# ARM® Compiler v5.06 for µVision®

**Version 5**

## armasm User Guide


ARM DUI0379G

# ARM® Compiler v5.06 for µVision®

## armasm User Guide



### Release Information

### Document History

| Issue | Date | Confidentiality | Change |
|---|---|---|---|
| A | May 2007 | Non-Confidential | Release for RVCT v3.1 for µVision |
| B | December 2008 | Non-Confidential | Release for RVCT v4.0 for µVision |
| C | June 2011 | Non-Confidential | Release for ARM Compiler v4.1 for µVision |
| D | July 2012 | Non-Confidential | Release for ARM Compiler v5.02 for µVision |
| E | 30 May 2014 | Non-Confidential | Release for ARM Compiler v5.04 for µVision |
| F | 12 December 2014 | Non-Confidential | Release for ARM Compiler v5.05 for µVision |
| G | 15 August 2015 | Non-Confidential | Release for ARM Compiler v5.06 for µVision |

### Non-Confidential Proprietary Notice

ARM Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

LES-PRE-20349

### Confidentiality Status

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

Unrestricted Access is an ARM internal classification.

### Product Status

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

### Web Address

*http://www.arm.com*

# Contents
# ARM® Compiler v5.06 for µVision® armasm User Guide

# List of Figures
# ARM® Compiler v5.06 for µVision® armasm User Guide

# List of Tables
# ARM® Compiler v5.06 for µVision® armasm User Guide

# Preface

This preface introduces the *ARM® Compiler v5.06 for µVision® armasm User Guide*.

It contains the following:

# About this book

ARM® Compiler for µVision® armasm User Guide. This document provides topic-based documentation for the ARM assembler (armasm). It contains information on command-line options, instruction sets, and assembler directives, and is available as PDF.

## 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 Architecture***
Gives an overview of the ARM architecture.

***Chapter 3 Structure of Assembly Language Modules***
Describes the structure of assembly language source files.

***Chapter 4 Writing ARM Assembly Language***
Describes the use of a few basic assembly language instructions and the use of macros.

***Chapter 5 Condition Codes***
Describes condition codes and the conditional execution of ARM and Thumb code.

***Chapter 6 Using the Assembler***
Describes how to use the ARM assembler, `armasm`.

***Chapter 7 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 8 VFP Programming***
Describes the assembly programming of VFP hardware.

***Chapter 9 Assembler Command-line Options***
Describes the command-line options supported by the ARM assembler, `armasm`.

***Chapter 10 ARM and Thumb Instructions***
Describes the ARM and Thumb instructions supported by the ARM assembler, `armasm`.

***Chapter 11 VFP Instructions***
Describes the assembly programming of the VFP hardware.

***Chapter 12 Directives Reference***
Describes the directives that are provided by the ARM assembler, `armasm`.

***Chapter 13 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 v5.06 for µVision® armasm User Guide*.
- The number ARM DUI0379G.
- 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 Information Center*.
- *ARM Technical Support Knowledge Articles*.
- *Support and Maintenance*.
- *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:

- A freestanding assembler, `armasm`.
- An optimizing inline assembler and a non-optimizing embedded assembler built into the C and C++ compilers. These use the same syntax for assembly instructions.

**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 unsupported features is operating correctly.

### Related information

*Using the Inline and Embedded Assemblers of the ARM Compiler.*

## 1.2 Key features of the assembler

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

It supports:

- *Unified Assembly Language* (UAL) for both ARM and Thumb® code.
- *Vector Floating Point* (VFP) instructions in ARM and Thumb code.
- Directives in assembly source code.
- Processing of user-defined macros.

### Related concepts

*1.3 How the assembler works* on page 1-25.

*4.1 About the Unified Assembler Language* on page 4-58.

*4.21 Use of macros* on page 4-85.

### Related references

*Chapter 12 Directives Reference* on page 12-574.

## 1.3 How the assembler works

The ARM assembler 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

*6.13 Two pass assembler diagnostics* on page 6-124.
*4.24 Instruction and directive relocations* on page 4-89.

### Related references

*1.4 Directives that can be omitted in pass 2 of the assembler* on page 1-27.

*9.18 --diag_error=tag[,tag,...]* on page 9-214.

*9.15 --debug* on page 9-211.

## 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`.
- `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 1-25.
*6.13 Two pass assembler diagnostics* on page 6-124.

# Chapter 2
# Overview of the ARM Architecture

Gives an overview of the ARM architecture.

It contains the following sections:

## 2.1 About the ARM architecture

The ARM architecture is a load-store architecture, with a 32-bit addressing range.

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.

It is assumed that you are using a processor that implements the ARMv4 or later architecture. All these processors have a 32-bit addressing range.

### Related information

*ARM Architecture Reference Manual.*

## 2.2 ARM, Thumb, and ThumbEE instruction sets

ARM instructions are 32 bits wide. Thumb instructions are 16 or 32-bits wide.

The ARM instruction set is a set of 32-bit instructions providing a comprehensive range of operations.

ARMv4T and later define a 16-bit instruction set called Thumb. Most of the functionality of the 32-bit ARM instruction set is available, but some operations require more instructions. The Thumb instruction set provides better code density, at the expense of performance.

ARMv6T2 introduces Thumb-2 technology. This is a major enhancement to the Thumb instruction set by providing 32-bit Thumb instructions. The 32-bit and 16-bit Thumb instructions together provide almost exactly the same functionality as the ARM instruction set. This version of the Thumb instruction set achieves the high performance of ARM code along with the benefits of better code density.

ARMv7 includes Thumb-2 technology. ARMv7-M only supports the Thumb instruction set. Therefore, interworking instructions in ARMv7-M must not attempt to change to ARM state. ARMv7-R supports both ARM and Thumb instruction sets.

ARMv7 defines the *Thumb Execution Environment* (ThumbEE). The ThumbEE instruction set is based on Thumb, with some changes and additions to make it a better target for dynamically generated code, that is, code compiled on the device either shortly before or during execution.

——— **Note** ———

ARM deprecates the use of ThumbEE instructions.

———————————

### Related references

*2.18 ARM and Thumb instruction set overview* on page 2-48.

## 2.3 Changing between ARM, Thumb, and ThumbEE state

The processor must be in the correct *instruction set state* for the instructions it is executing.

A processor that is executing ARM instructions is operating in *ARM state*. A processor that is executing Thumb instructions is operating in *Thumb state*. A processor that is executing ThumbEE instructions is operating in *ThumbEE state*. A processor can also operate in another state called the *Jazelle® state*. These are called *instruction set states*. The assembler cannot directly assemble code for the Jazelle state.

A processor in one instruction set state cannot execute instructions from another instruction set. For example, a processor in ARM state cannot execute Thumb instructions, and a processor in Thumb state cannot execute ARM 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 the assembler to generate ARM or Thumb instruction encodings, you must set the assembler mode using an `ARM` or `THUMB` directive. To generate ThumbEE code, use the `THUMBX` directive. Assembly code using `CODE32` and `CODE16` directives can still be assembled, but ARM recommends using `ARM` and `THUMB` 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 ARM and Thumb states when performing a branch.

### Related references

*10.22 BLX* on page 10-314.
*10.23 BX* on page 10-316.
*12.7 ARM or CODE32* on page 12-585.
*12.65 THUMB* on page 12-649.
*12.66 THUMBX* on page 12-650.

## 2.4 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 and ARMv7-M do not support the same modes as other ARM architectures and profiles. The processor modes listed here do not apply to ARMv6-M and ARMv7-M.

**Table 2-1 ARM processor modes**

| Processor mode | Architectures | Mode number |
|---|---|---|
| User | All | 0b10000 |
| FIQ | All | 0b10001 |
| IRQ | All | 0b10010 |
| Supervisor | All | 0b10011 |
| Monitor | Security Extensions only | 0b10110 |
| Abort | All | 0b10111 |
| Hyp | Virtualization Extensions only | 0b11010 |
| Undefined | All | 0b11011 |
| System | ARMv4 and later | 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.

On an implementation that includes the Security Extensions, in all modes except Monitor mode and Hyp mode, code can run in either a secure state or in a non-secure state. In Monitor mode, code can only run in a secure state, and in Hyp mode, code can only run in a non-secure state.

### Related concepts

*2.5 Processor modes in ARMv6-M and ARMv7-M* on page 2-34.

### Related information

*ARM Architecture Reference Manual.*

## 2.5 Processor modes in ARMv6-M and ARMv7-M

The processor modes available in ARMv6-M and ARMv7-M 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

*2.4 Processor modes, and privileged and unprivileged software execution* on page 2-33.

### Related information

*ARM Architecture Reference Manual.*

## 2.6 VFP hardware

There are several VFP architecture versions and variants.

The VFP hardware, together with associated support code, provides single-precision and double-precision floating-point arithmetic, as defined by *ANSI/IEEE Std. 754-1985 IEEE Standard for Binary Floating-Point Arithmetic*. This document is referred to as the IEEE 754 standard.

The VFP hardware uses a register bank that is distinct from the ARM core register bank.

There are several versions of the VFP architecture, including VFPv2, VFPv3, VFPv3 with half precision extensions, and VFPv4. In addition, FPv5 is available for ARM Cortex®-M7 processors. There are variants of VFPv3 and VFPv4 that differ in the number of accessible registers or in their support for trapping floating-point exceptions.

### Related concepts

*8.1 Architecture support for VFP* on page 8-163.

*8.2 Half-precision extension for VFP* on page 8-164.

*8.5 VFP views of the extension register bank* on page 8-168.

### Related references

*Chapter 8 VFP Programming* on page 8-161.

### Related information

*Vector Floating-Point (VFP) architectures.*

## 2.7 ARM registers

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

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

- 13 general-purpose registers R0-R12.
- One *Stack Pointer* (SP).
- One *Link Register* (LR).
- One *Program Counter* (PC).
- One *Application Program Status Register* (APSR).

**Note**

The Link Register can also be used as a general-purpose register. The Stack Pointer can be used as a general-purpose register in ARM state only.

Additional registers are available in privileged software execution.

ARM processors, with the exception of ARMv6-M and ARMv7-M based processors, have a total of 37 registers, with 3 additional registers if the Security Extensions are implemented, and in ARMv7-A only, 3 more if the Virtualization Extensions are implemented. 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 that are available in privileged software execution, with the exception of ARMv6-M and ARMv7-M, are:

- Two Supervisor mode registers for banked SP and LR.
- Two Abort mode registers for banked SP and LR.
- Two Undefined mode registers for banked SP and LR.
- Two Interrupt mode registers for banked SP and LR.
- Seven FIQ mode registers for banked R8-R12, SP and LR.
- Two Monitor mode registers for banked SP and LR. These are only present if the Security Extensions are implemented.
- Two Hyp mode registers for banked SP, and to hold the return address from Hyp mode. These are only present if the Virtualization Extensions are implemented.
- One *Saved Program Status Register* (SPSR) for each exception 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 except ARMv6-M and ARMv7-M:

Application level view

System level views

Privileged modes

Exception modes

| | User mode | System mode | Hyp mode † | Supervisor mode | Monitor mode ‡ | Abort mode | Undefined mode | IRQ mode | FIQ mode |
|---|---|---|---|---|---|---|---|---|---|
| R0 | R0_usr | | | | | | | | |
| R1 | R1_usr | | | | | | | | |
| R2 | R2_usr | | | | | | | | |
| R3 | R3_usr | | | | | | | | |
| R4 | R4_usr | | | | | | | | |
| R5 | R5_usr | | | | | | | | |
| R6 | R6_usr | | | | | | | | |
| R7 | R7_usr | | | | | | | | |
| R8 | R8_usr | | | | | | | | R8_fiq |
| R9 | R9_usr | | | | | | | | R9_fiq |
| R10 | R10_usr | | | | | | | | R10_fiq |
| R11 | R11_usr | | | | | | | | R11_fiq |
| R12 | R12_usr | | | | | | | | R12_fiq |
| SP | SP_usr | | SP_hyp† | SP_svc | SP_mon ‡ | SP_abt | SP_und | SP_irq | SP_fiq |
| LR | LR_usr | | | LR_svc | LR_mon ‡ | LR_abt | LR_und | LR_irq | LR_fiq |
| PC | PC | | | | | | | | |
| APSR | CPSR | | | | | | | | |
| | | | SPSR_hyp† | SPSR_svc | SPSR_mon ‡ | SPSR_abt | SPSR_und | SPSR_irq | SPSR_fiq |
| | | | ELR_hyp† | | | | | | |

† Hyp mode and the associated banked registers are implemented only as part of the Virtualization Extensions

‡ Monitor mode and the associated banked registers are implemented only as part of the Security Extensions

**Figure 2-1 Organization of general-purpose registers and Program Status Registers**

In ARMv6-M and ARMv7-M based processors, SP is an alias for the two banked stack pointer registers:

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

### Related concepts

*2.8 General-purpose registers* on page 2-38.

*2.13 Program Counter* on page 2-43.

*2.14 Application Program Status Register* on page 2-44.

*2.17 Saved Program Status Registers* on page 2-47.

*2.16 Current Program Status Register* on page 2-46.

*2.4 Processor modes, and privileged and unprivileged software execution* on page 2-33.

### Related information

*ARM Architecture Reference Manual.*

## 2.8 General-purpose registers

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

With the exception of ARMv6-M and ARMv7-M based processors, there are 30 (or 32 if Security Extensions are implemented) general-purpose 32-bit registers, that include 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, 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. Use of SP as a general purpose register is discouraged. In Thumb, SP is strictly defined as the stack pointer. The instruction descriptions mention 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.

**Note**

When using the `--use_frame_pointer` option with `armcc`, do not use `R11` as a general-purpose register.

### Related concepts

*2.13 Program Counter* on page 2-43.
*2.9 Register accesses* on page 2-39.

### Related references

*2.10 Predeclared core register names* on page 2-40.
*10.60 MRS (PSR to general-purpose register)* on page 10-376.
*10.63 MSR (general-purpose register to PSR)* on page 10-380.

### Related information

*--use_frame_pointer (compiler option).*

## 2.9 Register accesses

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

Most 16-bit Thumb instructions can only access R0 to R7. Only a small number of these 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 Thumb instructions can access R0 to R12, and LR. However, apart from a few designated stack manipulation instructions, most Thumb instructions cannot use SP. Except for a few specific instructions where PC is useful, most Thumb instructions cannot use PC.

In ARM 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 ARM instruction, in any way that is not possible in the corresponding Thumb instruction, is deprecated. Explicit use of the PC in an ARM 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

*2.8 General-purpose registers* on page 2-38.
*2.13 Program Counter* on page 2-43.
*2.14 Application Program Status Register* on page 2-44.
*2.16 Current Program Status Register* on page 2-46.
*2.17 Saved Program Status Registers* on page 2-47.
*4.19 The Read-Modify-Write operation* on page 4-83.

### Related references

*2.10 Predeclared core register names* on page 2-40.
*10.60 MRS (PSR to general-purpose register)* on page 10-376.
*10.63 MSR (general-purpose register to PSR)* on page 10-380.

## 2.10 Predeclared core register names

Many of the core register names have synonyms.

The following table shows the predeclared core registers:

**Table 2-2 Predeclared core registers**

| 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` and `SB` | Static base register. This is a synonym for `R9`. |
| `ip` and `IP` | Intra procedure call scratch register. This is a synonym for `R12`. |
| `sp` and `SP` | Stack pointer. This is a synonym for `R13`. |
| `lr` and `LR` | Link register. This is a synonym for `R14`. |
| `pc` and `PC` | Program counter. This is a synonym for `R15`. |

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

### Related concepts

*2.8 General-purpose registers* on page 2-38.

## 2.11 Predeclared extension register names

You can write the names of VFP registers either in upper case or lower case.

The following extension register names are predeclared:

**Table 2-3 Predeclared extension registers**

| Register names | Meaning |
|---|---|
| `d0-d31` and `D0-D31` | VFP double-precision registers.<br>**Note**<br>Some versions of VFP have sixteen double-precision registers, `D0-D15`. |
| `s0-s31` and `S0-S31` | VFP single-precision registers |

### Related concepts

*8.4 Extension register bank mapping in VFP* on page 8-166.

## 2.12 Predeclared coprocessor names

There are ranges of predeclared coprocessor names and coprocessor register names. All names are case-sensitive.

**Table 2-4 Predeclared coprocessor registers**

| Register name | Meaning |
|---|---|
| `p0-p15` | Coprocessors 0-15 |
| `c0-c15` | Coprocessor registers 0-15 |

### Related references

*10.26 CDP and CDP2* on page 10-320.
*10.51 MCR and MCR2* on page 10-366.
*10.58 MRC and MRC2* on page 10-374.

## 2.13 Program Counter

You can use the Program Counter explicitly, for example in some ARM 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 ARM state). Branch instructions load the destination address into PC. You can also load the PC directly using data processing instructions. For example, to branch to the address in a general purpose register, use:

```
MOV PC,R0
```

During execution, PC does not contain the address of the currently executing instruction. The address of the currently executing instruction is typically PC–8 for ARM, or PC–4 for Thumb.

