 64-bit numeric literal.

The range of numbers permitted is 0 to 2^{64}-1.

In addition to the characters normally permitted in a numeric literal, you can prefix *literal* with a minus sign. In this case, the range of numbers permitted is -2^{63} to -1.

The result of specifying *–n* is the same as the result of specifying 2^{64}*–n*.

expr
is either:

- A numeric expression.
- A PC-relative expression.

Note

armasm accepts expressions in DCQ and DCQU directives only when you are assembling for AArch64 targets.

Usage

DCQ inserts up to three bytes of padding before the first defined eight-byte block, if necessary, to achieve four-byte alignment.

Use DCQU if you do not require alignment.

Correct example

```
        AREA    MiscData, DATA, READWRITE
data    DCQ     -225,2_101      ; 2_101 means binary 101.
```

Incorrect example

```
number  EQU     2               ; This code assembles for AArch64 targets only.
        DCQU    number          ; For AArch32 targets, DCQ and DCQU only accept
                                ; literals, not expressions.
```

Related concepts

12.14 Numeric expressions on page 12-311.

Related references

21.15 DCB on page 21-1661.
21.16 DCD and DCDU on page 21-1662.
21.22 DCW and DCWU on page 21-1668.
21.64 SPACE or FILL on page 21-1718.

21.22 DCW and DCWU

The DCW directive allocates one or more halfwords of memory, aligned on two-byte boundaries, and defines the initial runtime contents of the memory. DCWU is the same, except that the memory alignment is arbitrary.

Syntax

{*label*} DCW{U} *expr*{,*expr*}...

where:

expr
is a numeric expression that evaluates to an integer in the range -32768 to 65535.

Usage

DCW inserts a byte of padding before the first defined halfword if necessary to achieve two-byte alignment.

Use DCWU if you do not require alignment.

Examples

```
data    DCW     -225,2*number   ; number must already be defined
        DCWU    number+4
```

Related concepts

12.14 Numeric expressions on page 12-311.

Related references

21.15 DCB on page 21-1661.
21.16 DCD and DCDU on page 21-1662.
21.21 DCQ and DCQU on page 21-1667.
21.64 SPACE or FILL on page 21-1718.

21.23 END

The END directive informs the assembler that it has reached the end of a source file.

Syntax

```
END
```

Usage

Every assembly language source file must end with END on a line by itself.

If the source file has been included in a parent file by a GET directive, the assembler returns to the parent file and continues assembly at the first line following the GET directive.

If END is reached in the top-level source file during the first pass without any errors, the second pass begins.

If END is reached in the top-level source file during the second pass, the assembler finishes the assembly and writes the appropriate output.

Related references

21.43 GET or INCLUDE on page 21-1690.

21.24 ENDFUNC or ENDP

The `ENDFUNC` directive marks the end of an AAPCS-conforming function. `ENDP` is a synonym for `ENDFUNC`.

Related references

21.41 FUNCTION or PROC on page 21-1688.

21.25 ENTRY

The ENTRY directive declares an entry point to a program.

Syntax

```
ENTRY
```

Usage

A program must have an entry point. You can specify an entry point in the following ways:

- Using the ENTRY directive in assembly language source code.
- Providing a `main()` function in C or C++ source code.
- Using the `armlink --entry` command-line option.

You can declare more than one entry point in a program, although a source file cannot contain more than one ENTRY directive. For example, a program could contain multiple assembly language source files, each with an ENTRY directive. Or it could contain a C or C++ file with a `main()` function and one or more assembly source files with an ENTRY directive.

If the program contains multiple entry points, then you must select one of them. You do this by exporting the symbol for the ENTRY directive that you want to use as the entry point, then using the `armlink --entry` option to select the exported symbol.

Example

```
        AREA   ARMex, CODE, READONLY
        ENTRY      ; Entry point for the application.
        EXPORT ep1 ; Export the symbol so the linker can find it
ep1                ; in the object file.
        ; code
        END
```

When you invoke `armlink`, if other entry points are declared in the program, then you must specify `--entry=ep1`, to select `ep1`.

Related information

Image entry points.
--entry=location.

21.26 EQU

The EQU directive gives a symbolic name to a numeric constant, a register-relative value or a PC-relative value.

Syntax

name EQU *expr*{, *type*}

where:

name
is the symbolic name to assign to the value.

expr
is a register-relative address, a PC-relative address, an absolute address, or a 32-bit integer constant.

type
is optional. *type* can be any one of:

- ARM.
- THUMB.
- CODE32.
- CODE16.
- DATA.

You can use *type* only if *expr* is an absolute address. If *name* is exported, the *name* entry in the symbol table in the object file is marked as ARM, THUMB, CODE32, CODE16, or DATA, according to *type*. This can be used by the linker.

Usage

Use EQU to define constants. This is similar to the use of **#define** to define a constant in C.

* is a synonym for EQU.

Examples

```
abc EQU 2               ; Assigns the value 2 to the symbol abc.
xyz EQU label+8         ; Assigns the address (label+8) to the
                        ; symbol xyz.
fiq EQU 0x1C, CODE32    ; Assigns the absolute address 0x1C to
                        ; the symbol fiq, and marks it as code.
```

Related references

21.48 KEEP on page 21-1697.
21.27 EXPORT or GLOBAL on page 21-1673.

21.27 EXPORT or GLOBAL

The `EXPORT` directive declares a symbol that can be used by the linker to resolve symbol references in separate object and library files. `GLOBAL` is a synonym for `EXPORT`.

Syntax

```
EXPORT {[WEAK]}

EXPORT symbol {[SIZE=n]}

EXPORT symbol {[type{,set}]}

EXPORT symbol [attr{,type{,set}}{,SIZE=n}]

EXPORT symbol [WEAK {,attr}{,type{,set}}{,SIZE=n}]
```

where:

symbol
: is the symbol name to export. The symbol name is case-sensitive. If *symbol* is omitted, all symbols are exported.

WEAK
: *symbol* is only imported into other sources if no other source exports an alternative *symbol*. If `[WEAK]` is used without *symbol*, all exported symbols are weak.

attr
: can be any one of:

  DYNAMIC
  : sets the ELF symbol visibility to `STV_DEFAULT`.

  PROTECTED
  : sets the ELF symbol visibility to `STV_PROTECTED`.

  HIDDEN
  : sets the ELF symbol visibility to `STV_HIDDEN`.

  INTERNAL
  : sets the ELF symbol visibility to `STV_INTERNAL`.

type
: specifies the symbol type:

  DATA
  : *symbol* is treated as data when the source is assembled and linked.

  CODE
  : *symbol* is treated as code when the source is assembled and linked.

  `ELFTYPE=`*n*
  : *symbol* is treated as a particular ELF symbol, as specified by the value of *n*, where *n* can be any number from `0` to `15`.

  If unspecified, the assembler determines the most appropriate type. Usually the assembler determines the correct type so you are not required to specify it.

set
: specifies the instruction set:

  ARM
  : *symbol* is treated as an A32 symbol.

  THUMB
  : *symbol* is treated as a T32 symbol.

  If unspecified, the assembler determines the most appropriate set.

n

specifies the size and can be any 32-bit value. If the `SIZE` attribute is not specified, the assembler calculates the size:

- For `PROC` and `FUNCTION` symbols, the size is set to the size of the code until its `ENDP` or `ENDFUNC`.
- For other symbols, the size is the size of instruction or data on the same source line. If there is no instruction or data, the size is zero.

Usage

Use `EXPORT` to give code in other files access to symbols in the current file.

Use the `[WEAK]` attribute to inform the linker that a different instance of *`symbol`* takes precedence over this one, if a different one is available from another source. You can use the `[WEAK]` attribute with any of the symbol visibility attributes.

Examples

```
        AREA    Example,CODE,READONLY
        EXPORT  DoAdd           ; Export the function name
                                ; to be used by external modules.
DoAdd   ADD     r0,r0,r1
```

Symbol visibility can be overridden for duplicate exports. In the following example, the last `EXPORT` takes precedence for both binding and visibility:

```
        EXPORT  SymA[WEAK]      ; Export as weak-hidden
        EXPORT  SymA[DYNAMIC]   ; SymA becomes non-weak dynamic.
```

The following examples show the use of the `SIZE` attribute:

```
        EXPORT symA [SIZE=4]
        EXPORT symA [DATA, SIZE=4]
```

Related references

21.45 IMPORT and EXTERN on page 21-1693.

Related information

ELF for the Arm Architecture.