**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

*7.5 Register-relative and PC-relative expressions* on page 7-136.

### Related references

*10.16 B* on page 10-305.
*10.21 BL* on page 10-312.
*10.22 BLX* on page 10-314.
*10.23 BX* on page 10-316.
*10.24 BXJ* on page 10-318.
*10.25 CBZ and CBNZ* on page 10-319.
*10.148 TBB and TBH* on page 10-493.

## 2.14 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.

On ARMv5TE, ARMv6 and later architectures, the APSR also holds the Q (saturation) flag.

On ARMv6 and later, 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

*5.1 Conditional instructions* on page 5-96.
*5.4 Updates to the condition flags* on page 5-99.

### Related references

*10.60 MRS (PSR to general-purpose register)* on page 10-376.
*10.63 MSR (general-purpose register to PSR)* on page 10-380.
*10.99 SEL* on page 10-430.

## 2.15 The Q flag

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

In ARMv5TE, ARMv6 and later, the Q flag is set to 1 when saturation has occurred in saturating arithmetic instructions, or when overflow has occurred 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

*4.19 The Read-Modify-Write operation* on page 4-83.

### Related references

*10.60 MRS (PSR to general-purpose register)* on page 10-376.
*10.63 MSR (general-purpose register to PSR)* on page 10-380.
*10.74 QADD* on page 10-398.
*10.121 SMULxy* on page 10-454.
*10.123 SMULWy* on page 10-456.

## 2.16 Current Program Status Register

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

The CPSR holds:

- The APSR flags.
- The processor mode.
- The interrupt disable flags.
- The instruction set state (ARM, Thumb, ThumbEE, or Jazelle).
- The endianness state (on ARMv4T and later).
- The execution state bits for the IT block (on ARMv6T2 and later).

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. `SETEND` is the preferred instruction to write to the E bit.

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

**Note**

The CSPR is not present in ARMv6-M and ARMv7-M processors.

### Related concepts

*5.4 Updates to the condition flags* on page 5-99.
*2.17 Saved Program Status Registers* on page 2-47.

### Related references

*10.39 IT* on page 10-338.
*10.60 MRS (PSR to general-purpose register)* on page 10-376.
*10.63 MSR (general-purpose register to PSR)* on page 10-380.
*10.100 SETEND* on page 10-432.

## 2.17 Saved Program Status Registers

A *Saved Program Status Register* (SPSR) stores the current value of the CPSR when an exception is taken so that the CPSR 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` instruction. You cannot access the SPSR using `MSR` or `MRS` in User or System mode.

### Related concepts

*2.16 Current Program Status Register* on page 2-46.

## 2.18 ARM and Thumb instruction set overview

ARM and Thumb instructions can be grouped by functional area.

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

Thumb 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 Thumb code or ARM code.

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

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

**Table 2-5 Instruction groups**

| Instruction Group | Description |
|---|---|
| Branch and control | These instructions do the following:<br>• Branch to subroutines.<br>• Branch backwards to form loops.<br>• Branch forward in conditional structures.<br>• Make following instructions conditional without branching.<br>• Change the processor between ARM state and Thumb 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.<br>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.<br>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. |
| Coprocessor | These instructions support a general way to extend the ARM architecture. They also enable the control of the CP15 System Control coprocessor registers. |

### Related concepts

*4.13 Load and store multiple register instructions* on page 4-75.

### Related references

*Chapter 10 ARM and Thumb Instructions* on page 10-270.

## 2.19 Access to the inline barrel shifter

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

The second operand to many ARM and Thumb 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 Thumb instructions give almost the same access to the barrel shifter as ARM instructions.

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

### Related concepts

*4.3 Load immediate values* on page 4-60.

*4.4 Load immediate values using MOV and MVN* on page 4-61.

# Chapter 3
# Structure of Assembly Language Modules

Describes the structure of assembly language source files.

It contains the following sections:

## 3.1 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`, `v1-v8`, and Wireless MMX registers) 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     ARMex, CODE, READONLY
                                ; Name this block of code ARMex
        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      ; ARM 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

*7.6 Labels* on page 7-137.
*7.10 Numeric local labels* on page 7-141.
*7.13 String literals* on page 7-144.

### Related references

*3.2 Literals* on page 3-53.
*7.1 Symbol naming rules* on page 7-132.
*7.15 Syntax of numeric literals* on page 7-146.

## 3.2 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

*3.1 Syntax of source lines in assembly language* on page 3-51.

## 3.3 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 `ARMex` that contains code:

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

### Related concepts

*3.4 An example ARM assembly language module* on page 3-55.

### Related references

*12.6 AREA* on page 12-582.

### Related information

*Information about scatter files.*

## 3.4 An example ARM assembly language module

An ARM 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 assembly language module

The following example defines a single section called `ARMex` that contains code and is marked as being `READONLY`.

```
        AREA     ARMex, CODE, READONLY
                                ; Name this block of code ARMex
        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      ; ARM semihosting (formerly SWI)
        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

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 termination

After executing the main code, the application terminates by returning control to the debugger. You do this using the ARM semihosting SVC (`0x123456` by default), with the following parameters:

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

### 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

*3.3 ELF sections and the AREA directive* on page 3-54.

### Related references

*12.24 END* on page 12-602.
*12.26 ENTRY* on page 12-604.

### Related information

*What is semihosting?*.

# Chapter 4
# Writing ARM Assembly Language

Describes the use of a few basic assembly language instructions and the use of macros.

It contains the following sections:

## 4.1 About the Unified Assembler Language

Unified Assembler Language (UAL) is a common syntax for ARM and Thumb instructions.

UAL supersedes earlier versions of both the ARM and Thumb assembler languages.

Code written using UAL can be assembled for ARM or Thumb for any ARM processor. The assembler faults the use of unavailable instructions.

*RealView® Compilation Tools* (RVCT) v2.1 and earlier can only assemble the pre-UAL syntax. Later versions of RVCT and ARM Compiler toolchain can assemble code written in pre-UAL and UAL syntax.

You can use directives or command-line options to instruct the assembler whether you are using UAL or pre-UAL syntax. By default, the assembler expects source code to be written in UAL. If you use any of the `CODE32`, `ARM`, `THUMB`, or `THUMBX` directives, or if you assemble with any of the `--32`, `--arm`, `--thumb`, or `--thumbx` command-line options, the assembler accepts UAL syntax. The assembler also accepts source code written in pre-UAL ARM assembly language when you use the `CODE32` or `ARM` directives.

The assembler accepts source code written in pre-UAL Thumb assembly language when you assemble using the `--16` command-line option, or you use the `CODE16` directive in the source code.

**Note**

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

### Related references

*9.1 --16* on page 9-195.
*12.7 ARM or CODE32* on page 12-585.
*9.2 --32* on page 9-196.
*9.4 --arm* on page 9-199.
*9.63 --thumb* on page 9-260.
*9.64 --thumbx* on page 9-261.

## 4.2 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                   ; ARM semihosting (formerly SWI)
doadd   ADD     r0, r0, r1                  ; Subroutine code
        BX      lr                          ; Return from subroutine
        END                                 ; Mark end of file
```

### Related concepts

*4.16 Stack operations for nested subroutines* on page 4-79.

### Related references

*10.16 B* on page 10-305.

### Related information

*Procedure Call Standard for the ARM Architecture.*

## 4.3 Load immediate values

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

ARM and Thumb 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.

In ARMv6T2 and later, 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 Thumb instructions is much smaller.

### Related concepts

*4.4 Load immediate values using MOV and MVN* on page 4-61.

*4.5 Load immediate values using MOV32* on page 4-64.

*4.6 Load immediate values using LDR Rd, =const* on page 4-65.

*8.6 Load values to VFP registers* on page 8-169.

### Related references

*10.46 LDR pseudo-instruction* on page 10-356.

## 4.4 Load immediate values using MOV and MVN

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

In ARM state:

- 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.
- In ARMv6T2 and later, 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 ARM 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 4-1 ARM 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 ARM instruction in ARMv6T2 and later:

**Table 4-2 ARM 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 ARM state only. All the other values in this table are also available in 32-bit Thumb instructions.

**c**

These values are only available in ARMv6T2 and later. They are not available directly as operands in other instructions.

---

In Thumb state in ARMv6T2 and later:

- 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 Thumb, the 16-bit Thumb `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 Thumb `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 4-3 32-bit Thumb immediate values**

| Binary | Decimal | Step | Hexadecimal | MVN value[a] | Notes |
|---|---|---|---|---|---|
| `00000000000000000000000abcdefgh` | 0-255 | 1 | `0x0-0xFF` | –1 to –256 | - |
| `00000000000000000000000abcdefgh0` | 0-510 | 2 | `0x0-0x1FE` | –2 to –512 | - |
| `00000000000000000000000abcdefgh00` | 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` | - |

**Table 4-3 32-bit Thumb immediate values (continued)**

| Binary | Decimal | Step | Hexadecimal | MVN value[a] | Notes |
|---|---|---|---|---|---|
| abcdefgh00000000abcdefgh00000000 | (bit pattern) | - | 0xXY00XY00 | 0xXYFFXYFF | - |
| 00000000000000000000abcdefghijkl | 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 Thumb instruction:

**Table 4-4 32-bit Thumb 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 ARM and Thumb, 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

*4.3 Load immediate values* on page 4-60.

## 4.5 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.

In ARMv6T2 and later, both ARM and Thumb 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

*7.5 Register-relative and PC-relative expressions* on page 7-136.

### Related references

*10.56 MOV32 pseudo-instruction* on page 10-372.

## 4.6 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

*4.7 Literal pools* on page 4-66.

### Related references

*10.46 LDR pseudo-instruction* on page 10-356.

## 4.7 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 ARM or Thumb 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 Thumb `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 ARM 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          ; ARM 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

*4.6 Load immediate values using LDR Rd, =const* on page 4-65.

### Related references

*12.51 LTORG* on page 12-632.

## 4.8 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

*4.9 Load addresses to a register using ADR* on page 4-69.
*4.10 Load addresses to a register using ADRL* on page 4-71.
*4.5 Load immediate values using MOV32* on page 4-64.
*4.11 Load addresses to a register using LDR Rd, =label* on page 4-72.

## 4.9 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 ARM 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 ARM 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            ; ARM 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

*4.11 Load addresses to a register using LDR Rd, =label* on page 4-72.
*4.10 Load addresses to a register using ADRL* on page 4-71.

### Related references

*10.11 ADR (PC-relative)* on page 10-295.

## 4.10 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.

`ADRL` is not available in Thumb state on processors before ARMv6T2.

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

*4.9 Load addresses to a register using ADR* on page 4-69.
*4.11 Load addresses to a register using LDR Rd, =label* on page 4-72.

## 4.11 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 ARM 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            ; ARM 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 ARM 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           ; ARM 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

*4.10 Load addresses to a register using ADRL* on page 4-71.

*4.6 Load immediate values using LDR Rd, =const* on page 4-65.

### Related references

*10.46 LDR pseudo-instruction* on page 10-356.

*12.15 DCB* on page 12-593.

## 4.12 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

*4.13 Load and store multiple register instructions* on page 4-75.

### Related references

*4.14 Load and store multiple register instructions in ARM and Thumb* on page 4-76.

*10.55 MOV* on page 10-370.

## 4.13 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

*4.15 Stack implementation using LDM and STM* on page 4-77.

*4.16 Stack operations for nested subroutines* on page 4-79.

*4.17 Block copy with LDM and STM* on page 4-80.

### Related references

*4.14 Load and store multiple register instructions in ARM and Thumb* on page 4-76.

## 4.14 Load and store multiple register instructions in ARM and Thumb

Instructions are available in both the ARM and Thumb 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 ARM instructions is deprecated.

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

### Related concepts

*4.13 Load and store multiple register instructions* on page 4-75.

## 4.15 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 4-5 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 4-6 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 `armcc` 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

*4.13 Load and store multiple register instructions* on page 4-75.

### Related references

*10.41 LDM* on page 10-342.

### Related information

*Procedure Call Standard for the ARM Architecture.*

## 4.16 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
```

**Note**

Use this with care in mixed ARM and Thumb systems. In ARMv4T systems, you cannot change state by popping directly into PC. In these cases you must pop the address into a temporary register and use the `BX` instruction.

In ARMv5T and later, you can change state in this way.

### Related concepts

*4.2 Register usage in subroutine calls* on page 4-59.

*4.13 Load and store multiple register instructions* on page 4-75.

### Related information

*Procedure Call Standard for the ARM Architecture.*

## 4.17 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 ARM 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            ; ARM 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             ; ARM 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?.*

## 4.18 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

*6.16 Address alignment* on page 6-128.

### Related references

*2.7 ARM registers* on page 2-36.

## 4.19 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.

The following example shows how to use the read-modify-write procedure to change some bits in the VFP system register FPSCR, without affecting the other bits:

```
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

*2.9 Register accesses* on page 2-39.
*2.15 The Q flag* on page 2-45.

### Related references

*10.60 MRS (PSR to general-purpose register)* on page 10-376.
*10.63 MSR (general-purpose register to PSR)* on page 10-380.
*11.21 VMRS* on page 11-560.