21.28 EXPORTAS

The EXPORTAS directive enables you to export a symbol from the object file, corresponding to a different symbol in the source file.

Syntax

```
EXPORTAS symbol1, symbol2
```

where:

symbol1
: is the symbol name in the source file. *symbol1* must have been defined already. It can be any symbol, including an area name, a label, or a constant.

symbol2
: is the symbol name you want to appear in the object file.

The symbol names are case-sensitive.

Usage

Use EXPORTAS to change a symbol in the object file without having to change every instance in the source file.

Examples

```
    AREA data1, DATA       ; Starts a new area data1.
    AREA data2, DATA       ; Starts a new area data2.
    EXPORTAS data2, data1  ; The section symbol referred to as data2
                           ; appears in the object file string table as data1.
one EQU  2
    EXPORTAS one, two      ; The symbol 'two' appears in the object
    EXPORT one             ; file's symbol table with the value 2.
```

Related references

21.27 EXPORT or GLOBAL on page 21-1673.

21.29 FIELD

The `FIELD` directive describes space within a storage map that has been defined using the `MAP` directive.

Syntax

`{`*`label`*`} FIELD` *`expr`*

where:

`label`
is an optional label. If specified, *`label`* is assigned the value of the storage location counter, `{VAR}`. The storage location counter is then incremented by the value of *`expr`*.

`expr`
is an expression that evaluates to the number of bytes to increment the storage counter.

Usage

If a storage map is set by a `MAP` directive that specifies a *`base-register`*, the base register is implicit in all labels defined by following `FIELD` directives, until the next `MAP` directive. These register-relative labels can be quoted in load and store instructions.

`#` is a synonym for `FIELD`.

Examples

The following example shows how register-relative labels are defined using the `MAP` and `FIELD` directives:

```
    MAP     0,r9        ; set {VAR} to the address stored in R9
    FIELD   4           ; increment {VAR} by 4 bytes
Lab FIELD   4           ; set Lab to the address [R9 + 4]
                        ; and then increment {VAR} by 4 bytes
    LDR     r0,Lab      ; equivalent to LDR r0,[r9,#4]
```

When using the `MAP` and `FIELD` directives, you must ensure that the values are consistent in both passes. The following example shows a use of `MAP` and `FIELD` that causes inconsistent values for the symbol `x`. In the first pass `sym` is not defined, so `x` is at `0x04+R9`. In the second pass, `sym` is defined, so `x` is at `0x00+R0`. This example results in an assembly error.

```
    MAP 0, r0
    if :LNOT: :DEF: sym
      MAP 0, r9
      FIELD 4  ; x is at 0x04+R9 in first pass
    ENDIF
x   FIELD 4    ; x is at 0x00+R0 in second pass
sym LDR r0, x  ; inconsistent values for x results in assembly error
```

Related concepts

1.3 How the assembler works on page 1-49.

Related references

21.52 MAP on page 21-1703.
1.4 Directives that can be omitted in pass 2 of the assembler on page 1-51.

21.30 FRAME ADDRESS

The FRAME ADDRESS directive describes how to calculate the canonical frame address for the following instructions.

Syntax

```
FRAME ADDRESS reg{,offset}
```

where:

reg
is the register on which the canonical frame address is to be based. This is SP unless the function uses a separate frame pointer.

offset
is the offset of the canonical frame address from *reg*. If *offset* is zero, you can omit it.

Usage

Use FRAME ADDRESS if your code alters which register the canonical frame address is based on, or if it changes the offset of the canonical frame address from the register. You must use FRAME ADDRESS immediately after the instruction that changes the calculation of the canonical frame address.

You can only use FRAME ADDRESS in functions with FUNCTION and ENDFUNC or PROC and ENDP directives.

Note

If your code uses a single instruction to save registers and alter the stack pointer, you can use FRAME PUSH instead of using both FRAME ADDRESS and FRAME SAVE.

If your code uses a single instruction to load registers and alter the stack pointer, you can use FRAME POP instead of using both FRAME ADDRESS and FRAME RESTORE.

Example

```
_fn     FUNCTION        ; CFA (Canonical Frame Address) is value
                        ; of SP on entry to function
        PUSH    {r4,fp,ip,lr,pc}
        FRAME PUSH {r4,fp,ip,lr,pc}
        SUB     sp,sp,#4            ; CFA offset now changed
        FRAME ADDRESS sp,24         ; - so we correct it
        ADD     fp,sp,#20
        FRAME ADDRESS fp,4          ; New base register
        ; code using fp to base call-frame on, instead of SP
```

Related references

21.31 FRAME POP on page 21-1678.
21.32 FRAME PUSH on page 21-1679.

21.31 FRAME POP

The `FRAME POP` directive informs the assembler when the callee reloads registers.

Syntax

There are the following alternative syntaxes for `FRAME POP`:

```
FRAME POP {reglist}
FRAME POP {reglist},n
FRAME POP n
```

where:

reglist
: is a list of registers restored to the values they had on entry to the function. There must be at least one register in the list.

n
: is the number of bytes that the stack pointer moves.

Usage

`FRAME POP` is equivalent to a `FRAME ADDRESS` and a `FRAME RESTORE` directive. You can use it when a single instruction loads registers and alters the stack pointer.

You must use `FRAME POP` immediately after the instruction it refers to.

You can only use it within functions with `FUNCTION` and `ENDFUNC` or `PROC` and `ENDP` directives. You do not have to do this after the last instruction in a function.

If *n* is not specified or is zero, the assembler calculates the new offset for the canonical frame address from `{reglist}`. It assumes that:

- Each AArch32 register popped occupies four bytes on the stack.
- Each VFP single-precision register popped occupies four bytes on the stack, plus an extra four-byte word for each list.
- Each VFP double-precision register popped occupies eight bytes on the stack, plus an extra four-byte word for each list.

Related references

21.30 FRAME ADDRESS on page 21-1677.
21.34 FRAME RESTORE on page 21-1681.

21.32 FRAME PUSH

The `FRAME PUSH` directive informs the assembler when the callee saves registers, normally at function entry.

Syntax

There are the following alternative syntaxes for `FRAME PUSH`:

```
FRAME PUSH {reglist}

FRAME PUSH {reglist},n

FRAME PUSH n
```

where:

reglist
: is a list of registers stored consecutively below the canonical frame address. There must be at least one register in the list.

n
: is the number of bytes that the stack pointer moves.

Usage

`FRAME PUSH` is equivalent to a `FRAME ADDRESS` and a `FRAME SAVE` directive. You can use it when a single instruction saves registers and alters the stack pointer.

You must use `FRAME PUSH` immediately after the instruction it refers to.

You can only use it within functions with `FUNCTION` and `ENDFUNC` or `PROC` and `ENDP` directives.

If *n* is not specified or is zero, the assembler calculates the new offset for the canonical frame address from `{reglist}`. It assumes that:

- Each AArch32 register pushed occupies four bytes on the stack.
- Each VFP single-precision register pushed occupies four bytes on the stack, plus an extra four-byte word for each list.
- Each VFP double-precision register popped occupies eight bytes on the stack, plus an extra four-byte word for each list.

Example

```
p   PROC ; Canonical frame address is SP + 0
    EXPORT  p
    PUSH    {r4-r6,lr}
         ; SP has moved relative to the canonical frame address,
         ; and registers R4, R5, R6 and LR are now on the stack
    FRAME PUSH {r4-r6,lr}
         ; Equivalent to:
         ; FRAME ADDRESS    sp,16       ; 16 bytes in {R4-R6,LR}
         ; FRAME SAVE    {r4-r6,lr},-16
```

Related references

21.30 FRAME ADDRESS on page 21-1677.
21.36 FRAME SAVE on page 21-1683.

21.33 FRAME REGISTER

The `FRAME REGISTER` directive maintains a record of the locations of function arguments held in registers.

Syntax

```
FRAME REGISTER reg1, reg2
```

where:

reg1
: is the register that held the argument on entry to the function.

reg2
: is the register in which the value is preserved.