## 4.20 Optional hash with immediate constants

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

This applies to ARM, Thumb, and VFP 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.
```

This can be suppressed 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 10 ARM and Thumb Instructions* on page 10-270.
*Chapter 11 VFP Instructions* on page 11-537.

## 4.21 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

*4.22 Test-and-branch macro example* on page 4-86.

*4.23 Unsigned integer division macro example* on page 4-87.

### Related references

*12.52 MACRO and MEND* on page 12-633.

## 4.22 Test-and-branch macro example

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

In ARM code in any processor and in Thumb code in processors before ARMv6T2, 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

*4.21 Use of macros* on page 4-85.
*4.23 Unsigned integer division macro example* on page 4-87.
*7.10 Numeric local labels* on page 7-141.

## 4.23 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

*4.21 Use of macros* on page 4-85.

*4.22 Test-and-branch macro example* on page 4-86.

*7.10 Numeric local labels* on page 7-141.

## 4.24 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 ARM and Thumb instructions, except the Thumb doubleword instruction, can be relocated.

`PLD`, `PLDW`, and `PLI`
: All ARM and Thumb instructions can be relocated.

`B`, `BL`, and `BLX`
: All ARM and Thumb instructions can be relocated.

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

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

`VLDR`
: Only ARM 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

*12.6 AREA* on page 12-582.
*12.28 EXPORT or GLOBAL* on page 12-606.
*12.46 IMPORT and EXTERN* on page 12-626.
*12.58 REQUIRE* on page 12-642.

*12.57 RELOC* on page 12-641.
*12.15 DCB* on page 12-593.
*12.16 DCD and DCDU* on page 12-594.
*12.22 DCW and DCWU* on page 12-600.
*10.43 LDR (PC-relative)* on page 10-347.
*10.11 ADR (PC-relative)* on page 10-295.
*10.71 PLD and PLI* on page 10-393.
*10.16 B* on page 10-305.
*10.25 CBZ and CBNZ* on page 10-319.
*10.40 LDC and LDC2* on page 10-340.
*11.12 VLDR (floating-point)* on page 11-551.

### Related information

*ELF for the ARM Architecture.*

## 4.25 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

*4.26 Exception tables and Unwind tables* on page 4-92.

### Related references

*12.3 About frame directives* on page 12-578.

### Related information

*Procedure Call Standard for the ARM Architecture.*

## 4.26 Exception tables and Unwind tables

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

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

*4.25 Frame directives* on page 4-91.

### Related references

*12.3 About frame directives* on page 12-578.
*9.29 --exceptions, --no_exceptions* on page 9-225.
*9.30 --exceptions_unwind, --no_exceptions_unwind* on page 9-226.
*12.40 FRAME UNWIND ON* on page 12-619.
*12.41 FRAME UNWIND OFF* on page 12-620.
*12.42 FUNCTION or PROC* on page 12-621.
*12.25 ENDFUNC or ENDP* on page 12-603.

### Related information

*Exception Handling ABI for the ARM Architecture.*

## 4.27 Assembly language changes after RVCT v2.1

The assembler accepts ARM and Thumb instructions written in either UAL or pre-UAL syntax. Some older versions of the assembler only accept pre-UAL syntax.

The assembly language accepted by the RVCT v2.1 assembler and earlier is called pre-UAL ARM and Thumb. In RVCT 2.2 and later, the assembler accepts both the UAL and the pre-UAL ARM and Thumb syntax. The assembler accepts the pre-UAL Thumb syntax only if it is preceded by a `CODE16` directive, or if the source file is assembled with the `--16` command-line option.

For the convenience of programmers who are familiar with the ARM assembly language accepted in RVCT v2.1 and earlier, the following table highlights the differences between the UAL and pre-UAL ARM assembly language syntax:

**Table 4-7 Changes from earlier ARM assembly language**

| Change | Pre-UAL ARM syntax | Preferred UAL syntax |
|---|---|---|
| The default addressing mode for `LDM` and `STM` is `IA` | `LDMIA, STMIA` | `LDM, STM` |
| You can use the `PUSH` and `POP` mnemonics for full, descending stack operations in ARM in addition to Thumb. | `STMFD sp!, {reglist}`<br>`LDMFD sp!, {reglist}` | `PUSH {reglist}`<br>`POP {reglist}` |
| You can use the `LSL`, `LSR`, `ASR`, `ROR`, and `RRX` instruction mnemonics for instructions with rotations and no other operation, in ARM in addition to Thumb. | `MOV Rd, Rn, LSL shift`<br>`MOV Rd, Rn, LSR shift`<br>`MOV Rd, Rn, ASR shift`<br>`MOV Rd, Rn, ROR shift`<br>`MOV Rd, Rn, RRX` | `LSL Rd, Rn, shift`<br>`LSR Rd, Rn, shift`<br>`ASR Rd, Rn, shift`<br>`ROR Rd, Rn, shift`<br>`RRX Rd, Rn` |
| Use the *label* form for PC-relative addressing. Do not use the *offset* form in new code. | `LDR Rd, [pc, #offset]` | `LDR Rd, label` |
| Specify both registers for doubleword memory accesses. You must still obey rules about the register combinations you can use. | `LDRD Rd, addr_mode` | `LDRD Rd, Rd2, addr_mode` |
| {*cond*}, if used, is always the last element of all instructions. | `ADD{cond}S`<br>`LDR{cond}SB` | `ADDS{cond}`<br>`LDRSB{cond}` |

In addition, some flexibility is permitted that was not permitted in previous assemblers as the following table shows:

**Table 4-8 Relaxation of requirements**

| Relaxation | Permitted syntax | Preferred syntax |
|---|---|---|
| If the destination register is the same as the first operand, you can use a two register form of the instruction. | `ADD r1, r3` | `ADD r1, r1, r3` |

You can write source code for Thumb processors earlier than ARMv6T2 using UAL.

If you are writing Thumb code for a processor earlier than ARMv6T2, you must restrict yourself to instructions that are available on the processor. The assembler generates error messages if you attempt to use an instruction that is not available.

If you are writing Thumb code for an ARMv6T2 or later processor, you can minimize your code size by using 16-bit instructions wherever possible.

The following table shows the main differences between the UAL and the pre-UAL Thumb assembly language:

**Table 4-9 Differences between pre-UAL Thumb syntax and UAL syntax**

| Change | Pre-UAL Thumb syntax | UAL syntax |
|---|---|---|
| The default addressing mode for `LDM` and `STM` is `IA` | `LDMIA`, `STMIA` | `LDM`, `STM` |
| You must use the `S` postfix on instructions that update the flags. This change is essential to avoid conflict with 32-bit Thumb instructions. | `ADD r1, r2, r3`<br>`SUB r4, r5, #6`<br>`MOV r0, #1`<br>`LSR r1, r2, #1` | `ADDS r1, r2, r3`<br>`SUBS r4, r5, #6`<br>`MOVS r0, #1`<br>`LSRS r1, r2, #1` |
| The preferred form for ALU instructions specifies three registers, even if the destination register is the same as the first operand. However, the UAL syntax allows the two register syntax. | `ADD r7, r8`<br>`SUB r1, #80` | `ADD r7, r7, r8`<br>`SUBS r1, r1, #80` |
| If *Rd* and *Rn* are both Lo registers, `MOV Rd, Rn` is disassembled as `ADDS Rd, Rn, #0`. | `MOV r2, r3`<br>`MOV r8, r9`<br>`CPY r0, r1`<br>`LSL r2, r3, #0` | `ADDS r2, r3, #0`<br>`MOV r8, r9`<br>`MOV r0, r1`<br>`MOVS r2, r3` |
| `NEG Rd, Rm` is disassembled as `RSBS Rd, Rm, #0`. | `NEG Rd, Rm` | `RSBS Rd, Rm, #0` |
| When using the `LDR Rd,=const` literal load pseudo-instruction, in pre-UAL syntax, the generated instruction might affect the condition code flags. In UAL syntax, the generated instruction sequence is guaranteed to not affect the condition code flags. | `LDR r0,=0`<br>`; generates the instruction:`<br>`MOVS r0,#0` | `LDR r0,=0`<br>`; generates the sequence:`<br>`LDR r0,{pc}+n`<br>`...`<br>`DCD 0` |

### Related references

*12.7 ARM or CODE32* on page 12-585.
*12.11 CODE16* on page 12-589.
*9.1 --16* on page 9-195.

# Chapter 5
# **Condition Codes**

Describes condition codes and the conditional execution of ARM and Thumb code.

It contains the following sections:

## 5.1 Conditional instructions

ARM and Thumb 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.

The instructions that you can make conditional depends on whether the processor is in ARM state or Thumb state.

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

*5.4 Updates to the condition flags* on page 5-99.

*5.2 Conditional execution in ARM state* on page 5-97.

*5.3 Conditional execution in Thumb state* on page 5-98.

### Related references

*5.5 Condition code suffixes and related flags* on page 5-100.

## 5.2 Conditional execution in ARM state

To execute ARM instructions conditionally you can either append a two letter suffix to the mnemonic, or you can use a conditional branch instruction.

Almost all ARM 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.

### Example conditional instructions to control execution

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

### Example conditional branch to control execution

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

### Related concepts

*5.3 Conditional execution in Thumb state* on page 5-98.

## 5.3 Conditional execution in Thumb state

To execute Thumb instructions conditionally, you can either use an `IT` instruction, or a conditional branch instruction.

In Thumb state on processors before ARMv6T2, the only mechanism for conditional execution is a conditional branch. You can conditionally skip over the instruction using a conditional branch instruction.

In Thumb state on ARMv6T2 or later processors, instructions can also be conditionally executed by using any of the following:

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

The Thumb `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 almost all Thumb instructions to be conditionally executed, based on the value of the condition flags and the condition code suffix specified. Each `IT` instruction provides conditional execution for up to four following instructions.

### Example conditional instructions using IT block

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

### Related concepts

*5.2 Conditional execution in ARM state* on page 5-97.

### Related references

*10.39 IT* on page 10-338.
*10.25 CBZ and CBNZ* on page 10-319.

## 5.4 Updates to the condition flags

Most ARM and Thumb data processing instructions only update the condition flags if you append an S suffix to the mnemonic. These instructions can update all or a subset of the flags.

In ARM state, and in Thumb state on ARMv6T2 or later processors, most data processing instructions have an option to update the condition flags in the *Application Program Status Register* (APSR) 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.

In Thumb state on processors before ARMv6T2, most data processing instructions update the condition flags automatically according to the result of the operation. There is no option to leave the flags unchanged and not update them. Other instructions cannot update the flags.

The instruction also determines the flags that get updated. Some instructions update all flags, and some instructions only update a subset of the flags. If a flag is not updated, the original value is preserved. The description of each ARM and Thumb instruction includes the effect 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.

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 concepts

*5.1 Conditional instructions* on page 5-96.

### Related references

*5.5 Condition code suffixes and related flags* on page 5-100.
*Chapter 10 ARM and Thumb Instructions* on page 10-270.

## 5.5 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 5-1 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 ARM instructions, and encoded in a preceding IT instruction for Thumb instructions. An instruction with a condition code is only executed if the condition flags in the APSR meet the specified condition.

In Thumb state on processors before ARMv6T2, the {*cond*} field is only permitted on certain branch instructions because there is no IT instruction on these processors.

The following is an example of conditional execution:

```
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

*5.1 Conditional instructions* on page 5-96.

*5.4 Updates to the condition flags* on page 5-99.

### Related references

*5.6 Comparison of condition code meanings in integer and floating-point code* on page 5-101.

*Chapter 10 ARM and Thumb Instructions* on page 10-270.

## 5.6 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 5-2 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

*5.1 Conditional instructions* on page 5-96.

*5.4 Updates to the condition flags* on page 5-99.

### Related references

*5.5 Condition code suffixes and related flags* on page 5-100.

*11.4 VCMP, VCMPE* on page 11-543.

### Related information

*ARM Architecture Reference Manual.*

## 5.7 Benefits of using conditional execution

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

You can use conditional execution of ARM instructions to reduce the number of branch instructions in your code. This improves code density. The `IT` instruction in Thumb 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, for example the ARM Cortex-R7 processor, 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

*5.8 Example showing the benefits of using conditional instructions* on page 5-104.

## 5.8 Example showing the benefits of using conditional instructions

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

This topic illustrates the difference between using branches and using conditional instructions. It uses the Euclid algorithm for the *Greatest Common Divisor* (gcd) to demonstrate 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:

**Note**

The detailed analysis of execution speed only applies to an ARM7™ processor. The code density calculations apply to all ARM processors.

### Example of conditional execution using branches in ARM code

This is an ARM code implementation of the gcd algorithm using branches, without using any other conditional instructions. Conditional execution is achieved 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 ARM
        B        gcd
less
        SUBS     r1, r1, r0  ; could be SUB r1, r1, r0 for ARM
        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 5-3 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 |

**Table 5-3 Conditional branches only (continued)**

| R0: a | R1: b | Instruction | Cycles (ARM7) |
|---|---|---|---|
| 1 | 1 | `BEQ end` | 3 |
| | | | Total = 13 |

### Example of conditional execution using conditional instructions in ARM code

This is an ARM code implementation of the gcd algorithm using individual conditional instructions in ARM 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 5-4 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 Thumb code

In ARMv6T2 and later architectures, you can use the `IT` instruction to write conditional instructions in Thumb code. The Thumb code implementation of the gcd algorithm using conditional instructions is very similar to the implementation in ARM code. The implementation in Thumb code is:

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

This assembles equally well to ARM or Thumb code. The assembler checks the `IT` instructions, but omits them on assembly to ARM code.

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

### Example of conditional execution code using branches in Thumb code

In architectures before ARMv6T2, there is no `IT` instruction and therefore Thumb instructions cannot be executed conditionally except for the `B` branch instruction. The gcd algorithm must be written with conditional branches and is very similar to the ARM code implementation using branches, without conditional instructions.

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

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

### Related concepts

*5.7 Benefits of using conditional execution* on page 5-103.
*5.9 Optimization for execution speed* on page 5-107.

### Related references

*10.39 IT* on page 10-338.
*5.5 Condition code suffixes and related flags* on page 5-100.

### Related information

*ARM Architecture Reference Manual.*

## 5.9 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 6
# Using the Assembler

Describes how to use the ARM assembler, `armasm`.

It contains the following sections:

## 6.1 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 or pre-UAL ARM or Thumb assembly language.

**Note**

The inline and embedded assemblers are part of the C and C++ compilers and do not use any command-line syntax for invocation. However, to pass additional assembler options when the compiler invokes `armasm` for embedded assembly, you can use the `armcc -A` option.

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.

### Related information

*Order of compiler command-line options.*

*Compiler command-line options listed by group.*

## 6.2 Specify command-line options with an environment variable

The ARMCC5_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 ARMCC5_ASMOPT and inserts it at the front of the command string. This means that options specified in ARMCC5_ASMOPT can be overridden by arguments on the command line.

### Related concepts

*6.1 armasm command-line syntax* on page 6-109.

### Related information

*Toolchain environment variables.*

## 6.3 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 input.o | armasm -o output.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 --bigend -o output.o -
   ```

2. Enter your input. For example:

   ```
           AREA     ARMex, CODE, READONLY
                                   ; Name this block of code ARMex
           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      ; ARM semihosting (formerly SWI)
           END                     ; Mark end of file
   ```

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

### Related concepts

*6.1 armasm command-line syntax* on page 6-109.
*13.1 Overview of via files* on page 13-654.

### Related references

*9.47 --maxcache=n* on page 9-244.

### Related information

*Rules for specifying command-line options.*

## 6.4 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 6-1 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 *PVVbbbb* where:<br>**P**<br>is the major version.<br>**VV**<br>is the minor version.<br>**bbbb**<br>is the build number. |
| \|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 32 if the assembler is assembling ARM code, or 16 if it is assembling Thumb code. |
| {CPU} | Holds the name of the selected processor. The default is "ARM7TDMI". If an architecture was specified in the command-line --cpu option, {CPU} holds the value "Generic ARM". |
| {ENDIAN} | Has the value "big" if the assembler is in big-endian mode, or "little" if it is in little-endian mode. |
| {FPIC} | Has the Boolean value {True} if --apcs=/fpic is set. The default is {False}. |
| {FPU} | Holds the name of the selected FPU. The default is "SoftVFP". |
| {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} = "4T"
```

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

The built-in variable |ads$version| must be all in lowercase. The names of the other 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 6-2 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 6-3 Predefined macros**

| Name | Value | Meaning |
|---|---|---|
| {TARGET_ARCH_ARM} | *num* | The number of the ARM base architecture of the target processor irrespective of whether the assembler is assembling for ARM or Thumb. |
| {TARGET_ARCH_THUMB} | *num* | The number of the Thumb base architecture of the target processor irrespective of whether the assembler is assembling for ARM or Thumb. The value is defined as zero if the target does not support Thumb. |
| {TARGET_ARCH_*XX*} | – | *XX* represents the target architecture and its value depends on the target processor. For example, if you specify the assembler option --cpu=4T or --cpu=ARM7TDMI then {TARGET_ARCH_4T} is defined. |
| {TARGET_FEATURE_EXTENSION_REGISTER_COUNT} | *num* | The number of 64-bit extension registers available in VFP. |
| {TARGET_FEATURE_CLZ} | – | If the target processor supports the CLZ instruction (that is, ARMv5T and later except ARMv6-M). |
| {TARGET_FEATURE_DIVIDE} | – | If the target processor supports the hardware divide instructions SDIV and UDIV. |
| {TARGET_FEATURE_DOUBLEWORD} | – | If the target processor supports the LDRD and STRD instructions (that is, ARMv5TE and later except ARMv6-M). |
| {TARGET_FEATURE_DSPMUL} | – | If the DSP-enhanced multiplier (for example the SMLA*xy* instruction) is available, for example in ARMv5TE. |

**Table 6-3 Predefined macros (continued)**

| Name | Value | Meaning |
|---|---|---|
| {TARGET_FEATURE_MULTIPLY} | – | If the target processor supports the long multiply instructions SMULL, SMLAL, UMULL, and UMLAL (that is, all architectures except ARMv6-M). |
| {TARGET_FEATURE_UNALIGNED} | – | If the target processor has support for unaligned access (that is, ARMv6 and later except ARMv6-M). |
| {TARGET_FPU_SOFTVFP} | – | If assembling with the option --fpu=softvfp. |
| {TARGET_FPU_SOFTVFP_VFP} | – | If assembling for a target processor with softvfp and VFP hardware, for example --fpu=softvfp+vfpv3. |
| {TARGET_FPU_VFP} | – | If assembling for a target processor with VFP hardware, without using softvfp, for example --fpu=vfpv3. |
| {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. |
| {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. |

The following table shows the possible values for {TARGET_ARCH_ARM} and {TARGET_ARCH_THUMB}, and for *XX* in the TARGET_ARCH_XX built-in variables. It also shows how these values relate to versions of the ARM architecture.

**Table 6-4 {TARGET_ARCH_ARM} in relation to {TARGET_ARCH_THUMB}**

| ARM architecture | {TARGET_ARCH_ARM} | {TARGET_ARCH_THUMB} | *xx* |
|---|---|---|---|
| v4 | 4 | 0 | 4 |
| v4T | 4 | 1 | 4T |
| v5T | 5 | 2 | 5T |
| v5TE | 5 | 2 | 5TE |
| v5TEJ | 5 | 2 | 5TEJ |
| v6 | 6 | 3 | 6 |
| v6K | 6 | 3 | 6K |
| v6Z | 6 | 3 | 6Z |
| v6T2 | 6 | 4 | 6T2 |
| v6-M | 0 | 3 | 6_M |
| v6S-M | 0 | 3 | 6S_M |
| v7-R | 7 | 4 | 7_R |
| v7-M | 0 | 4 | 7_M |
| v7E-M | 0 | 4 | 7E_M |

### Related concepts

*6.5 Identifying versions of armasm in source code* on page 6-116.

### Related references

*9.14 --cpu=name* on page 9-209.

*9.33 --fpu=name* on page 9-229.

## 6.5 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} / 1000000) >= 5
  ; using armasm in ARM Compiler 5 or above
ELSE
  ; using armasm in ARM Compiler 4.1 or earlier
ENDIF
```

The assembler also defines the built-in variable `|ads$version|` for legacy code. This variable did not exist before ADS and RVCT. If you have to build versions of your code using legacy development tools, you can test for the built-in variable `|ads$version|`. If this variable is not defined, then the assembler is part of a legacy development toolchain. Use code similar to the following:

```
IF :DEF: |ads$version|
  ; code for RealView or ADS
ELSE
  ; code for SDT (a legacy development toolchain)
ENDIF
```

### Related references

*6.4 Built-in variables and constants* on page 6-112.

## 6.6 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

*6.7 Interlocks diagnostics* on page 6-118.
*6.8 Automatic IT block generation* on page 6-119.
*6.9 Thumb branch target alignment* on page 6-120.
*6.10 Thumb code size diagnostics* on page 6-121.
*6.11 ARM and Thumb instruction portability diagnostics* on page 6-122.
*6.12 Instruction width diagnostics* on page 6-123.
*6.13 Two pass assembler diagnostics* on page 6-124.

### Related references

*9.18 --diag_error=tag[,tag,...]* on page 9-214.

## 6.7 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`.

### Related concepts

*6.8 Automatic IT block generation* on page 6-119.

*6.9 Thumb branch target alignment* on page 6-120.

*6.12 Instruction width diagnostics* on page 6-123.

*6.6 Diagnostic messages* on page 6-117.

### Related references

*9.22 --diag_warning=tag[,tag,...]* on page 9-218.

## 6.8 Automatic IT block generation

The assembler can automatically insert an IT block for conditional instructions in Thumb 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
```

the assembler generates the following instructions:

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

You can receive warning messages about this automatic generation of `IT` blocks when assembling Thumb code. To do this, use the following command-line option when invoking the assembler:

```
armasm --diag_warning 1763
```

### Related concepts

*6.6 Diagnostic messages* on page 6-117.

### Related references

*9.22 --diag_warning=tag[,tag,...]* on page 9-218.

## 6.9 Thumb branch target alignment

The assembler can issue warnings about non word-aligned branch targets in Thumb code.

On some processors, non word-aligned Thumb 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 the assembler reports such warnings, use the following command-line option when invoking the assembler:

```
armasm --diag_warning 1604
```

### Related concepts

*6.6 Diagnostic messages* on page 6-117.

### Related references

*9.22 --diag_warning=tag[,tag,...]* on page 9-218.

## 6.10 Thumb code size diagnostics

The assembler can issue a warning when it assembles a Thumb instruction to a 32-bit encoding when it could have used a 16-bit encoding.

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

To enable this warning, use the following command-line option when invoking the assembler:

```
armasm --diag_warning 1813
```

### Related concepts

*6.17 Instruction width selection in Thumb* on page 6-129.

*2.2 ARM, Thumb, and ThumbEE instruction sets* on page 2-31.

*6.6 Diagnostic messages* on page 6-117.

### Related references

*9.22 --diag_warning=tag[,tag,...]* on page 9-218.

## 6.11 ARM and Thumb instruction portability diagnostics

The assembler can issue warnings about instructions that cannot assemble to both ARM and Thumb code.

There are a few UAL instructions that can assemble as either ARM code or Thumb code, but not both. You can identify these instructions in the source code using the following command-line option when invoking the assembler:

```
armasm --diag_warning 1812
```

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 ARM, Thumb, and ThumbEE instruction sets* on page 2-31.

*6.6 Diagnostic messages* on page 6-117.

### Related references

*9.22 --diag_warning=tag[,tag,...]* on page 9-218.

## 6.12 Instruction width diagnostics

The assembler can issue a warning when it assembles a Thumb branch 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 can 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 following command-line option when invoking the assembler:

```
armasm --diag_warning 1607
```

**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

*6.6 Diagnostic messages* on page 6-117.

### Related references

*9.22 --diag_warning=tag[,tag,...]* on page 9-218.

## 6.13 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

*6.8 Automatic IT block generation* on page 6-119.
*6.9 Thumb branch target alignment* on page 6-120.
*6.12 Instruction width diagnostics* on page 6-123.
*6.6 Diagnostic messages* on page 6-117.
*1.3 How the assembler works* on page 1-25.

### Related references

*9.22 --diag_warning=tag[,tag,...]* on page 9-218.
*1.4 Directives that can be omitted in pass 2 of the assembler* on page 1-27.

## 6.14 Conditional assembly

Conditional assembly works differently from conditional compilation using the C preprocessor.

The C preprocessor performs textual transformations of macro identifiers into their definitions. Transformation occurs at the point at which the identifier is used. The C preprocessor is controlled by the following:

- Preprocessor directives embedded in the C source file, for example, `#define`.
- Compiler command-line options, for example `-D` and `-U`. These have the same effect as a `#define` or `#undef` directive at the beginning of each source file.

For example, in the following code, the C preprocessor replaces `y` with `x+1` at the point at which `y` is used, and therefore `example()` returns 0:

```
#define x 1
#define y x+1
#define x 2
int example()
{
#if y == 2
    return 1;
#else
    return 0;
#endif
}
```

Conditional assembly is based on variables, and works on each line in turn. Unlike the C preprocessor, the assembler evaluates expressions. Conditional assembly is controlled by the following:

- Assembler directives that declare and set the value of variables, for example `GBL`*x*, `LCL`*x* and `SET`*x*.
- Assembler directives that control the flow of the assembly, for example `WHILE`, `IF` and `ELSE`.
- Assembler directives that define macros, for example `MACRO`.
- The assembler command-line option `--predefine`, which pre-executes a `GBL`*x* and `SET`*x* directive.

For example, in the following code, the assembler evaluates `x+1` at the point at which the `SETA` directive occurs, and therefore `MOV` sets `r0` to 1:

```
    GBLA    x
    GBLA    y
x   SETA    1
y   SETA    x+1
x   SETA    2
    AREA    example, CODE
    IF y == 2
        MOV     r0, #1
    ELSE
        MOV     r0, #0
    ENDIF
    END
```

### Related references

*12.2 About assembly control directives* on page 12-577.

*9.57 --predefine "directive"* on page 9-254.

### Related information

*-Dname[(parm-list)][=def] compiler option.*

*-Uname compiler option.*

## 6.15 Using the C preprocessor

The assembler can invoke the compiler 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 when invoking the assembler. This causes `armasm` to call `armcc` to preprocess the file before assembling it.

`armasm` looks for the `armcc` 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 certain options to `armcc` if present on the command line. These are shown in the following table:

**Table 6-5 Command-line options**

| | | | |
|---|---|---|---|
| `--16` | `--bi` | `--diag_suppress` | `--li` |
| `--32` | `--cpu` | `--diag_warning` | `--library_type` |
| `--apcs` | `--diag_error` | `--fpu` | `--thumb` |
| `--arm` | `--diag_remark` | `--fpumode` | `--unaligned_access`, `--no_unaligned_access` |
| `--arm_only` | `--diag_style` | `-i` | |

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

**Table 6-6 armcc equivalent command-line options**

| armasm | armcc |
|---|---|
| `--16` | `--thumb` |
| `--32` | `--arm` |
| `-i` | `-I` |

To pass other simple compiler options, such as the preprocessor option `-D`, you must use the `--cpreproc_opts` command-line option. `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 --cpreproc --cpreproc_opts=-D,RELEASE,-U,ALPHA source.s
```

### Preprocessing an assembly language source file manually

If you want to use complex preprocessor options, you must manually call `armcc` to preprocess the file before calling `armasm`. The following example shows the commands you write to manually preprocess and assemble a file, `source.s`:

```
armcc -E source.s > preprocessed.s
armasm preprocessed.s
```

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

### Related references

*9.11 --cpreproc* on page 9-206.

*9.12 --cpreproc_opts=option[,option,...]* on page 9-207.

### Related information

*Compiler command-line options listed by group.*

## 6.16 Address alignment

The handling of unaligned addresses in load and store instructions depends on the ARM architecture version.

### ARMv7

In 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.

### ARMv5 and earlier

For word transfers, you must ensure that addresses are 4-byte aligned. Otherwise, if alignment checking is enabled, an alignment exception occurs. If alignment checking is unavailable, or if it is available but disabled, the following occur:

- For `STR`, `LDR`, `STM`, and `LDM`, the specified address is rounded down to a multiple of four.
- Additionally, for `LDR` only:
  1. Four bytes of data are loaded from the resulting address.
  2. The loaded data is rotated right by one, two or three bytes according to bits [1:0] of the address.

  For a little-endian memory system, this causes the addressed byte to occupy the least significant byte of the register. For a big-endian memory system, it causes the addressed byte to occupy:
  - Bits[31:24] if bit[0] of the address is 0.
  - Bits[15:8] if bit[0] of the address is 1.

Addresses must be halfword-aligned for halfword transfers, and doubleword-aligned for doubleword transfers.

### ARMv6

ARMv6 can be configured to support either the ARMv5 or ARMv7 alignment models, depending on the value of the U bit in the SCTLR. ARMv6-M faults all unaligned data accesses.

### --no_unaligned_access

If all your data accesses are aligned, you can use the `--no_unaligned_access` command-line option to avoid linking in any library functions that support unaligned accesses.

### Related references

*9.65 --unaligned_access, --no_unaligned_access* on page 9-262.

## 6.17 Instruction width selection in Thumb

If the assembler can select either a 16-bit or a 32-bit encoding for a Thumb instruction, in general it selects the 16-bit encoding. You can override this by specifying a `.W` or `.N` mnemonic qualifier.

If you are writing Thumb code for ARMv6T2 or later processors, some 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, the assembler 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, the assembler always generates a 32-bit instruction.
- In all other cases, the assembler 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. The assembler faults if it cannot generate an instruction with the specified width.

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

### Related concepts

*10.2 Instruction width specifiers* on page 10-281.
*6.10 Thumb code size diagnostics* on page 6-121.

# Chapter 7
# 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:

## 7.1 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|`, `|$t.x|`, or `|$d|` as program labels. These are mapping symbols that mark the beginning of ARM, Thumb, ThumbEE, and data within the object file.
- Symbols beginning with the characters $v are mapping symbols that relate to VFP and might be output when building for a target with VFP. 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

*7.10 Numeric local labels* on page 7-141.

### Related references

*2.10 Predeclared core register names* on page 2-40.

*2.11 Predeclared extension register names* on page 2-41.

*2.12 Predeclared coprocessor names* on page 2-42.

*6.4 Built-in variables and constants* on page 6-112.

## 7.2 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

*7.14 Numeric expressions* on page 7-145.

*7.12 String expressions* on page 7-143.

*7.3 Numeric constants* on page 7-134.

*7.17 Logical expressions* on page 7-148.

### Related references

*12.43 GBLA, GBLL, and GBLS* on page 12-622.

*12.50 LCLA, LCLL, and LCLS* on page 12-631.

*12.63 SETA, SETL, and SETS* on page 12-647.

## 7.3 Numeric constants

You can define 32-bit numeric constants using the `EQU` assembler directive.

Numeric constants are 32-bit integers. 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*.

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

*7.14 Numeric expressions* on page 7-145.

### Related references

*7.15 Syntax of numeric literals* on page 7-146.
*12.27 EQU* on page 12-605.

## 7.4 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

*3.1 Syntax of source lines in assembly language* on page 3-51.
*7.1 Symbol naming rules* on page 7-132.

## 7.5 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 ARM code, the value of the PC is the address of the current instruction plus 8 bytes.
- In Thumb 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.

### 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

*7.6 Labels* on page 7-137.

### Related references

*12.53 MAP* on page 12-636.

## 7.6 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

*7.7 Labels for PC-relative addresses* on page 7-138.

*7.8 Labels for register-relative addresses* on page 7-139.

*7.9 Labels for absolute addresses* on page 7-140.

### Related references

*3.1 Syntax of source lines in assembly language* on page 3-51.

*12.28 EXPORT or GLOBAL* on page 12-606.

## 7.7 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 ARM to Thumb state or Thumb to ARM state in this way, the processor does not change the state properly.

### Related references

*12.6 AREA* on page 12-582.
*12.15 DCB* on page 12-593.
*12.16 DCD and DCDU* on page 12-594.
*12.18 DCFD and DCFDU* on page 12-596.
*12.19 DCFS and DCFSU* on page 12-597.
*12.20 DCI* on page 12-598.
*12.21 DCQ and DCQU* on page 12-599.
*12.22 DCW and DCWU* on page 12-600.

## 7.8 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.

### Example of storage map definitions

```
        MAP     0,r9
        MAP     0xff,r9
```

### Related references

*12.17 DCDO* on page 12-595.
*12.27 EQU* on page 12-605.
*12.53 MAP* on page 12-636.
*12.64 SPACE or FILL* on page 12-648.

## 7.9 Labels for absolute addresses

A label can represent the absolute address of code or data.

These labels are numeric constants in the range 0 to $2^{32}$-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:

- ARM code with the `ARM` directive.
- Thumb 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 ARM code
```

### Related concepts

*7.6 Labels* on page 7-137.
*7.7 Labels for PC-relative addresses* on page 7-138.
*7.8 Labels for register-relative addresses* on page 7-139.

### Related references

*12.27 EQU* on page 12-605.

## 7.10 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

*7.6 Labels* on page 7-137.

### Related references

*3.1 Syntax of source lines in assembly language* on page 3-51.
*7.11 Syntax of numeric local labels* on page 7-142.
*12.52 MACRO and MEND* on page 12-633.
*12.49 KEEP* on page 12-630.
*12.62 ROUT* on page 12-646.

## 7.11 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

*7.10 Numeric local labels* on page 7-141.

### Related references

*12.62 ROUT* on page 12-646.

## 7.12 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

*7.13 String literals* on page 7-144.
*7.19 Unary operators* on page 7-150.
*7.2 Variables* on page 7-133.

### Related references

*7.22 String manipulation operators* on page 7-153.
*12.63 SETA, SETL, and SETS* on page 12-647.

## 7.13 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

*3.1 Syntax of source lines in assembly language* on page 3-51.
*9.50 --no_esc* on page 9-247.

## 7.14 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. 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}`.

### Example

```
a   SETA    256*256          ; 256*256 is a numeric expression
    MOV     r1,#(a*22)       ; (a*22) is a numeric expression
```

### Related concepts

*7.20 Binary operators* on page 7-151.

*7.2 Variables* on page 7-133.

*7.3 Numeric constants* on page 7-134.

### Related references

*7.15 Syntax of numeric literals* on page 7-146.

*12.63 SETA, SETL, and SETS* on page 12-647.

## 7.15 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*.
- 0x*hexadecimal-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 the range 0 to $2^{32}$-1 (except in `DCQ` and `DCQU` directives, where the range is 0 to $2^{64}$-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

*7.3 Numeric constants* on page 7-134.

## 7.16 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 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 VFP.

### Examples

```
    DCFD    1E308,-4E-100
    DCFS    1.0
    DCFS    0.02
    DCFD    3.725e15
    DCFS    0x7FC00000              ; Quiet NaN
    DCFD    &FFF0000000000000       ; Minus infinity
```

### Related concepts

*7.3 Numeric constants* on page 7-134.

### Related references

*7.15 Syntax of numeric literals* on page 7-146.

## 7.17 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

*7.26 Boolean operators* on page 7-157.
*7.25 Relational operators* on page 7-156.

## 7.18 Logical literals

Logical or Boolean literals can have one of two values, `{TRUE}` or `{FALSE}`.

### Related concepts

*7.13 String literals* on page 7-144.

### Related references

*7.15 Syntax of numeric literals* on page 7-146.

## 7.19 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 7-1 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` | Returns an 8-digit hexadecimal string corresponding to a numeric expression, or the string `"T"` or `"F"` if used on a logical expression. |
| `: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 7-2 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`<br>`-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, 0-15 corresponding to R0-R15. |

### Related concepts

*7.20 Binary operators* on page 7-151.

## 7.20 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

*7.28 Difference between operator precedence in assembly language and C* on page 7-159.

### Related references

*7.21 Multiplicative operators* on page 7-152.
*7.22 String manipulation operators* on page 7-153.
*7.23 Shift operators* on page 7-154.
*7.24 Addition, subtraction, and logical operators* on page 7-155.
*7.25 Relational operators* on page 7-156.
*7.26 Boolean operators* on page 7-157.

## 7.21 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 7-3 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

*7.20 Binary operators* on page 7-151.

*7.5 Register-relative and PC-relative expressions* on page 7-136.

*7.14 Numeric expressions* on page 7-145.

### Related references

*7.15 Syntax of numeric literals* on page 7-146.

## 7.22 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 7-4 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

*7.12 String expressions* on page 7-143.
*7.14 Numeric expressions* on page 7-145.

## 7.23 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 7-5 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

*7.20 Binary operators* on page 7-151.

## 7.24 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 7-6 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

*7.20 Binary operators* on page 7-151.

## 7.25 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 7-7 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

*7.20 Binary operators* on page 7-151.

## 7.26 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 7-8 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

*7.20 Binary operators* on page 7-151.

## 7.27 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

*7.19 Unary operators* on page 7-150.

*7.20 Binary operators* on page 7-151.

*7.28 Difference between operator precedence in assembly language and C* on page 7-159.

### Related references

*7.21 Multiplicative operators* on page 7-152.

*7.22 String manipulation operators* on page 7-153.

*7.23 Shift operators* on page 7-154.

*7.24 Addition, subtraction, and logical operators* on page 7-155.

*7.25 Relational operators* on page 7-156.

*7.26 Boolean operators* on page 7-157.

## 7.28 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 7-9 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 7-10 Operator precedence in C**

| C precedence |
|---|
| unary operators |
| `* / %` |
| `+ -` (as binary operators) |
| `<< >>` |
| `< <= > >=` |
| `== !=` |
| `&` |
| `^` |
| `\|` |
| `&&` |
| `\|\|` |

### Related concepts

*7.20 Binary operators* on page 7-151.

### Related references

*7.27 Operator precedence* on page 7-158.