Usage

Use the `FRAME REGISTER` directive when you use a register to preserve an argument that was held in a different register on entry to a function.

You can only use it within functions with `FUNCTION` and `ENDFUNC` or `PROC` and `ENDP` directives.

21.34 FRAME RESTORE

The `FRAME RESTORE` directive informs the assembler that the contents of specified registers have been restored to the values they had on entry to the function.

Syntax

```
FRAME RESTORE {reglist}
```

where:

reglist
is a list of registers whose contents have been restored. There must be at least one register in the list.

Usage

You can only use `FRAME RESTORE` within functions with `FUNCTION` and `ENDFUNC` or `PROC` and `ENDP` directives. Use it immediately after the callee reloads registers from the stack. You do not have to do this after the last instruction in a function.

reglist can contain integer registers or floating-point registers, but not both.

Note

If your code uses a single instruction to load registers and alter the stack pointer, you can use `FRAME POP` instead of using both `FRAME RESTORE` and `FRAME ADDRESS`.

Related references

21.31 FRAME POP on page 21-1678.

21.35 FRAME RETURN ADDRESS

The `FRAME RETURN ADDRESS` directive provides for functions that use a register other than LR for their return address.

Syntax

```
FRAME RETURN ADDRESS reg
```

where:

reg
is the register used for the return address.

Usage

Use the `FRAME RETURN ADDRESS` directive in any function that does not use LR for its return address. Otherwise, a debugger cannot backtrace through the function.

You can only use `FRAME RETURN ADDRESS` within functions with `FUNCTION` and `ENDFUNC` or `PROC` and `ENDP` directives. Use it immediately after the `FUNCTION` or `PROC` directive that introduces the function.

Note

Any function that uses a register other than LR for its return address is not AAPCS compliant. Such a function must not be exported.

21.36 FRAME SAVE

The FRAME SAVE directive describes the location of saved register contents relative to the canonical frame address.

Syntax

```
FRAME SAVE {reglist}, offset
```

where:

reglist
is a list of registers stored consecutively starting at *offset* from the canonical frame address. There must be at least one register in the list.

Usage

You can only use FRAME SAVE within functions with FUNCTION and ENDFUNC or PROC and ENDP directives.

Use it immediately after the callee stores registers onto the stack.

reglist can include registers which are not required for backtracing. The assembler determines which registers it requires to record in the DWARF call frame information.

Note

If your code uses a single instruction to save registers and alter the stack pointer, you can use FRAME PUSH instead of using both FRAME SAVE and FRAME ADDRESS.

Related references

21.32 FRAME PUSH on page 21-1679.

21.37 FRAME STATE REMEMBER

The FRAME STATE REMEMBER directive saves the current information on how to calculate the canonical frame address and locations of saved register values.

Syntax

```
FRAME STATE REMEMBER
```

Usage

During an inline exit sequence the information about calculation of canonical frame address and locations of saved register values can change. After the exit sequence another branch can continue using the same information as before. Use FRAME STATE REMEMBER to preserve this information, and FRAME STATE RESTORE to restore it.

These directives can be nested. Each FRAME STATE RESTORE directive must have a corresponding FRAME STATE REMEMBER directive.

You can only use FRAME STATE REMEMBER within functions with FUNCTION and ENDFUNC or PROC and ENDP directives.

Example

```
        ; function code
        FRAME STATE REMEMBER
            ; save frame state before in-line exit sequence
        POP     {r4-r6,pc}
            ; do not have to FRAME POP here, as control has
            ; transferred out of the function
        FRAME STATE RESTORE
            ; end of exit sequence, so restore state
exitB   ; code for exitB
        POP     {r4-r6,pc}
        ENDP
```

Related references

21.38 FRAME STATE RESTORE on page 21-1685.
21.41 FUNCTION or PROC on page 21-1688.

21.38 FRAME STATE RESTORE

The `FRAME STATE RESTORE` directive restores information about how to calculate the canonical frame address and locations of saved register values.

Syntax

```
FRAME STATE RESTORE
```

Usage

You can only use `FRAME STATE RESTORE` within functions with `FUNCTION` and `ENDFUNC` or `PROC` and `ENDP` directives.

Related references

21.37 FRAME STATE REMEMBER on page 21-1684.
21.41 FUNCTION or PROC on page 21-1688.

21.39 FRAME UNWIND ON

The `FRAME UNWIND ON` directive instructs the assembler to produce unwind tables for this and subsequent functions.

Syntax

```
FRAME UNWIND ON
```

Usage

You can use this directive outside functions. In this case, the assembler produces unwind tables for all following functions until it reaches a `FRAME UNWIND OFF` directive.

Note

A `FRAME UNWIND` directive is not sufficient to turn on exception table generation. Furthermore a `FRAME UNWIND` directive, without other `FRAME` directives, is not sufficient information for the assembler to generate the unwind information.

Related references

11.26 --exceptions, --no_exceptions on page 11-254.
11.27 --exceptions_unwind, --no_exceptions_unwind on page 11-255.

21.40 FRAME UNWIND OFF

The `FRAME UNWIND OFF` directive instructs the assembler to produce no unwind tables for this and subsequent functions.

Syntax

```
FRAME UNWIND OFF
```

Usage

You can use this directive outside functions. In this case, the assembler produces no unwind tables for all following functions until it reaches a `FRAME UNWIND ON` directive.

Related references

11.26 --exceptions, --no_exceptions on page 11-254.
11.27 --exceptions_unwind, --no_exceptions_unwind on page 11-255.

21.41 FUNCTION or PROC

The FUNCTION directive marks the start of a function. PROC is a synonym for FUNCTION.

Syntax

label FUNCTION [{*reglist1*} [, {*reglist2*}]]

where:

reglist1
is an optional list of callee-saved AArch32 registers. If *reglist1* is not present, and your debugger checks register usage, it assumes that the AAPCS is in use. If you use empty brackets, this informs the debugger that all AArch32 registers are caller-saved.

reglist2
is an optional list of callee-saved VFP registers. If you use empty brackets, this informs the debugger that all VFP registers are caller-saved.

Usage

Use FUNCTION to mark the start of functions. The assembler uses FUNCTION to identify the start of a function when producing DWARF call frame information for ELF.

FUNCTION sets the canonical frame address to be R13 (SP), and the frame state stack to be empty.

Each FUNCTION directive must have a matching ENDFUNC directive. You must not nest FUNCTION and ENDFUNC pairs, and they must not contain PROC or ENDP directives.

You can use the optional *reglist* parameters to inform the debugger about an alternative procedure call standard, if you are using your own. Not all debuggers support this feature. See your debugger documentation for details.

If you specify an empty *reglist*, using {}, this indicates that all registers for the function are caller-saved. Typically you do this when writing a reset vector where the values in all registers are unknown on execution. This avoids problems in a debugger if it tries to construct a backtrace from the values in the registers.

Note

FUNCTION does not automatically cause alignment to a word boundary (or halfword boundary for T32). Use ALIGN if necessary to ensure alignment, otherwise the call frame might not point to the start of the function.

Examples

```
       ALIGN        ; Ensures alignment.
dadd   FUNCTION     ; Without the ALIGN directive this might not be word-aligned.
       EXPORT  dadd
       PUSH       {r4-r6,lr}    ; This line automatically word-aligned.
       FRAME PUSH {r4-r6,lr}
       ; subroutine body
       POP        {r4-r6,pc}
       ENDFUNC
func6  PROC {r4-r8,r12},{D1-D3} ; Non-AAPCS-conforming function.
       ...
       ENDP
func7  FUNCTION {}  ; Another non-AAPCS-conforming function.
       ...
       ENDFUNC
```

Related references

21.38 FRAME STATE RESTORE on page 21-1685.

21.30 FRAME ADDRESS on page 21-1677.

21.5 ALIGN on page 21-1648.

21.42 GBLA, GBLL, and GBLS

The GBLA, GBLL, and GBLS directives declare and initialize global variables.

Syntax

gblx variable

where:

gblx
is one of GBLA, GBLL, or GBLS.

variable
is the name of the variable. *variable* must be unique among symbols within a source file.

Usage

The GBLA directive declares a global arithmetic variable, and initializes its value to 0.

The GBLL directive declares a global logical variable, and initializes its value to {FALSE}.