# Chapter 8
# VFP Programming

Describes the assembly programming of VFP hardware.

It contains the following sections:

## 8.1 Architecture support for VFP

VFP is an optional extension to the ARM architecture. There are versions that provide additional instructions.

Most VFP and the shared instructions are available in all versions of the VFP architecture. Where this is not true, the descriptions of the instructions specify the applicable VFP architecture versions.

VFPv3 has variants that do not support all VFPv3 registers and floating-point data types. VFPv3 with half-precision extension and fused multiply-add extension is called VFPv4. There is a single-precision only version of VFPv4, called FPv4-SP.

For details of the implemented VFP architecture and variant, you must always refer to the Technical Reference Manual for your processor. To get VFP, you must specify the FPU or have it implicit in the processor specified with `--cpu`.

VFP instructions, including the half-precision and fused multiply-add instructions, are treated as Undefined Instructions on systems that do not support the necessary architecture extension. Even on systems that support VFP, the instructions are undefined if the necessary coprocessors are not enabled in the Coprocessor Access Control Register (CP15 CPACR).

### Related concepts

*8.2 Half-precision extension for VFP* on page 8-164.

*8.3 Fused Multiply-Add extension for VFP* on page 8-165.

*8.16 VFP vector mode* on page 8-179.

### Related information

*Floating-point support.*

*Further reading.*

## 8.2 Half-precision extension for VFP

The Half-precision extension optionally extends the VFPv3 architecture.

It provides VFP instructions that perform conversion between single-precision (32-bit) and half-precision (16-bit) floating-point numbers.

The half-precision instructions are only available on VFP systems that implement the half-precision extension. The VFP variants that implement the half-precision extension are VFPv3-FP16, VFPv3-D16-FP16, and VFPv4.

### Related concepts

*8.1 Architecture support for VFP* on page 8-163.

## 8.3 Fused Multiply-Add extension for VFP

The Fused Multiply-Add extension optionally extends the VFPv3 architecture.

It provides VFP instructions that perform multiply and accumulate operations with a single rounding step, so suffers from less loss of accuracy than performing a multiplication followed by an add.

The fused multiply-add instructions are only available on VFP systems that implement the fused multiply-add extension. The VFP system that implements the fused multiply-add extension is VFPv4.

### Related concepts

*8.1 Architecture support for VFP* on page 8-163.

## 8.4 Extension register bank mapping in VFP

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

The following figure shows the views of the VFP 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`, `S1`.

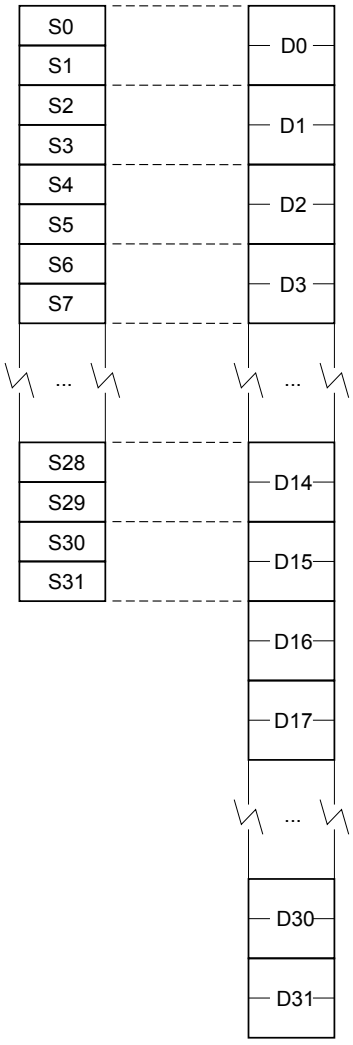


**Figure 8-1 VFP extension register bank**

**Note**

The figure applies to a VFP implementation with 32 double precision registers. The following versions of VFP use 16 double precision registers, `D0-D15`.

- VFPv2.
- VFPv3-D16.
- VFPv4-D16.

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 the elements of a vector in `D6` by referring to `S12`, and the most significant half of the elements by referring to `S13`.

### Related concepts

*8.5 VFP views of the extension register bank* on page 8-168.

## 8.5 VFP views of the extension register bank

VFP can view the extension register bank as thirty-two 32-bit registers, or as either sixteen or thirty-two 64-bit registers, depending on the VFP version.

VFPv3, VFPv3-FP16, and VFPv4 can view the extension register bank 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.

VFPv2, VFPv3-D16, VFPv3-D16-FP16, and VFPv4-D16 can view the extension register bank as:

- Sixteen 64-bit registers, `D0-D15`.
- Thirty-two 32-bit registers, `S0-S31`.
- A combination of registers from these views.

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

### Related concepts

*8.4 Extension register bank mapping in VFP* on page 8-166.

## 8.6 Load values to VFP registers

There are different ways to load immediate values into VFP registers.

In VFPv3 and later, the `VMOV` and `VMVN` instructions load a limited range of floating-point immediate values.

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

### Related references

*11.14 VLDR pseudo-instruction* on page 11-553.
*11.17 VMOV (floating-point)* on page 11-556.

## 8.7 Conditional execution of VFP instructions

You can execute VFP instructions conditionally, in the same way as most ARM and Thumb instructions.

In ARM state, you can use a condition code to control the execution of VFP instructions. The instruction is executed conditionally, according to the status flags in the APSR, in the same way as almost all ARM instructions.

In Thumb state, you can use an `IT` instruction to set condition codes on up to four following VFP instructions.

### Related concepts

*5.2 Conditional execution in ARM state* on page 5-97.

*5.3 Conditional execution in Thumb state* on page 5-98.

### Related references

*5.6 Comparison of condition code meanings in integer and floating-point code* on page 5-101.

*10.8 Condition code suffixes* on page 10-289.

## 8.8 Floating-point exceptions in VFP

The VFP 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 VFP 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. See also the Technical Reference Manual for your processor.

### Related concepts

*8.12 Flush-to-zero mode* on page 8-175.

### Related references

*Chapter 11 VFP Instructions* on page 11-537.

### Related information

*ARM Architecture Reference Manual.*

*Further reading.*

## 8.9 VFP data types

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

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

The following table shows the data type specifiers available in VFP instructions:

**Table 8-1 VFP data type specifiers**

| | 16-bit | 32-bit | 64-bit |
|---|---|---|---|
| Unsigned integer | `U16` | `U32` | not available |
| Signed integer | `S16` | `S32` | not available |
| Floating-point number | `F16` | `F32` (or `F`) | `F64` (or `D`) |

The data type 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 pages for details. However, the data type description is flexible:
  - 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.
- The `F16` data type is only available on systems that implement the half-precision architecture extension.

## 8.10 Extended notation extension for VFP

The assembler supports an extension to the architectural VFP assembly syntax, called extended notation. This allows you to define register names that include data type specifiers or scalar indexes, for convenience.

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

*8.9 VFP data types* on page 8-172.

### Related references

*12.23 DN and SN* on page 12-601.

## 8.11 VFP system registers

Some VFP system registers are accessible in all implementations of VFP.

These registers are:

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

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

### Related concepts

*4.19 The Read-Modify-Write operation* on page 4-83.

### Related information

*ARM Architecture Reference Manual.*

*Further reading.*

## 8.12 Flush-to-zero mode

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 VFP 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.

In VFPv2, flush-to-zero mode can either preserve the sign bit or flush to +0, depending on the implementation. Flush-to-zero mode in VFPv3 and VFPv4 always preserves the sign bit.

### Related concepts

*8.14 The effects of using flush-to-zero mode in VFP* on page 8-177.

### Related references

*8.13 When to use flush-to-zero mode in VFP* on page 8-176.

*8.15 VFP operations not affected by flush-to-zero mode* on page 8-178.

## 8.13 When to use flush-to-zero mode in VFP

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

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

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

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

Numbers already in registers are not affected by changing mode.

### Related concepts

*8.12 Flush-to-zero mode* on page 8-175.

*8.14 The effects of using flush-to-zero mode in VFP* on page 8-177.

## 8.14 The effects of using flush-to-zero mode in VFP

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

*8.12 Flush-to-zero mode* on page 8-175.

### Related references

*8.15 VFP operations not affected by flush-to-zero mode* on page 8-178.

## 8.15 VFP operations not affected by flush-to-zero mode

Some VFP 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

*8.12 Flush-to-zero mode* on page 8-175.

### Related references

*11.2 VABS (floating-point)* on page 11-541.
*11.24 VNEG (floating-point)* on page 11-563.
*11.12 VLDR (floating-point)* on page 11-551.
*11.32 VSTR (floating-point)* on page 11-571.
*11.11 VLDM (floating-point)* on page 11-550.
*11.31 VSTM (floating-point)* on page 11-570.
*11.18 VMOV (between one ARM register and single precision VFP)* on page 11-557.
*11.19 VMOV (between two ARM registers and one or two extension registers)* on page 11-558.

## 8.16 VFP vector mode

VFP vector mode allows you to use VFP instructions on vectors of floating-point numbers. ARM deprecates VFP vector mode.

Usually the VFP core only works on a single register. However, many VFP arithmetic instructions can also operate on vectors of up to eight single-precision or four double-precision numbers, enabling *Single Instruction Multiple Data* (SIMD) vectorization.

In addition, the floating-point load and store instructions have multiple register forms, enabling vectors to be transferred to and from memory easily.

**Note**

ARM deprecates the use of VFP vector mode.

### Related concepts

*8.1 Architecture support for VFP* on page 8-163.

*8.17 Vectors in the VFP extension register bank* on page 8-180.

### Related information

*ARM Architecture Reference Manual.*

## 8.17 Vectors in the VFP extension register bank

In VFP vector mode, the VFP extension register bank can be viewed as a collection of smaller banks. A vector consists of multiple registers from the same bank. Each of these smaller banks is treated either as a bank of 8 single-precision registers or 4 double-precision registers.

In VFPv2, VFPv3-D16, and VFPv3-D16-FP16 the VFP extension register bank can be viewed as a collection of:

- Four banks of single-precision registers, s0 to s7, s8 to s15, s16 to s23, and s24 to s31.
- Four banks of double-precision registers, d0 to d3, d4 to d7, d8 to d11, and d12 to d15.
- Any combination of single-precision and double-precision banks.

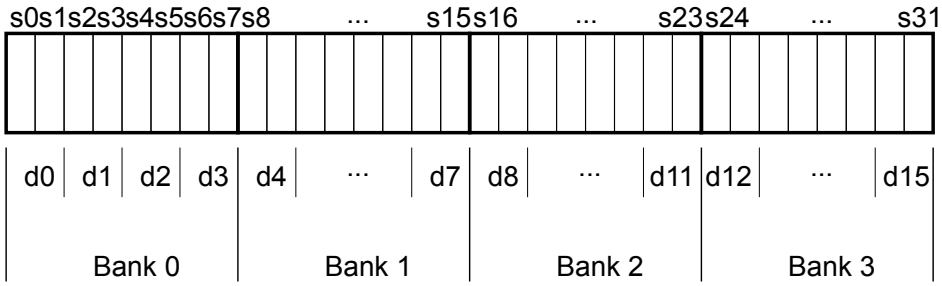


**Figure 8-2 VFPv2 register banks**

In VFPv3 and VFPv3-FP16, the VFP extension register bank can be viewed as a collection of:

- Four banks of single-precision registers, s0 to s7, s8 to s15, s16 to s23, and s24 to s31.
- Eight banks of double-precision registers, d0 to d3, d4 to d7, d8 to d11, d12 to d15, d16 to d19, d20 to d23, d24 to d27, and d28 to d31.
- Any combination of single-precision and double-precision banks.

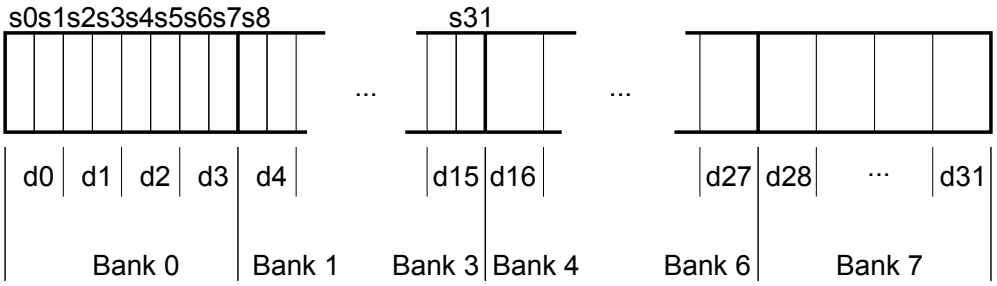


**Figure 8-3 VFPv3 register banks**

A vector, in a VFP instruction, can use up to eight single-precision registers, or four double-precision registers, from the same bank. The number of registers used by a vector is controlled by the `LEN` bits in the `FPSCR`.

**Note**

The value of the `LEN` bits is not a sufficient condition to perform vector operations using VFP. Whether a VFP operation is scalar, vector or mixed depends on which bank the specified operand and destination registers are in.

A vector can start from any register and wraps around to the beginning of the bank. The first register used by an operand vector is the register that is specified as the operand in the individual VFP instructions. The first register used by the destination vector is the register that is specified as the destination in the individual VFP instructions.

### Related concepts

*8.18 VFP vector wrap-around* on page 8-182.

*8.19 VFP vector stride* on page 8-183.
*8.20 Restriction on vector length* on page 8-184.
*8.21 Control of scalar, vector, and mixed operations* on page 8-185.

### Related information

*ARM Architecture Reference Manual.*

## 8.18 VFP vector wrap-around

In VFP vector mode, if a vector extends beyond the end of a bank, it wraps around to the beginning of the same bank.

For example:

- A vector of length 6 starting at s5 is {s5, s6, s7, s0, s1, s2}.
- A vector of length 3 starting at s15 is {s15, s8, s9}.
- A vector of length 4 starting at s22 is {s22, s23, s16, s17}.
- A vector of length 2 starting at d7 is {d7, d4}.
- A vector of length 3 starting at d10 is {d10, d11, d8}.

A vector cannot contain registers from more than one bank.

### Related concepts

*8.17 Vectors in the VFP extension register bank* on page 8-180.

*8.20 Restriction on vector length* on page 8-184.

*8.19 VFP vector stride* on page 8-183.

### Related information

*ARM Architecture Reference Manual.*

## 8.19 VFP vector stride

In VFP vector mode, vectors can occupy consecutive or alternate registers. This is controlled by the STRIDE bits in the FPSCR.

For example:

- A vector of length 3, stride 2, starting at s1, is {s1, s3, s5}.
- A vector of length 4, stride 2, starting at s6, is {s6, s0, s2, s4}.
- A vector of length 2, stride 2, starting at d1, is {d1, d3}.
- A vector of length 4, stride 1, starting at d0, is {d0, d1, d2, d3}.

### Related concepts

*8.17 Vectors in the VFP extension register bank* on page 8-180.

*8.18 VFP vector wrap-around* on page 8-182.

*8.20 Restriction on vector length* on page 8-184.

### Related information

*ARM Architecture Reference Manual.*

## 8.20 Restriction on vector length

In VFP vector mode, a vector cannot use the same register twice. This means that vector length is restricted.

Enabling for vector wrap-around, you cannot have:

- A single-precision vector with length > 4 and stride = 2.
- A double-precision vector with length > 4 and stride = 1.
- A double-precision vector with length > 2 and stride = 2.

### Related concepts

*8.17 Vectors in the VFP extension register bank* on page 8-180.

*8.18 VFP vector wrap-around* on page 8-182.

### Related information

*ARM Architecture Reference Manual.*

## 8.21 Control of scalar, vector, and mixed operations

Whether a VFP arithmetic instruction operates on scalars, vectors, or a mixture of both depends on the LEN bits in the FPSCR and also on which register bank the destination and operand registers are in.

Use the `LEN` bits in the `FPSCR` to control the length of vectors. When `LEN` is 1 all VFP operations are scalar.

When `LEN` is greater than 1, the VFP operation can be scalar, vector or mixed. The behavior of VFP arithmetic operations depends on which register bank the destination and operand registers are in.

The first bank of registers, s0 to s7 or d0 to d3 and the fifth bank of registers d16 to d19 are scalar banks. All other banks are vector banks. A vector operation or mixed operation is one where the destination register is in one of the vector banks.

Given instructions of the following general forms:

```
Op  Fd,Fn,Fm
Op  Fd,Fm
```

where:

*Op*
: is the VFP instruction.

*Fd*
: is the destination register.

*Fn*
: is an operand register.

*Fm*
: is the only or second operand register.

the behavior of the operation is as follows:

- If *Fd* is in the first or fifth bank of registers then the operation is scalar.
- If *Fm* is in the first or fifth bank of registers, but *Fd* is not, then the operation is mixed.
- If neither *Fd* nor *Fm* are in the first or fifth bank of registers, the operation is vector.

In scalar operations, *Op* acts on the value in *Fm*, and the value in *Fn* if present. The result is placed in *Fd*.

In vector operations, *Op* acts on the values in the vector starting at *Fm*, together with the values in the vector starting at *Fn* if present. The results are placed in the vector starting at *Fd*.

In mixed operations, with a single operand, *Op* acts on the single value in *Fm* and `LEN` copies of the result are placed in the vector starting at *Fd*.

In mixed operations, with two operands, *Op* acts on the single value in *Fm*, together with the values in the vector starting at *Fn*. The results are placed in the vector starting at *Fd*.

### Related concepts

*8.17 Vectors in the VFP extension register bank* on page 8-180.
*8.18 VFP vector wrap-around* on page 8-182.
*8.19 VFP vector stride* on page 8-183.
*8.20 Restriction on vector length* on page 8-184.

### Related information

*ARM Architecture Reference Manual.*

## 8.22 Overview of VFP directives and vector notation

To use vector notation, you must use pre-UAL syntax and mnemonics. You can use assembler directives to check you are using the correct syntax.

This applies only to `armasm`. The inline assemblers in the C and C++ compilers do not accept these directives or vector notation.

The use of VFP vector mode is deprecated, and vector notation is not supported in UAL. To use vector notation, you must use the pre-UAL mnemonics. You can mix pre-UAL VFP mnemonics and UAL VFP mnemonics.

You can make assertions about VFP vector lengths and strides in your code, and have them checked by the assembler, by using the following directives:

- `VFPASSERT SCALAR`.
- `VFPASSERT VECTOR`.

If you use the `VFPASSERT` directives, you must specify vector details in all VFP data processing instructions written using pre-UAL mnemonics. If you do not use the `VFPASSERT` directives you must not use this notation.

### Related concepts

*8.23 Pre-UAL VFP syntax and mnemonics* on page 8-187.

### Related references

*8.25 VFPASSERT SCALAR* on page 8-190.
*8.26 VFPASSERT VECTOR* on page 8-191.
*8.24 Vector notation* on page 8-189.

## 8.23 Pre-UAL VFP syntax and mnemonics

There are differences between pre-UAL and UAL syntax and mnemonics for VFP instructions.

Where UAL mnemonics use `.F32` to specify single-precision data, pre-UAL mnemonics use `S` appended to the instruction mnemonic. For example, `VABS.F32` was `FABSS`.

Where UAL mnemonics use `.F64` to specify double-precision data, pre-UAL mnemonics use `D` appended to the instruction mnemonic. For example, `VCMPE.F64` was `FCMPED`.

### Pre-UAL VFP mnemonics

The following table shows the pre-UAL mnemonics of those instructions that are affected by VFP vector mode. All other VFP instructions are always scalar regardless of the settings of `LEN` and `STRIDE`.

**Table 8-2 Pre-UAL VFP mnemonics**

| UAL mnemonic | Equivalent pre-UAL mnemonic |
|---|---|
| VABS | FABS |
| VADD | FADD |
| VDIV | FDIV |
| VMLA | FMAC |
| VMLS | FNMAC |
| VMOV (immediate) | FCONST [a] |
| VMOV (register) | FCPY |
| VMUL | FMUL |
| VNEG | FNEG |
| VNMLA | FNMSC |
| VNMLS | FMSC |
| VNMUL | FNMUL |
| VSQRT | FSQRT |
| VSUB | FSUB |

### Immediate values in FCONST

The following table shows the floating-point values you can load using `FCONST`. Trailing zeroes are omitted for clarity. The immediate value you must put in the `FCONST` instruction is the decimal representation of the binary number `abcdefgh`, where:

**a**
: is 0 for positive numbers, or 1 for negative numbers.

**bcd**
: is shown in the column headings.

**efgh**
: is shown in the row headings.

Alternatively, you can use `0x` followed by the hexadecimal representation.

[a] The immediate in `VMOV` (immediate) is the floating-point number you want to load. The immediate in `FCONST` is the number encoded in the instruction to produce the floating-point number you want to load.

**Table 8-3 Floating-point values for use with FCONST**

| bcd | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
|---|---|---|---|---|---|---|---|---|
| **efgh** | | | | | | | | |
| 0000 | 2.0 | 4.0 | 8.0 | 16.0 | 0.125 | 0.25 | 0.5 | 1.0 |
| 0001 | 2.125 | 4.25 | 8.5 | 17.0 | 0.1328125 | 0.265625 | 0.53125 | 1.0625 |
| 0010 | 2.25 | 4.5 | 9.0 | 18.0 | 0.140625 | 0.28125 | 0.5625 | 1.125 |
| 0011 | 2.375 | 4.75 | 9.5 | 19.0 | 0.1484375 | 0.296875 | 0.59375 | 1.1875 |
| 0100 | 2.5 | 5.0 | 10.0 | 20.0 | 0.15625 | 0.3125 | 0.625 | 1.25 |
| 0101 | 2.625 | 5.25 | 10.5 | 21.0 | 0.1640625 | 0.328125 | 0.65625 | 1.3125 |
| 0110 | 2.75 | 5.5 | 11.0 | 22.0 | 0.171875 | 0.34375 | 0.6875 | 1.375 |
| 0111 | 2.875 | 5.75 | 11.5 | 23.0 | 0.1796875 | 0.359375 | 0.71875 | 1.4375 |
| 1000 | 3.0 | 6.0 | 12.0 | 24.0 | 0.1875 | 0.375 | 0.75 | 1.5 |
| 1001 | 3.125 | 6.25 | 12.5 | 25.0 | 0.1953125 | 0.390625 | 0.78125 | 1.5625 |
| 1010 | 3.25 | 6.5 | 13.0 | 26.0 | 0.203125 | 0.40625 | 0.8125 | 1.625 |
| 1011 | 3.375 | 6.75 | 13.5 | 27.0 | 0.2109375 | 0.421875 | 0.84375 | 1.6875 |
| 1100 | 3.5 | 7.0 | 14.0 | 28.0 | 0.21875 | 0.4375 | 0.875 | 1.75 |
| 1101 | 3.625 | 7.25 | 14.5 | 29.0 | 0.2265625 | 0.453125 | 0.90625 | 1.8125 |
| 1110 | 3.75 | 7.5 | 15.0 | 30.0 | 0.234375 | 0.46875 | 0.9375 | 1.875 |
| 1111 | 3.875 | 7.75 | 15.5 | 31.0 | 0.2421875 | 0.484375 | 0.96875 | 1.9375 |

## 8.24 Vector notation

In vector notation, you specify vectors of VFP registers using angle brackets.

You specify scalar and vector registers in pre-UAL VFP data processing instructions as follows:

- `s`*n* is a single-precision scalar register *n*.
- `s`*n*`<>` is a single-precision vector whose length and stride are given by the current vector length and stride, as defined by `VFPASSERT VECTOR`. The vector starts at register *n*.
- `s`*n*`<`*L*`>` is a single-precision vector of length L, stride 1. The vector starts at register *n*.
- `s`*n*`<`*L*`:`*S*`>` is a single-precision vector of length *L*, stride *S*. The vector starts at register *n*.
- `d`*n* is a double-precision scalar register *n*.
- `d`*n*`<>` is a double-precision vector whose length and stride are given by the current vector length and stride, as defined by `VFPASSERT VECTOR`. The vector starts at register *n*.
- `d`*n*`<`*L*`>` is a double-precision vector of length L, stride 1. The vector starts at register *n*.
- `d`*n*`<`*L*`:`*S*`>` is a double-precision vector of length *L*, stride *S*. The vector starts at register *n*.

You can use this vector notation with names defined using the `DN` and `SN` directives.

You must not use this vector notation in the `DN` and `SN` directives themselves.

### Related references

*8.25 VFPASSERT SCALAR* on page 8-190.
*8.26 VFPASSERT VECTOR* on page 8-191.
*12.23 DN and SN* on page 12-601.

## 8.25 VFPASSERT SCALAR

The `VFPASSERT SCALAR` directive informs the assembler that the following VFP instructions are in scalar mode. This forces the instruction syntax to be scalar.

### Syntax

```
VFPASSERT SCALAR
```

### Usage

Use the `VFPASSERT SCALAR` directive to mark the end of any block of code where the VFP mode is `VECTOR`.

Place the `VFPASSERT SCALAR` directive immediately after the instruction where the change occurs. This is usually an `FMXR` instruction, but might be a `BL` instruction.

If a function expects VFP to be in vector mode on exit, place a `VFPASSERT SCALAR` directive immediately after the last instruction. Such a function would not be AAPCS compliant.

**Note**

This directive does not generate any code. It is only an assertion by the programmer. The assembler produces error messages if any such assertions are inconsistent with each other, or with any vector notation in VFP data processing instructions.

The assembler faults vector notation in VFP data processing instructions following a `VFPASSERT SCALAR` directive, even if the vector length is 1.

### Example

```
    VFPASSERT   SCALAR            ; scalar mode
    faddd       d4, d4, d0        ; okay
    fadds       s4<3>, s8<3>, s0  ; ERROR, vectors in scalar mode
    fabss       s24<1>, s28<1>    ; ERROR, vectors in scalar mode
                                  ; (even though length==1)
```

### Related references

*8.24 Vector notation* on page 8-189.
*8.26 VFPASSERT VECTOR* on page 8-191.

### Related information

*Procedure Call Standard for the ARM Architecture.*

## 8.26 VFPASSERT VECTOR

The `VFPASSERT VECTOR` directive informs the assembler that the following VFP instructions are in vector mode. It can also specify the length and stride of the vectors.

### Syntax

`VFPASSERT VECTOR{<{n{:s}}>}`

where:

*n*
is the vector length, 1-8.

*s*
is the vector stride, 1-2.

### Usage

Use the `VFPASSERT VECTOR` directive to mark the start of a block of instructions where the VFP mode is `VECTOR`, and to mark changes in the length or stride of vectors.

Place the `VFPASSERT VECTOR` directive immediately after the instruction where the change occurs. This is usually an `FMXR` instruction, but might be a `BL` instruction.

If a function expects VFP to be in vector mode on entry, place a `VFPASSERT VECTOR` directive immediately before the first instruction. Such a function would not be AAPCS compliant.

**Note**

This directive does not generate any code. It is only an assertion by the programmer. The assembler produces error messages if any such assertions are inconsistent with each other, or with any vector notation in VFP data processing instructions.

### Example

```
VMRS    r10,FPSCR          ; UAL mnemonic - could be FMRX instead.
BIC     r10,r10,#0x00370000
ORR     r10,r10,#0x00020000    ; set length = 3, stride = 1
VMSR    FPSCR,r10
VFPASSERT VECTOR           ; assert vector mode, unspecified length
                           ; and stride
faddd  d4, d4, d0          ; ERROR, scalars in vector mode
fadds  s16<3>, s8<3>, s0   ; okay
fabss  s24<1>, s28<1>      ; wrong length, but not faulted
                           ; (unspecified)
VMRS    r10,FPSCR
BIC     r10,r10,#0x00370000
ORR     r10,r10,#0x00030000    ; set length = 4, stride = 1
VMSR    FPSCR,r10
VFPASSERT VECTOR<4>        ; assert vector mode, length 4, stride 1
fadds  s24<4>, s8<4>, s0   ; okay
fabss  s24<2>, s24<2>      ; ERROR, wrong length
VMRS    r10,FPSCR
BIC     r10,r10,#0x00370000
ORR     r10,r10,#0x00130000    ; set length = 4, stride = 2
VMSR    FPSCR,r10
VFPASSERT VECTOR<4:2>      ; assert vector mode, length 4, stride 2
fadds  s8<4>, s16<4>, s0   ; ERROR, wrong stride because omitting
                           ; stride causes a default stride of 1.
fabss  s16<4:2>, s28<4:2>  ; okay
fadds  s8<>, s16<>, s2     ; okay (s8 and s16 both have
                           ; length 4 and stride 2. s2 is scalar.)
```

### Related references

*8.24 Vector notation* on page 8-189.
*8.25 VFPASSERT SCALAR* on page 8-190.

### Related information

*Procedure Call Standard for the ARM Architecture.*

# Chapter 9
# Assembler Command-line Options

Describes the command-line options supported by the ARM assembler, `armasm`.

It contains the following sections:

## 9.1 --16

Instructs the assembler to interpret instructions as Thumb instructions using the pre-UAL Thumb syntax.

This option is equivalent to a `CODE16` directive at the head of the source file. Use the `--thumb` option to specify Thumb instructions using the UAL syntax.

### Related references

*9.63 --thumb* on page 9-260.
*12.11 CODE16* on page 12-589.

## 9.2 --32

A synonym for the `--arm` command-line option.

### Related references

*9.4 --arm* on page 9-199.

## 9.3 --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`
: `/interwork` specifies that the code in the input file can interwork between ARM and Thumb safely. The default is `/nointerwork`.

`/inter, /nointer`
: Are synonyms for `/interwork` and `/nointerwork`.

`/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+vfp...`), then `/hardfp` gives an error.

### 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 --apcs=/inter/ropi inputfile.s
```

### Related information

*--apcs=qualifier...qualifier compiler option.*
*Procedure Call Standard for the ARM Architecture.*
*Application Binary Interface (ABI) for the ARM Architecture.*

## 9.4 --arm

Targets the ARM instruction set. The assembler is permitted to generate both ARM and Thumb code, but recognizes that ARM code is preferred.

This option instructs the assembler to interpret instructions as ARM instructions. It does not, however, guarantee ARM-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.

### Related references

*9.5 --arm_only* on page 9-200.
*12.7 ARM or CODE32* on page 12-585.

## 9.5 --arm_only

Enforces ARM-only code. The assembler behaves as if Thumb is absent from the target architecture.

This option instructs the assembler to only generate ARM code. This is similar to `--arm` but also has the property that the assembler does not permit the generation of any Thumb code.

### Related references

*9.4 --arm* on page 9-199.

## 9.6 --bi

A synonym for the `--bigend` command-line option.

### Related references

*9.7 --bigend* on page 9-202.
*9.45 --littleend* on page 9-242.

## 9.7 --bigend

Generates code suitable for an ARM processor using big-endian memory access.

The default is `--littleend`.

### Related references

*9.45 --littleend* on page 9-242.

## 9.8 --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

*9.18 --diag_error=tag[,tag,...]* on page 9-214.
*9.22 --diag_warning=tag[,tag,...]* on page 9-218.

## 9.9 --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**

This option is deprecated. Use `--diag_warning 1206` instead.

### Related references

*9.22 --diag_warning=tag[,tag,...]* on page 9-218.

## 9.10 --compatible=name

Generates code that is compatible with multiple target processors.

### Syntax

```
--compatible=name
```

Where:

*name*
is the name of a target processor or `None`.

Processor names are not case-sensitive.

Specifying `None` generates code only for the processor specified by `--cpu`.

If multiple instances of this option are present on the command line, the last one specified overrides the previous instances. Specify `--compatible=None` at the end of the command line to turn off all other instances of the option.

### Default

The default is `None`.

### Usage

Using this option avoids having to reassemble the same source code for different targets.

See the following table. The valid combinations are:

- `--cpu=CPU_from_group1 --compatible=CPU_from_group2`.
- `--cpu=CPU_from_group2 --compatible=CPU_from_group1`.

**Table 9-1 Compatible processor or architecture combinations**

| | |
|---|---|
| Group 1 | `ARM7TDMI` |
| Group 2 | `Cortex-M0`, `Cortex-M1`, `Cortex-M3`, `Cortex-M4`, `7-M`, `6-M`, `6S-M`, `SC300`, `SC000` |

No other combinations are permitted.

The effect is to generate code that is compatible with both `--cpu` and `--compatible`. This means that only 16-bit Thumb instructions are used. (This is the intersection of the capabilities of group 1 and group 2.)

**Note**

Although the generated code is compatible with multiple targets, this code might be less efficient than compiling for a single target processor or architecture.

### Example

To generate code that is compatible with both the ARM7TDMI processor and the Cortex-M4 processor, specify:

```
armasm --cpu=arm7tdmi --compatible=cortex-m4 inputfile.s
```

### Related references

*9.14 --cpu=name* on page 9-209.

## 9.11 --cpreproc

Instructs the assembler to call `armcc` to preprocess the input file before assembling it.

### Related concepts

*6.15 Using the C preprocessor* on page 6-126.

### Related references

*9.12 --cpreproc_opts=option[,option,...]* on page 9-207.

## 9.12 --cpreproc_opts=option[,option,…]

Enables the assembler to pass options to the compiler 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.

### Example

```
armasm --cpreproc --cpreproc_opts=-DDEBUG=1,-UALPHA inputfile.s
```

### Related concepts

*6.15 Using the C preprocessor* on page 6-126.

### Related references

*9.11 --cpreproc* on page 9-206.

## 9.13 --cpu=list

Lists the architecture and processor names that are supported by the `--cpu=name` option.

### Syntax