The GBLS directive declares a global string variable and initializes its value to a null string, "".

Using one of these directives for a variable that is already defined re-initializes the variable.

The scope of the variable is limited to the source file that contains it.

Set the value of the variable with a SETA, SETL, or SETS directive.

Global variables can also be set with the --predefine assembler command-line option.

Examples

The following example declares a variable objectsize, sets the value of objectsize to 0xFF, and then uses it later in a SPACE directive:

```
            GBLA    objectsize    ; declare the variable name
objectsize  SETA    0xFF          ; set its value
            .
            .                     ; other code
            .
            SPACE   objectsize    ; quote the variable
```

The following example shows how to declare and set a variable when you invoke armasm. Use this when you want to set the value of a variable at assembly time. --pd is a synonym for --predefine.

```
armasm --cpu=8-A.32 --predefine "objectsize SETA 0xFF" sourcefile -o objectfile
```

Related references

21.49 LCLA, LCLL, and LCLS on page 21-1698.
21.63 SETA, SETL, and SETS on page 21-1716.
11.54 --predefine "directive" on page 11-282.

21.43 GET or INCLUDE

The GET directive includes a file within the file being assembled. The included file is assembled at the location of the GET directive. INCLUDE is a synonym for GET.

Syntax

GET *filename*

where:

filename
is the name of the file to be included in the assembly. The assembler accepts pathnames in either UNIX or MS-DOS format.

Usage

GET is useful for including macro definitions, EQUs, and storage maps in an assembly. When assembly of the included file is complete, assembly continues at the line following the GET directive.

By default the assembler searches the current place for included files. The current place is the directory where the calling file is located. Use the -i assembler command line option to add directories to the search path. File names and directory names containing spaces must not be enclosed in double quotes (" ").

The included file can contain additional GET directives to include other files.

If the included file is in a different directory from the current place, this becomes the current place until the end of the included file. The previous current place is then restored.

You cannot use GET to include object files.

Examples

```
  AREA   Example, CODE, READONLY
  GET    file1.s                 ; includes file1 if it exists in the current place
  GET    c:\project\file2.s      ; includes file2
  GET    c:\Program files\file3.s ; space is permitted
```

Related references

21.46 INCBIN on page 21-1695.
21.2 About assembly control directives on page 21-1645.

21.44 IF, ELSE, ENDIF, and ELIF

The `IF`, `ELSE`, `ENDIF`, and `ELIF` directives allow you to conditionally assemble sequences of instructions and directives.

Syntax

```
IF logical-expression
    ...;code
{ELSE
    ...;code}
ENDIF
```

where:

logical-expression
is an expression that evaluates to either `{TRUE}` or `{FALSE}`.

Usage

Use `IF` with `ENDIF`, and optionally with `ELSE`, for sequences of instructions or directives that are only to be assembled or acted on under a specified condition.

`IF...ENDIF` conditions can be nested.

The `IF` directive introduces a condition that controls whether to assemble a sequence of instructions and directives. `[` is a synonym for `IF`.

The `ELSE` directive marks the beginning of a sequence of instructions or directives that you want to be assembled if the preceding condition fails. `|` is a synonym for `ELSE`.

The `ENDIF` directive marks the end of a sequence of instructions or directives that you want to be conditionally assembled. `]` is a synonym for `ENDIF`.

The `ELIF` directive creates a structure equivalent to `ELSE IF`, without the requirement for nesting or repeating the condition.

Using ELIF

Without using `ELIF`, you can construct a nested set of conditional instructions like this:

```
IF logical-expression
        instructions
ELSE
    IF logical-expression2
        instructions
    ELSE
        IF logical-expression3
            instructions
        ENDIF
    ENDIF
ENDIF
```

A nested structure like this can be nested up to 256 levels deep.

You can write the same structure more simply using `ELIF`:

```
IF logical-expression
    instructions
ELIF logical-expression2
    instructions
ELIF logical-expression3
    instructions
ENDIF
```

This structure only adds one to the current nesting depth, for the `IF...ENDIF` pair.

Examples

The following example assembles the first set of instructions if `NEWVERSION` is defined, or the alternative set otherwise:

Assembly conditional on a variable being defined

```
        IF :DEF:NEWVERSION
            ; first set of instructions or directives
        ELSE
            ; alternative set of instructions or directives
        ENDIF
```

Invoking `armasm` as follows defines `NEWVERSION`, so the first set of instructions and directives are assembled:

```
armasm --cpu=8-A.32 --predefine "NEWVERSION SETL {TRUE}" test.s
```

Invoking `armasm` as follows leaves `NEWVERSION` undefined, so the second set of instructions and directives are assembled:

```
armasm --cpu=8-A.32 test.s
```

The following example assembles the first set of instructions if `NEWVERSION` has the value `{TRUE}`, or the alternative set otherwise:

Assembly conditional on a variable value

```
        IF NEWVERSION = {TRUE}
            ; first set of instructions or directives
        ELSE
            ; alternative set of instructions or directives
        ENDIF
```

Invoking `armasm` as follows causes the first set of instructions and directives to be assembled:

```
armasm --cpu=8-A.32 --predefine "NEWVERSION SETL {TRUE}" test.s
```

Invoking `armasm` as follows causes the second set of instructions and directives to be assembled:

```
armasm --cpu=8-A.32 --predefine "NEWVERSION SETL {FALSE}" test.s
```

Related references

12.25 Relational operators on page 12-322.
21.2 About assembly control directives on page 21-1645.

21.45 IMPORT and EXTERN

The IMPORT and EXTERN directives provide the assembler with a name that is not defined in the current assembly.

Syntax

```
directive symbol {[SIZE=n]}

directive symbol {[type]}

directive symbol [attr{,type}{,SIZE=n}]

directive symbol [WEAK {,attr}{,type}{,SIZE=n}]
```

where:

directive
: can be either:

  IMPORT
  : imports the symbol unconditionally.

  EXTERN
  : imports the symbol only if it is referred to in the current assembly.

symbol
: is a symbol name defined in a separately assembled source file, object file, or library. The symbol name is case-sensitive.

WEAK
: prevents the linker generating an error message if the symbol is not defined elsewhere. It also prevents the linker searching libraries that are not already included.

attr
: can be any one of:

  DYNAMIC
  : sets the ELF symbol visibility to STV_DEFAULT.

  PROTECTED
  : sets the ELF symbol visibility to STV_PROTECTED.

  HIDDEN
  : sets the ELF symbol visibility to STV_HIDDEN.

  INTERNAL
  : sets the ELF symbol visibility to STV_INTERNAL.

type
: specifies the symbol type:

  DATA
  : *symbol* is treated as data when the source is assembled and linked.

  CODE
  : *symbol* is treated as code when the source is assembled and linked.

  ELFTYPE=*n*
  : *symbol* is treated as a particular ELF symbol, as specified by the value of *n*, where *n* can be any number from 0 to 15.

  If unspecified, the linker determines the most appropriate type.

n
: specifies the size and can be any 32-bit value. If the SIZE attribute is not specified, the assembler calculates the size:

  - For PROC and FUNCTION symbols, the size is set to the size of the code until its ENDP or ENDFUNC.
  - For other symbols, the size is the size of instruction or data on the same source line. If there is no instruction or data, the size is zero.

Usage

The name is resolved at link time to a symbol defined in a separate object file. The symbol is treated as a program address. If `[WEAK]` is not specified, the linker generates an error if no corresponding symbol is found at link time.

If `[WEAK]` is specified and no corresponding symbol is found at link time:

- If the reference is the destination of a `B` or `BL` instruction, the value of the symbol is taken as the address of the following instruction. This makes the `B` or `BL` instruction effectively a `NOP`.
- Otherwise, the value of the symbol is taken as zero.

Example

The example tests to see if the C++ library has been linked, and branches conditionally on the result.

```
    AREA    Example, CODE, READONLY
    EXTERN  __CPP_INITIALIZE[WEAK]  ; If C++ library linked, gets the
                                    ; address of __CPP_INITIALIZE
                                    ; function.
    LDR     r0,=__CPP_INITIALIZE    ; If not linked, address is zeroed.
    CMP     r0,#0                   ; Test if zero.
    BEQ     nocplusplus             ; Branch on the result.
```

The following examples show the use of the `SIZE` attribute:

```
    EXTERN symA [SIZE=4]
    EXTERN symA [DATA, SIZE=4]
```

Related references

21.27 EXPORT or GLOBAL on page 21-1673.

Related information

ELF for the Arm Architecture.