```
--cpu=list
```

### Related references

*9.14 --cpu=name* on page 9-209.

## 9.14 --cpu=name

Enables code generation for the selected ARM processor

### Syntax

`--cpu=`*name*

Where *name* is the name of a processor. Enter *name* as shown on ARM data sheets, for example, `Cortex-M3`.

Processor names are not case-sensitive.

### Default

`armasm` assumes `--cpu=ARM7TDMI` if you do not specify a `--cpu` option.

### Usage

The following general points apply to processor 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.

**FPU**

- Some specifications of `--cpu` imply an `--fpu` selection.
  For example, when building with the `--arm` option, `--cpu=Cortex-R4F` implies `--fpu=vfpv3_d16`.

  ——— **Note** ———

  Any explicit FPU, set with `--fpu` on the command line, overrides an implicit FPU.

  ———————————

- If no `--fpu` option is specified and no `--cpu` option is specified, `--fpu=softvfp` is used.

**ARM/Thumb**

- Specifying a processor or architecture that supports Thumb instructions, such as `--cpu=ARM7TDMI`, does not make the assembler generate Thumb code. It only enables features of the processor to be used, such as long multiply. Use the `--thumb` option to generate Thumb code, unless the processor is a Thumb-only processor, for example Cortex-M4. In this case, `--thumb` is not required.

  ——— **Note** ———

  Specifying the target processor or architecture might make the generated object code incompatible with other ARM processors. For example, code generated for architecture ARMv6 might not run on an ARM920T processor, if the generated object code includes instructions specific to ARMv6. Therefore, you must choose the lowest common denominator processor suited to your purpose.

  ———————————

- If you are building for mixed ARM/Thumb systems for processors that support ARMv4T or ARMv5T, then you must specify the interworking option `--apcs=/interwork`. By default, this is enabled for processors that support ARMv5T or above.
- If you build for Thumb, that is with the `--thumb` option on the command line, the assembler generates as much of the code as possible using the Thumb instruction set. However, the assembler might generate ARM code for some parts of the compilation. For example, if you are generating code for a 16-bit Thumb processor and using VFP, any function containing floating-point operations is compiled for ARM.

### Restrictions

You cannot specify both a processor and an architecture on the same command-line.

### Example

```
armasm --cpu=Cortex-A17 inputfile.s
```

### Related references

*9.3 --apcs=qualifier...qualifier* on page 9-197.
*9.10 --compatible=name* on page 9-205.
*9.13 --cpu=list* on page 9-208.
*9.33 --fpu=name* on page 9-229.
*9.63 --thumb* on page 9-260.
*9.66 --unsafe* on page 9-263.

### Related information

*ARM Architecture Reference Manual.*

## 9.15 --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

*9.24 --dwarf2* on page 9-220.
*9.25 --dwarf3* on page 9-221.
*9.37 --keep* on page 9-234.
*9.34 -g* on page 9-231.

## 9.16 --depend=dependfile

Writes makefile dependency lines to a file.

Source file dependency lists are suitable for use with make utilities.

### Related references

*9.48 --md* on page 9-245.
*9.17 --depend_format=string* on page 9-213.

## 9.17 --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

*9.16 --depend=dependfile* on page 9-212.

## 9.18 --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 9-2 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

*9.8 --brief_diagnostics, --no_brief_diagnostics* on page 9-203.
*9.19 --diag_remark=tag[,tag,...]* on page 9-215.
*9.21 --diag_suppress=tag[,tag,...]* on page 9-217.
*9.22 --diag_warning=tag[,tag,...]* on page 9-218.

## 9.19 --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

*9.8 --brief_diagnostics, --no_brief_diagnostics* on page 9-203.
*9.18 --diag_error=tag[,tag,...]* on page 9-214.
*9.21 --diag_suppress=tag[,tag,...]* on page 9-217.
*9.22 --diag_warning=tag[,tag,...]* on page 9-218.

## 9.20 --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 ARM compiler style.

**`ide`**
: Include the line number and character count for any line that is in error. These values are displayed in parentheses.

### Usage

`--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`.

### Default

The default is `--diag_style=arm`.

### Related references

*9.8 --brief_diagnostics, --no_brief_diagnostics* on page 9-203.
*9.18 --diag_error=tag[,tag,...]* on page 9-214.
*9.19 --diag_remark=tag[,tag,...]* on page 9-215.
*9.21 --diag_suppress=tag[,tag,...]* on page 9-217.
*9.22 --diag_warning=tag[,tag,...]* on page 9-218.

## 9.21 --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 --diag_suppress=1293,187
```

You can specify the optional assembler prefix `A` before the tag number. For example:

```
armasm --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.

## 9.22 --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

*9.8 --brief_diagnostics, --no_brief_diagnostics* on page 9-203.
*9.18 --diag_error=tag[,tag,...]* on page 9-214.
*9.19 --diag_remark=tag[,tag,...]* on page 9-215.
*9.21 --diag_suppress=tag[,tag,...]* on page 9-217.

## 9.23 --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

*12.28 EXPORT or GLOBAL* on page 12-606.

## 9.24 --dwarf2

Uses DWARF 2 debug table format.

This option can be used with `--debug`, to instruct `armasm` to generate DWARF 2 debug tables.

### Related references

*9.15 --debug* on page 9-211.
*9.25 --dwarf3* on page 9-221.

## 9.25 --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

*9.15 --debug* on page 9-211.
*9.24 --dwarf2* on page 9-220.

## 9.26 --errors=errorfile

Redirects the output of diagnostic messages from stderr to the specified errors file.

## 9.27 --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 9-3 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

*12.6 AREA* on page 12-582.

## 9.28 --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 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.

### Related references

*12.6 AREA* on page 12-582.

### Related information

*Execute-only memory.*

*Building applications for execute-only memory.*

## 9.29 --exceptions, --no_exceptions

Enables or disables exception handling.

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

*9.30 --exceptions_unwind, --no_exceptions_unwind* on page 9-226.
*12.40 FRAME UNWIND ON* on page 12-619.
*12.41 FRAME UNWIND OFF* on page 12-620.
*12.42 FUNCTION or PROC* on page 12-621.
*12.25 ENDFUNC or ENDP* on page 12-603.

## 9.30 --exceptions_unwind, --no_exceptions_unwind

Enables or disables function unwinding for exception-aware code. This option is only effective if `--exceptions` is enabled.

The default is `--exceptions_unwind`.

For finer control, use the `FRAME UNWIND ON` and `FRAME UNWIND OFF` directives.

### Related references

*9.29 --exceptions, --no_exceptions* on page 9-225.
*12.40 FRAME UNWIND ON* on page 12-619.
*12.41 FRAME UNWIND OFF* on page 12-620.
*12.42 FUNCTION or PROC* on page 12-621.
*12.25 ENDFUNC or ENDP* on page 12-603.

## 9.31 --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 --fpmode ieee_full inputfile.s
```

### Related references

*9.33 --fpu=name* on page 9-229.

### Related information

*IEEE Standards Association.*

## 9.32 --fpu=list

Lists the FPU architecture names that are supported by the `--fpu=name` option.

### Example

```
armasm --fpu=list
```

### Related references

*9.31 --fpmode=model* on page 9-227.
*9.33 --fpu=name* on page 9-229.

## 9.33 --fpu=name

Specifies the target FPU architecture.

### Syntax

`--fpu=`*`name`*

Where *`name`* is one of:

`none`
: Selects no floating-point option. No floating-point code is to be used. This produces an error if your code contains floating-point instructions.

`vfpv2`
: Selects a hardware floating-point unit conforming to architecture VFPv2.

`vfpv3`
: Selects a hardware vector floating-point unit conforming to architecture VFPv3. VFPv3 is backwards compatible with VFPv2 except that VFPv3 cannot trap floating-point exceptions.

`vfpv3_fp16`
: Selects a hardware vector floating-point unit conforming to architecture VFPv3 that also provides the half-precision extensions.

`vfpv3_d16`
: Selects a hardware vector floating-point unit conforming to VFPv3-D16 architecture.

`vfpv3_d16_fp16`
: Selects a hardware vector floating-point unit conforming to VFPv3-D16 architecture, that also provides the half-precision extensions.

`vfpv4`
: Selects a hardware floating-point unit conforming to the VFPv4 architecture.

`vfpv4_d16`
: Selects a hardware floating-point unit conforming to the VFPv4-D16 architecture.

`fpv4-sp`
: Selects a hardware floating-point unit conforming to the single precision variant of the FPv4 architecture.

`fpv5_d16`
: Selects a hardware floating-point unit conforming to the FPv5-D16 architecture.

`fpv5-sp`
: Selects a hardware floating-point unit conforming to the single precision variant of the FPv5 architecture.

`softvfp`
: Selects software floating-point support where floating-point operations are performed by a floating-point library, `fplib`. This is the default if you do not specify a `--fpu` option, or if you select a CPU that does not have an FPU.

`softvfp+vfpv2`
: Selects a hardware floating-point unit conforming to VFPv2, with software floating-point linkage. Select this option if you are interworking Thumb code with ARM code on a system that implements a VFP unit.

`softvfp+vfpv3`
: Selects a hardware vector floating-point unit conforming to VFPv3, with software floating-point linkage.

`softvfp+vfpv3_fp16`
: Selects a hardware vector floating-point unit conforming to VFPv3-fp16, with software floating-point linkage.

`softvfp+vfpv3_d16`
: Selects a hardware vector floating-point unit conforming to VFPv3-D16, with software floating-point linkage.

`softvfp+vfpv3_d16_fp16`
: Selects a hardware vector floating-point unit conforming to VFPv3-D16-fp16, with software floating-point linkage.

`softvfp+vfpv4`
: Selects a hardware floating-point unit conforming to FPv4, with software floating-point linkage.

`softvfp+vfpv4_d16`
: Selects a hardware floating-point unit conforming to VFPv4-D16, with software floating-point linkage.

`softvfp+fpv4-sp`
: Selects a hardware floating-point unit conforming to FPv4-SP, with software floating-point linkage.

`softvfp+fpv5_d16`
: Selects a hardware floating-point unit conforming to FPv5-D16, with software floating-point linkage.

`softvfp+fpv5-sp`
: Selects a hardware floating-point unit conforming to FPv5-SP, with software floating-point linkage.

To obtain a full list of FPU architectures use the `--fpu=list` option.

### 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.

Any FPU explicitly selected using the `--fpu` option always overrides any FPU implicitly selected using the `--cpu` option. For example, the option `--cpu=ARM1136JF-S --fpu=softvfp` generates code that uses the software floating-point library `fplib`, even though the choice of CPU implies the use of architecture VFPv2.

`armasm` sets a build attribute corresponding to name in the object file. The linker determines compatibility between object files, and selection of libraries, accordingly.

To control floating-point linkage without affecting the choice of FPU, you can use `--apcs=/softfp` or `--apcs=/hardfp`.

### Restrictions

`armasm` only permits hardware VFP architectures, such as `--fpu=vfpv3` or `--fpu=softvfp+vfpv2`, to be specified when `MRRC` and `MCRR` instructions are supported in the processor instruction set. `MRRC` and `MCRR` instructions are not supported in 4, 4T, 5T and 6-M. Therefore, `armasm` does not allow the use of these CPU architectures with hardware VFP architectures.

Other than this, `armasm` does not check that `--cpu` and `--fpu` combinations are valid. Beyond the scope of the assembler, additional architectural constraints apply. For example, VFPv3 is not supported with architectures prior to ARMv7. Therefore, the combination of `--fpu` and `--cpu` options permitted by `armasm` does not necessarily translate to the actual device in use.

### Default

The default target FPU architecture is derived from use of the `--cpu` option.

If the CPU specified with `--cpu` has a VFP coprocessor, the default target FPU architecture is the VFP architecture for that CPU. For example, the option `--cpu ARM1136JF-S` implies the option `--fpu vfpv2`. If a VFP coprocessor is present, VFP instructions are generated.

### Related references

*9.31 --fpmode=model* on page 9-227.

## 9.34 -g

Enables the generation of debug tables.

This option is a synonym for `--debug`.

### Related references

*9.15 --debug* on page 9-211.

## 9.35 --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

*9.68 --version_number* on page 9-265.
*9.70 --vsn* on page 9-267.

## 9.36 -idir[,dir, …]

Adds directories to the source file include path.

Any directories added using this option have to be fully qualified.

### Related references

*12.44 GET or INCLUDE* on page 12-623.

## 9.37 --keep

Instructs the assembler to keep named local labels in the symbol table of the object file, for use by the debugger.

### Related references

*12.49 KEEP* on page 12-630.

## 9.38 --length=n

Sets the listing page length.

Length zero means an unpaged listing. The default is 66 lines.

### Related references

*9.43 --list=file* on page 9-240.

## 9.39 --li

A synonym for the `--littleend` command-line option.

### Related references

*9.45 --littleend* on page 9-242.
*9.7 --bigend* on page 9-202.

## 9.40 --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.

### Related information

*Building an application with microlib.*
*--library_type=lib compiler option.*

## 9.41 --liclinger=seconds

The time in seconds that a license is to remain checked out.

### Syntax

`--liclinger=`*`seconds`*

## 9.42 --licretry

If you are using floating licenses, `armasm` makes up to 10 attempts to obtain a license when invoked.

**Note**

This option is always enabled. `armasm` ignores this option if you specify it.

### Related information

*Toolchain environment variables.*
*ARM DS-5 License Management Guide.*

## 9.43 --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

*9.53 --no_terse* on page 9-250.
*9.71 --width=n* on page 9-268.
*9.38 --length=n* on page 9-235.
*9.72 --xref* on page 9-269.
*12.56 OPT* on page 12-639.

## 9.44 --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

*9.43 --list=file* on page 9-240.

## 9.45 --littleend

Generates code suitable for an ARM processor using little-endian memory access.

### Related references

*9.7 --bigend* on page 9-202.

## 9.46 -m

Instructs the assembler to write source file dependency lists to `stdout`.

### Related references

*9.48 --md* on page 9-245.

## 9.47 --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.

## 9.48 --md

Creates makefile dependency lists.

This option instructs the assembler to write source file dependency lists to *inputfile*.d.

### Related references

*9.46 -m* on page 9-243.

## 9.49 --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.

## 9.50 --no_esc

Instructs the assembler to ignore C-style escaped special characters, such as \n and \t.

## 9.51 --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

*12.28 EXPORT or GLOBAL* on page 12-606.
*12.46 IMPORT and EXTERN* on page 12-626.

## 9.52 --no_regs

Instructs `armasm` not to predefine register names.

**Note**

This option is deprecated. Use `--regnames=none` instead.

### Related references

*9.59 --regnames* on page 9-256.

## 9.53 --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

*9.43 --list=file* on page 9-240.

## 9.54 --no_warn

Turns off warning messages.

### Related references

*9.22 --diag_warning=tag[,tag,...]* on page 9-218.

## 9.55 -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.

## 9.56 --pd

A synonym for the `--predefine` command-line option.

### Related references

*9.57 --predefine "directive"* on page 9-254.

## 9.57 --predefine "directive"

Instructs `armasm` to pre-execute one of the `SETA`, `SETL`, or `SETS` directives.