21.46 INCBIN

The `INCBIN` directive includes a file within the file being assembled. The file is included as it is, without being assembled.

Syntax

`INCBIN` *`filename`*

where:

`filename`
is the name of the file to be included in the assembly. The assembler accepts pathnames in either UNIX or MS-DOS format.

Usage

You can use `INCBIN` to include executable files, literals, or any arbitrary data. The contents of the file are added to the current ELF section, byte for byte, without being interpreted in any way. Assembly continues at the line following the `INCBIN` directive.

By default, the assembler searches the current place for included files. The current place is the directory where the calling file is located. Use the `-i` assembler command-line option to add directories to the search path. File names and directory names containing spaces must not be enclosed in double quotes (" ").

Example

```
    AREA    Example, CODE, READONLY
    INCBIN  file1.dat             ; Includes file1 if it exists in the current place
    INCBIN  c:\project\file2.txt   ; Includes file2.
```

21.47 INFO

The `INFO` directive supports diagnostic generation on either pass of the assembly.

Syntax

`INFO` *`numeric-expression, string-expression{, severity}`*

where:

`numeric-expression`
: is a numeric expression that is evaluated during assembly. If the expression evaluates to zero:
  - No action is taken during pass one.
  - *`string-expression`* is printed as a warning during pass two if *`severity`* is 1.
  - *`string-expression`* is printed as a message during pass two if *`severity`* is 0 or not specified.

  If the expression does not evaluate to zero:
  - *`string-expression`* is printed as an error message and the assembly fails irrespective of whether *`severity`* is specified or not (non-zero values for *`severity`* are reserved in this case).

`string-expression`
: is an expression that evaluates to a string.

`severity`
: is an optional number that controls the severity of the message. Its value can be either 0 or 1. All other values are reserved.

Usage

`INFO` provides a flexible means of creating custom error messages.

`!` is very similar to `INFO`, but has less detailed reporting.

Examples

```
INFO    0, "Version 1.0"
IF endofdata <= label1
    INFO    4, "Data overrun at label1"
ENDIF
```

Related concepts

12.12 String expressions on page 12-309.
12.14 Numeric expressions on page 12-311.

Related references

21.8 ASSERT on page 21-1654.

21.48 KEEP

The KEEP directive instructs the assembler to retain named local labels in the symbol table in the object file.

Syntax

```
KEEP {label}
```

where:

label
is the name of the local label to keep. If *label* is not specified, all named local labels are kept except register-relative labels.

Usage

By default, the only labels that the assembler describes in its output object file are:

- Exported labels.
- Labels that are relocated against.

Use KEEP to preserve local labels. This can help when debugging. Kept labels appear in the Arm debuggers and in linker map files.

KEEP cannot preserve register-relative labels or numeric local labels.

Example

```
label   ADC     r2,r3,r4
        KEEP    label       ; makes label available to debuggers
        ADD     r2,r2,r5
```

Related concepts

12.10 Numeric local labels on page 12-307.

Related references

21.52 MAP on page 21-1703.

21.49 LCLA, LCLL, and LCLS

The LCLA, LCLL, and LCLS directives declare and initialize local variables.

Syntax

lclx variable

where:

lclx
is one of LCLA, LCLL, or LCLS.

variable
is the name of the variable. *variable* must be unique within the macro that contains it.

Usage

The LCLA directive declares a local arithmetic variable, and initializes its value to 0.

The LCLL directive declares a local logical variable, and initializes its value to {FALSE}.

The LCLS directive declares a local string variable, and initializes its value to a null string, "".

Using one of these directives for a variable that is already defined re-initializes the variable.

The scope of the variable is limited to a particular instantiation of the macro that contains it.

Set the value of the variable with a SETA, SETL, or SETS directive.

Example

```
        MACRO                           ; Declare a macro
$label  message $a                      ; Macro prototype line
        LCLS    err                     ; Declare local string
                                        ; variable err.
err     SETS    "error no: "            ; Set value of err
$label  ; code
        INFO    0, "err":CC::STR:$a     ; Use string
        MEND
```

Related references

21.42 GBLA, GBLL, and GBLS on page 21-1689.
21.63 SETA, SETL, and SETS on page 21-1716.
21.51 MACRO and MEND on page 21-1700.

21.50 LTORG

The LTORG directive instructs the assembler to assemble the current literal pool immediately.

Syntax

```
LTORG
```

Usage

The assembler assembles the current literal pool at the end of every code section. The end of a code section is determined by the AREA directive at the beginning of the following section, or the end of the assembly.

These default literal pools can sometimes be out of range of some LDR, VLDR, and WLDR pseudo-instructions. Use LTORG to ensure that a literal pool is assembled within range.

Large programs can require several literal pools. Place LTORG directives after unconditional branches or subroutine return instructions so that the processor does not attempt to execute the constants as instructions.

The assembler word-aligns data in literal pools.

Example

```
        AREA    Example, CODE, READONLY
start   BL      func1
func1                           ; function body
        ; code
        LDR     r1,=0x55555555  ; => LDR R1, [pc, #offset to Literal Pool 1]
        ; code
        MOV     pc,lr           ; end function
        LTORG                   ; Literal Pool 1 contains literal &55555555.
data    SPACE   4200            ; Clears 4200 bytes of memory starting at current location.
        END                     ; Default literal pool is empty.
```

Related references

13.54 LDR pseudo-instruction on page 13-417.
14.54 VLDR pseudo-instruction on page 14-661.

21.51 MACRO and MEND

The `MACRO` directive marks the start of the definition of a macro. Macro expansion terminates at the `MEND` directive.

Syntax

These two directives define a macro. The syntax is:

```
     MACRO
{$label}  macroname{$cond} {$parameter{,$parameter}...}
     ; code
     MEND
```

where:

$label
: is a parameter that is substituted with a symbol given when the macro is invoked. The symbol is usually a label.

macroname
: is the name of the macro. It must not begin with an instruction or directive name.

$cond
: is a special parameter designed to contain a condition code. Values other than valid condition codes are permitted.

$parameter
: is a parameter that is substituted when the macro is invoked. A default value for a parameter can be set using this format:

```
$parameter="default value"
```

Double quotes must be used if there are any spaces within, or at either end of, the default value.

Usage

If you start any `WHILE...WEND` loops or `IF...ENDIF` conditions within a macro, they must be closed before the `MEND` directive is reached. You can use `MEXIT` to enable an early exit from a macro, for example, from within a loop.

Within the macro body, parameters such as *$label*, *$parameter* or *$cond* can be used in the same way as other variables. They are given new values each time the macro is invoked. Parameters must begin with `$` to distinguish them from ordinary symbols. Any number of parameters can be used.

$label is optional. It is useful if the macro defines internal labels. It is treated as a parameter to the macro. It does not necessarily represent the first instruction in the macro expansion. The macro defines the locations of any labels.

Use `|` as the argument to use the default value of a parameter. An empty string is used if the argument is omitted.

In a macro that uses several internal labels, it is useful to define each internal label as the base label with a different suffix.

Use a dot between a parameter and following text, or a following parameter, if a space is not required in the expansion. Do not use a dot between preceding text and a parameter.

You can use the *$cond* parameter for condition codes. Use the unary operator `:REVERSE_CC:` to find the inverse condition code, and `:CC_ENCODING:` to find the 4-bit encoding of the condition code.

Macros define the scope of local variables.

Macros can be nested.

Examples

A macro that uses internal labels to implement loops:

```
 ; macro definition
                MACRO                   ; start macro definition
$label          xmac    $p1,$p2
                ; code
$label.loop1    ; code
                ; code
                BGE     $label.loop1
$label.loop2    ; code
                BL      $p1
                BGT     $label.loop2
                ; code
                ADR     $p2
                ; code
                MEND                    ; end macro definition
 ; macro invocation
abc             xmac    subr1,de        ; invoke macro
                ; code                  ; this is what is
abcloop1        ; code                  ; is produced when
                ; code                  ; the xmac macro is
                BGE     abcloop1        ; expanded
abcloop2        ; code
                BL      subr1
                BGT     abcloop2
                ; code
                ADR     de
                ; code
```

A macro that produces assembly-time diagnostics:

```
        MACRO                           ; Macro definition
        diagnose  $param1="default"     ; This macro produces
        INFO      0,"$param1"           ; assembly-time diagnostics
        MEND                            ; (on second assembly pass)
 ; macro expansion
        diagnose              ; Prints blank line at assembly-time
        diagnose "hello"      ; Prints "hello" at assembly-time
        diagnose |            ; Prints "default" at assembly-time
```

When variables are being passed in as arguments, use of | might leave some variables unsubstituted. To work around this, define the | in a `LCLS` or `GBLS` variable and pass this variable as an argument instead of |. For example:

```
        MACRO                   ; Macro definition
        m2 $a,$b=r1,$c          ; The default value for $b is r1
        add $a,$b,$c            ; The macro adds $b and $c and puts result in $a.
        MEND                    ; Macro end
        MACRO                   ; Macro definition
        m1 $a,$b                ; This macro adds $b to r1 and puts result in $a.
        LCLS def                ; Declare a local string variable for |
def     SETS "|"                ; Define |
        m2 $a,$def,$b           ; Invoke macro m2 with $def instead of |
                                ; to use the default value for the second argument.
        MEND                    ; Macro end
```

A macro that uses a condition code parameter:

```
        AREA    codx, CODE, READONLY
; macro definition
        MACRO
        Return$cond
        [ {ARCHITECTURE} <> "4"
          BX$cond lr
          |
          MOV$cond pc,lr
        ]
        MEND
; macro invocation
fun     PROC
        CMP      r0,#0
        MOVEQ    r0,#1
        ReturnEQ
        MOV      r0,#0
        Return
        ENDP
        END
```

Related concepts

6.22 Use of macros on page 6-130.
12.4 Assembly time substitution of variables on page 12-301.

Related references

21.53 MEXIT on page 21-1704.
21.42 GBLA, GBLL, and GBLS on page 21-1689.
21.49 LCLA, LCLL, and LCLS on page 21-1698.

21.52 MAP

The MAP directive sets the origin of a storage map to a specified address.

Syntax

```
MAP expr{,base-register}
```

where:

expr
is a numeric or PC-relative expression:
- If *base-register* is not specified, *expr* evaluates to the address where the storage map starts. The storage map location counter is set to this address.
- If *expr* is PC-relative, you must have defined the label before you use it in the map. The map requires the definition of the label during the first pass of the assembler.

base-register
specifies a register. If *base-register* is specified, the address where the storage map starts is the sum of *expr*, and the value in *base-register* at runtime.

Usage

Use the MAP directive in combination with the FIELD directive to describe a storage map.

Specify *base-register* to define register-relative labels. The base register becomes implicit in all labels defined by following FIELD directives, until the next MAP directive. The register-relative labels can be used in load and store instructions.

The MAP directive can be used any number of times to define multiple storage maps.

The storage-map location counter, {VAR}, is set to the same address as that specified by the MAP directive. The {VAR} counter is set to zero before the first MAP directive is used.

^ is a synonym for MAP.

Examples

```
        MAP     0,r9
        MAP     0xff,r9
```

Related concepts

1.3 How the assembler works on page 1-49.

Related references

21.29 FIELD on page 21-1676.
1.4 Directives that can be omitted in pass 2 of the assembler on page 1-51.

21.53 MEXIT

The `MEXIT` directive exits a macro definition before the end.

Usage

Use `MEXIT` when you require an exit from within the body of a macro. Any unclosed `WHILE...WEND` loops or `IF...ENDIF` conditions within the body of the macro are closed by the assembler before the macro is exited.

Example

```
        MACRO
$abc    example abc     $param1,$param2
        ; code
        WHILE condition1
            ; code
            IF condition2
                ; code
                MEXIT
            ELSE
                ; code
            ENDIF
        WEND
        ; code
        MEND
```

Related references

21.51 MACRO and MEND on page 21-1700.

21.54 NOFP

The NOFP directive ensures that there are no floating-point instructions in an assembly language source file.

Syntax

```
NOFP
```

Usage

Use NOFP to ensure that no floating-point instructions are used in situations where there is no support for floating-point instructions either in software or in target hardware.

If a floating-point instruction occurs after the NOFP directive, an `Unknown opcode` error is generated and the assembly fails.

If a NOFP directive occurs after a floating-point instruction, the assembler generates the error:

```
Too late to ban floating point instructions
```

and the assembly fails.

21.55 OPT

The OPT directive sets listing options from within the source code.

Syntax

OPT *n*

where:

n
is the OPT directive setting. The following table lists the valid settings:

Table 21-2 OPT directive settings

OPT n	Effect
1	Turns on normal listing.
2	Turns off normal listing.
4	Page throw. Issues an immediate form feed and starts a new page.
8	Resets the line number counter to zero.
16	Turns on listing for SET, GBL and LCL directives.
32	Turns off listing for SET, GBL and LCL directives.
64	Turns on listing of macro expansions.
128	Turns off listing of macro expansions.
256	Turns on listing of macro invocations.
512	Turns off listing of macro invocations.
1024	Turns on the first pass listing.
2048	Turns off the first pass listing.
4096	Turns on listing of conditional directives.
8192	Turns off listing of conditional directives.
16384	Turns on listing of MEND directives.
32768	Turns off listing of MEND directives.

Usage

Specify the --list= assembler option to turn on listing.

By default the --list= option produces a normal listing that includes variable declarations, macro expansions, call-conditioned directives, and MEND directives. The listing is produced on the second pass only. Use the OPT directive to modify the default listing options from within your code.

You can use OPT to format code listings. For example, you can specify a new page before functions and sections.

Example

```
        AREA    Example, CODE, READONLY
start   ; code
        ; code
        BL      func1
        ; code
        OPT 4               ; places a page break before func1
func1   ; code
```

Related references

11.40 --list=file on page 11-268.

21.56 QN, DN, and SN

The QN, DN, and SN directives define names for Advanced SIMD and floating-point registers.

Syntax

name directive expr{*.type*}{[*x*]}

where:

directive
is QN, DN, or SN.

name
is the name to be assigned to the extension register. *name* cannot be the same as any of the predefined names.

expr
Can be:

- An expression that evaluates to a number in the range:
  - 0-15 if you are using QN in A32/T32 Advanced SIMD code.
  - 0-31 otherwise.
- A predefined register name, or a register name that has already been defined in a previous directive.

type
is any Advanced SIMD or floating-point datatype.

[*x*]
is only available for Advanced SIMD code. [*x*] is a scalar index into a register.

type and [*x*] are *Extended notation*.

Usage

Use QN, DN, or SN to allocate convenient names to extension registers, to help you to remember what you use each one for.

The QN directive defines a name for a specified 128-bit extension register.

The DN directive defines a name for a specified 64-bit extension register.

The SN directive defines a name for a specified single-precision floating-point register.

Note

Avoid conflicting uses of the same register under different names.

You cannot specify a vector length in a DN or SN directive.

Examples

```
energy  DN  6   ; defines energy as a symbol for
                ; floating-point double-precision register 6
mass    SN  16  ; defines mass as a symbol for
                ; floating-point single-precision register 16
```

Extended notation examples

```
varA    DN      d1.U16
varB    DN      d2.U16
varC    DN      d3.U16
        VADD    varA,varB,varC       ; VADD.U16 d1,d2,d3
index   DN      d4.U16[0]
result  QN      q5.I32
        VMULL   result,varA,index    ; VMULL.U16 q5,d1,d4[0]
```

Related concepts

9.10 Advanced SIMD data types in A32/T32 instructions on page 9-194.

9.16 Extended notation extension for Advanced SIMD in A32/T32 code on page 9-200.

Related references

3.7 Predeclared core register names in AArch32 state on page 3-71.

3.8 Predeclared extension register names in AArch32 state on page 3-72.

21.57 RELOC

The RELOC directive explicitly encodes an ELF relocation in an object file.

Syntax

```
RELOC n, symbol

RELOC n
```

where:

n
must be an integer in the range 0 to 255 or one of the relocation names defined in the *Application Binary Interface for the Arm® Architecture*.

symbol
can be any PC-relative label.

Usage

Use RELOC *n*, *symbol* to create a relocation with respect to the address labeled by *symbol*.

If used immediately after an A32 or T32 instruction, RELOC results in a relocation at that instruction. If used immediately after a DCB, DCW, or DCD, or any other data generating directive, RELOC results in a relocation at the start of the data. Any addend to be applied must be encoded in the instruction or in the data.

If the assembler has already emitted a relocation at that place, the relocation is updated with the details in the RELOC directive, for example:

```
DCD     sym2 ; R_ARM_ABS32 to sym32
RELOC   55   ; ... makes it R_ARM_ABS32_NOI
```

RELOC is faulted in all other cases, for example, after any non-data generating directive, LTORG, ALIGN, or as the first thing in an AREA.

Use RELOC *n* to create a relocation with respect to the anonymous symbol, that is, symbol 0 of the symbol table. If you use RELOC *n* without a preceding assembler generated relocation, the relocation is with respect to the anonymous symbol.

Examples

```
IMPORT  impsym
LDR     r0,[pc,#-8]
RELOC   4, impsym
DCD     0
RELOC   2, sym
DCD     0,1,2,3,4       ; the final word is relocated
RELOC   38,sym2         ; R_ARM_TARGET1
DCD     impsym
RELOC   R_ARM_TARGET1   ; relocation code 38
```

Related information

Application Binary Interface for the Arm Architecture.

21.58 REQUIRE

The `REQUIRE` directive specifies a dependency between sections.

Syntax

`REQUIRE` *label*

where:

label
is the name of the required label.

Usage

Use `REQUIRE` to ensure that a related section is included, even if it is not directly called. If the section containing the `REQUIRE` directive is included in a link, the linker also includes the section containing the definition of the specified label.

21.59 REQUIRE8 and PRESERVE8

The REQUIRE8 and PRESERVE8 directives specify that the current file requires or preserves eight-byte alignment of the stack.

Note

This directive is required to support non-ABI conforming toolchains. It has no effect on AArch64 assembly and is not required when targeting AArch64.

Syntax

```
REQUIRE8 {bool}

PRESERVE8 {bool}
```

where:

bool
is an optional Boolean constant, either {TRUE} or {FALSE}.

Usage

Where required, if your code preserves eight-byte alignment of the stack, use PRESERVE8 to set the PRES8 build attribute on your file. If your code does not preserve eight-byte alignment of the stack, use PRESERVE8 {FALSE} to ensure that the PRES8 build attribute is not set. Use REQUIRE8 to set the REQ8 build attribute. If there are multiple REQUIRE8 or PRESERVE8 directives in a file, the assembler uses the value of the last directive.

The linker checks that any code that requires eight-byte alignment of the stack is only called, directly or indirectly, by code that preserves eight-byte alignment of the stack.

Note

If you omit both PRESERVE8 and PRESERVE8 {FALSE}, the assembler decides whether to set the PRES8 build attribute or not, by examining instructions that modify the SP. Arm recommends that you specify PRESERVE8 explicitly.

You can enable a warning by using the --diag_warning 1546 option when invoking armasm.

This gives you warnings like:

```
"test.s", line 37: Warning: A1546W: Stack pointer update potentially breaks 8 byte stack
alignment
       37 00000044         STMFD    sp!,{r2,r3,lr}
```

Examples

```
REQUIRE8
REQUIRE8     {TRUE}      ; equivalent to REQUIRE8
REQUIRE8     {FALSE}     ; equivalent to absence of REQUIRE8
PRESERVE8    {TRUE}      ; equivalent to PRESERVE8
PRESERVE8    {FALSE}     ; NOT exactly equivalent to absence of PRESERVE8
```

Related references

11.21 --diag_warning=tag[,tag,...] on page 11-249.

Related information

Eight-byte Stack Alignment.

21.60 RLIST

The RLIST (register list) directive gives a name to a set of general-purpose registers in A32/T32 code.

Syntax

name RLIST {*list-of-registers*}

where:

name
is the name to be given to the set of registers. *name* cannot be the same as any of the predefined names.

list-of-registers
is a comma-delimited list of register names and register ranges. The register list must be enclosed in braces.

Usage

Use RLIST to give a name to a set of registers to be transferred by the LDM or STM instructions.

LDM and STM always put the lowest physical register numbers at the lowest address in memory, regardless of the order they are supplied to the LDM or STM instruction. If you have defined your own symbolic register names it can be less apparent that a register list is not in increasing register order.

Use the --diag_warning 1206 assembler option to ensure that the registers in a register list are supplied in increasing register order. If registers are not supplied in increasing register order, a warning is issued.

Example

```
Context RLIST   {r0-r6,r8,r10-r12,pc}
```

Related references

3.7 Predeclared core register names in AArch32 state on page 3-71.
3.8 Predeclared extension register names in AArch32 state on page 3-72.

21.61 RN

The RN directive defines a name for a specified register.

Syntax

name RN *expr*

where:

name
is the name to be assigned to the register. *name* cannot be the same as any of the predefined names.

expr
evaluates to a register number from 0 to 15.

Usage

Use RN to allocate convenient names to registers, to help you to remember what you use each register for. Be careful to avoid conflicting uses of the same register under different names.

Examples

```
regname     RN  11  ; defines regname for register 11
sqr4        RN  r6  ; defines sqr4 for register 6
```

Related references

3.7 Predeclared core register names in AArch32 state on page 3-71.
3.8 Predeclared extension register names in AArch32 state on page 3-72.

21.62 ROUT

The ROUT directive marks the boundaries of the scope of numeric local labels.

Syntax

{*name*} ROUT

where:

name
is the name to be assigned to the scope.

Usage

Use the ROUT directive to limit the scope of numeric local labels. This makes it easier for you to avoid referring to a wrong label by accident. The scope of numeric local labels is the whole area if there are no ROUT directives in it.

Use the *name* option to ensure that each reference is to the correct numeric local label. If the name of a label or a reference to a label does not match the preceding ROUT directive, the assembler generates an error message and the assembly fails.

Example

```
            ; code
routineA    ROUT            ; ROUT is not necessarily a routine
            ; code
3routineA   ; code          ; this label is checked
            ; code
            BEQ     %4routineA   ; this reference is checked
            ; code
            BGE     %3      ; refers to 3 above, but not checked
            ; code
4routineA   ; code          ; this label is checked
            ; code
otherstuff  ROUT            ; start of next scope
```

Related concepts

12.10 Numeric local labels on page 12-307.

Related references

21.6 AREA on page 21-1650.

21.63 SETA, SETL, and SETS

The SETA, SETL, and SETS directives set the value of a local or global variable.

Syntax

variable setx expr

where:

variable
is the name of a variable declared by a GBLA, GBLL, GBLS, LCLA, LCLL, or LCLS directive.

setx
is one of SETA, SETL, or SETS.

expr
is an expression that is:

- Numeric, for SETA.
- Logical, for SETL.
- String, for SETS.

Usage

The SETA directive sets the value of a local or global arithmetic variable.

The SETL directive sets the value of a local or global logical variable.

The SETS directive sets the value of a local or global string variable.

You must declare *variable* using a global or local declaration directive before using one of these directives.

You can also predefine variable names on the command line.

Restrictions

The value you can specify using a SETA directive is limited to 32 bits. If you exceed this limit, the assembler reports an error. A possible workaround in A64 code is to use an EQU directive instead of SETA, although EQU defines a constant, whereas GBLA and SETA define a variable.

For example, replace the following code:

```
                    GBLA    MyAddress
     MyAddress      SETA    0x0000008000000000
```

with:

```
     MyAddress      EQU     0x0000008000000000
```

Examples

```
                GBLA    VersionNumber
VersionNumber   SETA    21
                GBLL    Debug
Debug           SETL    {TRUE}
                GBLS    VersionString
VersionString   SETS    "Version 1.0"
```

Related concepts

12.12 String expressions on page 12-309.
12.14 Numeric expressions on page 12-311.
12.17 Logical expressions on page 12-314.

Related references

21.42 GBLA, GBLL, and GBLS on page 21-1689.

21.49 LCLA, LCLL, and LCLS on page 21-1698.
11.54 --predefine "directive" on page 11-282.

21.64 SPACE or FILL

The SPACE directive reserves a zeroed block of memory. The FILL directive reserves a block of memory to fill with a given value.

Syntax

{*label*} SPACE *expr*

{*label*} FILL *expr*{,*value*{,*valuesize*}}

where:

label
: is an optional label.

expr
: evaluates to the number of bytes to fill or zero.

value
: evaluates to the value to fill the reserved bytes with. *value* is optional and if omitted, it is 0. *value* must be 0 in a NOINIT area.

valuesize
: is the size, in bytes, of *value*. It can be any of 1, 2, or 4. *valuesize* is optional and if omitted, it is 1.

Usage

Use the ALIGN directive to align any code following a SPACE or FILL directive.

% is a synonym for SPACE.

Example

```
        AREA    MyData, DATA, READWRITE
data1   SPACE   255       ; defines 255 bytes of zeroed store
data2   FILL    50,0xAB,1 ; defines 50 bytes containing 0xAB
```

Related concepts

12.14 Numeric expressions on page 12-311.

Related references

21.5 ALIGN on page 21-1648.
21.15 DCB on page 21-1661.
21.16 DCD and DCDU on page 21-1662.
21.21 DCQ and DCQU on page 21-1667.
21.22 DCW and DCWU on page 21-1668.

21.65 THUMB directive

The THUMB directive instructs the assembler to interpret subsequent instructions as T32 instructions, using the UAL syntax.

Note

Not supported for AArch64 state.

Syntax

```
THUMB
```

Usage

In files that contain code using different instruction sets, the THUMB directive must precede T32 code written in UAL syntax.

If necessary, this directive also inserts one byte of padding to align to the next halfword boundary.

This directive does not assemble to any instructions. It also does not change the state. It only instructs armasm to assemble T32 instructions as appropriate, and inserts padding if necessary.

Example

This example shows how you can use ARM and THUMB directives to switch state and assemble both A32 and T32 instructions in a single area.

```
        AREA ToT32, CODE, READONLY      ; Name this block of code
        ENTRY                           ; Mark first instruction to execute
        ARM                             ; Subsequent instructions are A32
start
        ADR     r0, into_t32 + 1        ; Processor starts in A32 state
        BX      r0                      ; Inline switch to T32 state
        THUMB                           ; Subsequent instructions are T32
into_t32
        MOVS    r0, #10                 ; New-style T32 instructions
```

Related references

21.7 ARM or CODE32 directive on page 21-1653.
21.11 CODE16 directive on page 21-1657.

21.66 TTL and SUBT

The TTL directive inserts a title at the start of each page of a listing file. The SUBT directive places a subtitle on the pages of a listing file.

Syntax

```
TTL title

SUBT subtitle
```

where:

title
: is the title.

subtitle
: is the subtitle.

Usage

Use the TTL directive to place a title at the top of each page of a listing file. If you want the title to appear on the first page, the TTL directive must be on the first line of the source file.

Use additional TTL directives to change the title. Each new TTL directive takes effect from the top of the next page.

Use SUBT to place a subtitle at the top of each page of a listing file. Subtitles appear in the line below the titles. If you want the subtitle to appear on the first page, the SUBT directive must be on the first line of the source file.

Use additional SUBT directives to change subtitles. Each new SUBT directive takes effect from the top of the next page.

Examples

```
TTL   First Title    ; places title on first and subsequent pages of listing file.
SUBT  First Subtitle ; places subtitle on second and subsequent pages of listing file.
```

21.67 WHILE and WEND

The `WHILE` directive starts a sequence of instructions or directives that are to be assembled repeatedly. The sequence is terminated with a `WEND` directive.

Syntax

```
WHILE logical-expression
  code
WEND
```

where:

logical-expression
is an expression that can evaluate to either `{TRUE}` or `{FALSE}`.

Usage

Use the `WHILE` directive, together with the `WEND` directive, to assemble a sequence of instructions a number of times. The number of repetitions can be zero.

You can use `IF...ENDIF` conditions within `WHILE...WEND` loops.

`WHILE...WEND` loops can be nested.

Example

```
        GBLA count                  ; declare local variable
count   SETA    1                   ; you are not restricted to
        WHILE   count <= 4          ; such simple conditions
count   SETA    count+1             ; In this case, this code is
            ; code                  ; executed four times
            ; code                  ;
        WEND
```

Related concepts

12.17 Logical expressions on page 12-314.

Related references

21.2 About assembly control directives on page 21-1645.

21.68 WN and XN

The WN, and XN directives define names for registers in A64 code.

The WN directive defines a name for a specified 32-bit register.

The XN directive defines a name for a specified 64-bit register.

Syntax

name directive expr

where:

name
: is the name to be assigned to the register. *name* cannot be the same as any of the predefined names.

directive
: is WN or XN.

expr
: evaluates to a register number from 0 to 30.

Usage

Use WN and XN to allocate convenient names to registers in A64 code, to help you to remember what you use each register for. Be careful to avoid conflicting uses of the same register under different names.

Examples

```
sqr4        WN w16  ; defines sqr4 for register w16
regname     XN 21   ; defines regname for register x21
```

Related references

4.5 Predeclared core register names in AArch64 state on page 4-85.
4.6 Predeclared extension register names in AArch64 state on page 4-86.

Chapter 22
Via File Syntax

Describes the syntax of via files accepted by `armasm`.

It contains the following sections:

22.1 Overview of via files

Via files are plain text files that allow you to specify assembler command-line arguments and options.

Typically, you use a via file to overcome the command-line length limitations. However, you might want to create multiple via files that:

- Group similar arguments and options together.
- Contain different sets of arguments and options to be used in different scenarios.

Note

In general, you can use a via file to specify any command-line option to a tool, including `--via`. This means that you can call multiple nested via files from within a via file.

Via file evaluation

When the assembler is invoked it:

1. Replaces the first specified `--via` *`via_file`* argument with the sequence of argument words extracted from the via file, including recursively processing any nested `--via` commands in the via file.
2. Processes any subsequent `--via` *`via_file`* arguments in the same way, in the order they are presented.

That is, via files are processed in the order you specify them, and each via file is processed completely including processing nested via files before processing the next via file.

Related references

22.2 Via file syntax rules on page 22-1725.
11.64 --via=filename on page 11-292.

22.2 Via file syntax rules

Via files must conform to some syntax rules.

- A via file is a text file containing a sequence of words. Each word in the text file is converted into an argument string and passed to the tool.
- Words are separated by whitespace, or the end of a line, except in delimited strings, for example:

  `--bigend --reduce_paths` (two words)

  `--bigend--reduce_paths` (one word)
- The end of a line is treated as whitespace, for example:

  ```
  --bigend
  --reduce_paths
  ```

  This is equivalent to:

  `--bigend --reduce_paths`
- Strings enclosed in quotation marks (`"`), or apostrophes (`'`) are treated as a single word. Within a quoted word, an apostrophe is treated as an ordinary character. Within an apostrophe delimited word, a quotation mark is treated as an ordinary character.

  Use quotation marks to delimit filenames or path names that contain spaces, for example:

  `--errors C:\My Project\errors.txt` (three words)

  `--errors "C:\My Project\errors.txt"` (two words)

  Use apostrophes to delimit words that contain quotes, for example:

  `-DNAME='"ARM Compiler"'` (one word)
- Characters enclosed in parentheses are treated as a single word, for example:

  `--option(x, y, z)` (one word)

  `--option (x, y, z)` (two words)
- Within quoted or apostrophe delimited strings, you can use a backslash (\) character to escape the quote, apostrophe, and backslash characters.
- A word that occurs immediately next to a delimited word is treated as a single word, for example:

  `--errors"C:\Project\errors.txt"`

  This is treated as the single word:

  `--errorsC:\Project\errors.txt`
- Lines beginning with a semicolon (`;`) or a hash (`#`) character as the first nonwhitespace character are comment lines. A semicolon or hash character that appears anywhere else in a line is not treated as the start of a comment, for example:

  ```
  -o objectname.axf      ;this is not a comment
  ```

  A comment ends at the end of a line, or at the end of the file. There are no multi-line comments, and there are no part-line comments.

Related concepts

22.1 Overview of via files on page 22-1724.

Related references

11.64 --via=filename on page 11-292.