You must enclose *`directive`* in quotes, for example:

```
armasm --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 concepts

*6.14 Conditional assembly* on page 6-125.

### Related references

*9.56 --pd* on page 9-253.
*12.43 GBLA, GBLL, and GBLS* on page 12-622.
*12.45 IF, ELSE, ENDIF, and ELIF* on page 12-624.
*12.63 SETA, SETL, and SETS* on page 12-647.

## 9.58 --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.

### Related information

*--reduce_paths, --no_reduce_paths compiler option.*

## 9.59 --regnames

Controls the predefinition of register names.

### 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

*9.52 --no_regs* on page 9-249.
*9.3 --apcs=qualifier...qualifier* on page 9-197.
*2.10 Predeclared core register names* on page 2-40.
*2.11 Predeclared extension register names* on page 2-41.
*2.12 Predeclared coprocessor names* on page 2-42.

## 9.60 --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 Thumb instruction.

## 9.61 --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

*9.69 --via=filename* on page 9-266.

## 9.62 --split_ldm

Instructs the assembler to fault `LDM` and `STM` instructions with a large number of registers.

**Note**

This option is deprecated.

This option faults `LDM` instructions if the maximum number of registers transferred exceeds:

- Five, for `LDM`s that do not load the PC.
- Four, for `LDM`s that load the PC.

This option faults `STM` instructions if the maximum number of registers transferred exceeds 5.

Avoiding large multiple register transfers can reduce interrupt latency on ARM systems that:

- Do not have a cache or a write buffer (for example, a cacheless ARM7TDMI).
- Use zero wait-state, 32-bit memory.

Also, avoiding large multiple register transfers:

- Always increases code size.
- Has no significant benefit for cached systems or processors with a write buffer.
- Has no benefit for systems without zero wait-state memory, or for systems with slow peripheral devices. Interrupt latency in such systems is determined by the number of cycles required for the slowest memory or peripheral access. This is typically much greater than the latency introduced by multiple register transfers.

### Related references

*10.41 LDM* on page 10-342.

## 9.63 --thumb

Targets the Thumb instruction set.

This option instructs the assembler to interpret instructions as Thumb instructions, using the UAL syntax. This is equivalent to a `THUMB` directive at the start of the source file.

### Related references

*9.4 --arm* on page 9-199.
*12.65 THUMB* on page 12-649.

## 9.64 --thumbx

Targets the ThumbEE instruction set.

This option instructs the assembler to interpret instructions as ThumbEE instructions, using the UAL syntax. This is equivalent to a `THUMBX` directive at the start of the source file.

**Note**

- ARM deprecates the use of ThumbEE instructions.
- For descriptions of ThumbEE instructions, see the *ARM Architecture Reference Manual*.

### Related references

*12.66 THUMBX* on page 12-650.

### Related information

*ARM Architecture Reference Manual.*

## 9.65 --unaligned_access, --no_unaligned_access

Enables or disables unaligned accesses to data on ARM architecture-based processors.

These options instruct the assembler to set an attribute in the object file to enable or disable the use of unaligned accesses.

## 9.66 --unsafe

Enables instructions for other architectures to be assembled without error.

It downgrades error messages to corresponding warning messages. It also suppresses warnings about operator precedence.

### Related concepts

*7.20 Binary operators* on page 7-151.

### Related references

*9.18 --diag_error=tag[,tag,...]* on page 9-214.

*9.22 --diag_warning=tag[,tag,...]* on page 9-218.

## 9.67 --untyped_local_labels

Causes the assembler not to set the Thumb bit for the address of a numeric local label referenced in an `LDR` pseudo instruction.

When this option is not used, if you reference a numeric local label in an `LDR` pseudo-instruction, and the label is in Thumb code, then the assembler sets the Thumb 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 Thumb bit set, then use this option.

**Note**

When using this option, if you use the address in a branch (register) instruction, the assembler treats it as an ARM code address, causing the branch to arrive in ARM state, meaning it would interpret this code as ARM instructions.

### Example

```
    THUMB
    ...
1
    ...
    LDR r0,=%B1 ; r0 contains the address of numeric local label "1".
                ; Thumb bit is not set if --untyped_local_labels was
                ; used.
    ...
```

### Related concepts

*7.10 Numeric local labels* on page 7-141.

### Related references

*10.46 LDR pseudo-instruction* on page 10-356.

*10.16 B* on page 10-305.

## 9.68 --version_number

Displays the version of `armasm` you are using.

### Usage

The assembler displays the version number in the format `nnnbbbb`, where:

- `nnn` is the version number.
- `bbbb` is the build number.

### Example

Version 5.06 build 0019 is displayed as `5060019`.

## 9.69 --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

*13.1 Overview of via files* on page 13-654.

### Related references

*13.2 Via file syntax rules* on page 13-655.

## 9.70 --vsn

Displays the version information and the license details.

### Example

```
> armasm --vsn
Product: ARM Compiler N.nn
Component: ARM Compiler N.nn (toolchain_build_number)
Tool: armasm [build_number]
license_type
Software supplied by: ARM Limited
```

## 9.71 --width=n

Sets the listing page width.

The default is 79 characters.

### Related references

*9.43 --list=file* on page 9-240.

## 9.72 --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

*9.43 --list=file* on page 9-240.

# Chapter 10
# ARM and Thumb Instructions

Describes the ARM and Thumb instructions supported by the ARM assembler, `armasm`.

Some instruction descriptions have an Architectures section. Instructions that do not have this section are available in all versions of the ARM instruction set, and all versions of the Thumb instruction set.

It contains the following sections:

## 10.1 ARM and Thumb instruction summary

Different ARM architectures support different sets of ARM and Thumb instructions.

The following table gives a summary of the availability of ARM and Thumb instructions in different versions of the ARM architecture:

**Table 10-1 Summary of ARM and Thumb instructions**

| Mnemonic | Brief description | *Arch.* on page 10-280 |
|---|---|---|
| ADC | Add with Carry | All |
| ADD | Add | All |
| ADR | Load program or register-relative address (short range) | All |
| ADRL pseudo-instruction | Load program or register-relative address (medium range) | x6M |
| AND | Logical AND | All |
| ASR | Arithmetic Shift Right | All |
| B | Branch | All |
| BFC | Bit Field Clear | T2 |
| BFI | Bit Field Insert | T2 |
| BIC | Bit Clear | All |
| BKPT | Breakpoint | 5 |
| BL | Branch with Link | All |
| BLX | Branch with Link, change instruction set | T |
| BX | Branch, change instruction set | T |
| BXJ | Branch, change to Jazelle | J, x7M |
| CBZ, CBNZ | Compare and Branch if {Non}Zero | T2 |
| CDP | Coprocessor Data Processing operation | x6M |
| CDP2 | Coprocessor Data Processing operation | 5, x6M |
| CLREX | Clear Exclusive | K, x6M |
| CLZ | Count leading zeros | 5, x6M |
| CMN, CMP | Compare Negative, Compare | All |
| CPS | Change Processor State | 6 |
| CPY pseudo-instruction | Copy | 6 |
| DBG | Debug | 7 |
| DMB | Data Memory Barrier | 7, 6M |
| DSB | Data Synchronization Barrier | 7, 6M |
| EOR | Exclusive OR | All |
| ERET | Exception Return | 7VE |
| HVC | Hypervisor Call | 7VE |
| ISB | Instruction Synchronization Barrier | 7, 6M |

**Table 10-1 Summary of ARM and Thumb instructions (continued)**

| Mnemonic | Brief description | *Arch.* on page 10-280 |
|---|---|---|
| IT | If-Then | T2 |
| LDC | Load Coprocessor | x6M |
| LDC2 | Load Coprocessor | 5, x6M |
| LDM | Load Multiple registers | All |
| LDR | Load Register with word | All |
| LDR pseudo-instruction | Load Register pseudo-instruction | All |
| LDRB | Load Register with byte | All |
| LDRBT | Load Register with byte, user mode | x6M |
| LDRD | Load Registers with two words | 5E, x6M |
| LDREX | Load Register Exclusive | 6, x6M |
| LDREXB, LDREXH | Load Register Exclusive Byte, Halfword | K, x6M |
| LDREXD | Load Register Exclusive Doubleword | K, x7M |
| LDRH | Load Register with halfword | All |
| LDRHT | Load Register with halfword, user mode | T2 |
| LDRSB | Load Register with signed byte | All |
| LDRSBT | Load Register with signed byte, user mode | T2 |
| LDRSH | Load Register with signed halfword | All |
| LDRSHT | Load Register with signed halfword, user mode | T2 |
| LDRT | Load Register with word, user mode | x6M |
| LSL | Logical Shift Left | All |
| LSR | Logical Shift Right | All |
| MCR | Move from Register to Coprocessor | x6M |
| MCR2 | Move from Register to Coprocessor | 5, x6M |
| MCRR | Move from Registers to Coprocessor | 5E, x6M |
| MCRR2 | Move from Registers to Coprocessor | 6, x6M |
| MLA | Multiply Accumulate | x6M |
| MLS | Multiply and Subtract | T2 |
| MOV | Move | All |
| MOVT | Move Top | T2 |
| MOV32 pseudo-instruction | Move 32-bit immediate to register | T2 |
| MRC | Move from Coprocessor to Register | x6M |
| MRC2 | Move from Coprocessor to Register | 5, x6M |
| MRRC | Move from Coprocessor to Registers | 5E, x6M |
| MRRC2 | Move from Coprocessor to Registers | 6, x6M |
| MRS | Move from PSR to register | All |

**Table 10-1 Summary of ARM and Thumb instructions (continued)**

| Mnemonic | Brief description | *Arch.* on page 10-280 |
|---|---|---|
| MRS | Move from system Coprocessor to Register | 7R |
| MSR | Move from register to PSR | All |
| MSR | Move from Register to system Coprocessor | 7R |
| MUL | Multiply | All |
| MVN | Move Not | All |
| NEG pseudo-instruction | Negate | All |
| NOP | No Operation | All |
| ORN | Logical OR NOT | T2 |
| ORR | Logical OR | All |
| PKHBT, PKHTB | Pack Halfwords | 6, 7EM |
| PLD | Preload Data | 5E, x6M |
| PLDW | Preload Data with intent to Write | 7MP |
| PLI | Preload Instruction | 7 |
| POP | POP registers from stack | All |
| PUSH | PUSH registers to stack | All |
| QADD | Signed saturating Add | 5E, 7EM |
| QADD8 | Signed saturating parallel byte-wise addition | 6, 7EM |
| QADD16 | Signed saturating parallel halfword-wise addition | 6, 7EM |
| QASX | Signed saturating parallel add and subtract halfwords with exchange | 6, 7EM |
| QDADD | Signed saturating Double and Add | 5E, 7EM |
| QDSUB | Signed saturating Double and Subtract | 5E, 7EM |
| QSAX | Signed saturating parallel subtract and add halfwords with exchange | 6, 7EM |
| QSUB | Signed saturating Subtract | 5E, 7EM |
| QSUB8 | Signed saturating parallel byte-wise subtraction | 6, 7EM |
| QSUB16 | Signed saturating parallel halfword-wise subtraction | 6, 7EM |
| RBIT | Reverse Bits | T2 |
| REV | Reverse byte order in a word | 6 |
| REV16 | Reverse byte order in two halfwords | 6 |
| REVSH | Reverse byte order in a halfword and sign extend | 6 |
| RFE | Return From Exception | T2, x7M |
| ROR | Rotate Right Register | All |
| RRX | Rotate Right with Extend | x6M |
| RSB | Reverse Subtract | All |
| RSC | Reverse Subtract with Carry | x7M |
| SADD8 | Signed parallel byte-wise addition | 6, 7EM |

**Table 10-1 Summary of ARM and Thumb instructions (continued)**

| Mnemonic | Brief description | *Arch.* on page 10-280 |
|---|---|---|
| SADD16 | Signed parallel halfword-wise addition | 6, 7EM |
| SASX | Signed parallel add and subtract halfwords with exchange | 6, 7EM |
| SBC | Subtract with Carry | All |
| SBFX | Signed Bit Field eXtract | T2 |
| SDIV | Signed divide | 7M, 7R |
| SEL | Select bytes according to APSR GE flags | 6, 7EM |
| SETEND | Set Endianness for memory accesses | 6, x7M |
| SEV | Set Event | K, 6M |
| SHADD8 | Signed halving parallel byte-wise addition | 6, 7EM |
| SHADD16 | Signed halving parallel halfword-wise addition | 6, 7EM |
| SHASX | Signed halving parallel add and subtract halfwords with exchange | 6, 7EM |
| SHSAX | Signed halving parallel subtract and add halfwords with exchange | 6, 7EM |
| SHSUB8 | Signed halving parallel byte-wise subtraction | 6, 7EM |
| SHSUB16 | Signed halving parallel halfword-wise subtraction | 6, 7EM |
| SMC | Secure Monitor Call | Z |
| SMLAxy | Signed Multiply with Accumulate (32 <= 16 x 16 + 32) | 5E, 7EM |
| SMLAD | Dual Signed Multiply Accumulate | 6, 7EM |
| | (32 <= 32 + 16 x 16 + 16 x 16) | |
| SMLAL | Signed Multiply Accumulate (64 <= 64 + 32 x 32) | x6M |
| SMLALxy | Signed Multiply Accumulate (64 <= 64 + 16 x 16) | 5E, 7EM |
| SMLALD | Dual Signed Multiply Accumulate Long | 6, 7EM |
| | (64 <= 64 + 16 x 16 + 16 x 16) | |
| SMLAWy | Signed Multiply with Accumulate (32 <= 32 x 16 + 32) | 5E, 7EM |
| SMLSD | Dual Signed Multiply Subtract Accumulate | 6, 7EM |
| | (32 <= 32 + 16 x 16 – 16 x 16) | |
| SMLSLD | Dual Signed Multiply Subtract Accumulate Long | 6, 7EM |
| | (64 <= 64 + 16 x 16 – 16 x 16) | |
| SMMLA | Signed top word Multiply with Accumulate (32 <= TopWord(32 x 32 + 32)) | 6, 7EM |
| SMMLS | Signed top word Multiply with Subtract (32 <= TopWord(32 - 32 x 32)) | 6, 7EM |
| SMMUL | Signed top word Multiply (32 <= TopWord(32 x 32)) | 6, 7EM |
| SMUAD, SMUSD | Dual Signed Multiply, and Add or Subtract products | 6, 7EM |
| SMULxy | Signed Multiply (32 <= 16 x 16) | 5E, 7EM |
| SMULL | Signed Multiply (64 <= 32 x 32) | x6M |

**Table 10-1 Summary of ARM and Thumb instructions (continued)**

| Mnemonic | Brief description | *Arch.* on page 10-280 |
|---|---|---|
| SMULWy | Signed Multiply (32 <= 32 x 16) | 5E, 7EM |
| SRS | Store Return State | T2, x7M |
| SSAT | Signed Saturate | 6, x6M |
| SSAT16 | Signed Saturate, parallel halfwords | 6, 7EM |
| SSAX | Signed parallel subtract and add halfwords with exchange | 6, 7EM |
| SSUB8 | Signed parallel byte-wise subtraction | 6, 7EM |
| SSUB16 | Signed parallel halfword-wise subtraction | 6, 7EM |
| STC | Store Coprocessor | x6M |
| STC2 | Store Coprocessor | 5, x6M |
| STM | Store Multiple registers | All |
| STR | Store Register with word | All |
| STRB | Store Register with byte | All |
| STRBT | Store Register with byte, user mode | x6M |
| STRD | Store Registers with two words | 5E, x6M |
| STREX | Store Register Exclusive | 6, x6M |
| STREXB, STREXH | Store Register Exclusive Byte, Halfword | K, x6M |
| STREXD | Store Register Exclusive Doubleword | K, x7M |
| STRH | Store Register with halfword | All |
| STRHT | Store Register with halfword, user mode | T2 |
| STRT | Store Register with word, user mode | x6M |
| SUB | Subtract | All |
| SUBS pc, lr | Exception return, no stack | T2, x7M |
| SVC (formerly SWI) | SuperVisor Call | All |
| SWP, SWPB | Swap registers and memory (ARM only) | All, x7M |
| SXTAB | Sign extend Byte, with Addition | 6, 7EM |
| SXTAB16 | Sign extend two Bytes, with Addition | 6, 7EM |
| SXTAH | Sign extend Halfword, with Addition | 6, 7EM |
| SXTB | Sign extend Byte | 6 |
| SXTH | Sign extend Halfword | 6 |
| SXTB16 | Sign extend two Bytes | 6, 7EM |
| SYS | Execute system coprocessor instruction | 7R |
| TBB, TBH | Table Branch Byte, Halfword | T2 |
| TEQ | Test Equivalence | x6M |
| TST | Test | All |
| UADD8 | Unsigned parallel byte-wise addition | 6, 7EM |

**Table 10-1 Summary of ARM and Thumb instructions (continued)**

| Mnemonic | Brief description | *Arch.* on page 10-280 |
|---|---|---|
| UADD16 | Unsigned parallel halfword-wise addition | |
| UASX | Unsigned parallel add and subtract halfwords with exchange | |
| UBFX | Unsigned Bit Field eXtract | T2 |
| UDIV | Unsigned divide | 7M, 7R |
| UHADD8 | Unsigned halving parallel byte-wise addition | 6, 7EM |
| UHADD16 | Unsigned halving parallel halfword-wise addition | 6, 7EM |
| UHASX | Unsigned halving parallel add and subtract halfwords with exchange | 6, 7EM |
| UHSAX | Unsigned halving parallel subtract and add halfwords with exchange | 6, 7EM |
| UHSUB8 | Unsigned halving parallel byte-wise subtraction | 6, 7EM |
| UHSUB16 | Unsigned halving parallel halfword-wise subtraction | 6, 7EM |
| UMAAL | Unsigned Multiply Accumulate Accumulate Long | 6, 7EM |
| | (64 <= 32 + 32 + 32 x 32) | |
| UMLAL | Unsigned Multiply Accumulate | x6M |
| | (64 <= 32 x 32 + 64), (64 <= 32 x 32) | |
| UMULL | Unsigned Multiply | x6M |
| | (64 <= 32 x 32 + 64), (64 <= 32 x 32) | |
| UQADD8 | Unsigned saturating parallel byte-wise addition | 6, 7EM |
| UQADD16 | Unsigned saturating parallel halfword-wise addition | 6, 7EM |
| UQASX | Unsigned saturating parallel add and subtract halfwords with exchange | 6, 7EM |
| UQSAX | Unsigned saturating parallel subtract and add halfwords with exchange | 6, 7EM |
| UQSUB8 | Unsigned saturating parallel byte-wise subtraction | 6, 7EM |
| UQSUB16 | Unsigned saturating parallel halfword-wise subtraction | 6, 7EM |
| USAD8 | Unsigned Sum of Absolute Differences | 6, 7EM |
| USADA8 | Accumulate Unsigned Sum of Absolute Differences | 6, 7EM |
| USAT | Unsigned Saturate | 6, x6M |
| USAT16 | Unsigned Saturate, parallel halfwords | 6, 7EM |
| USAX | Unsigned parallel subtract and add halfwords with exchange | 6, 7EM |
| USUB8 | Unsigned parallel byte-wise subtraction | 6, 7EM |
| USUB16 | Unsigned parallel halfword-wise subtraction | 6, 7EM |
| UXTAB | Zero extend Byte with Addition | 6, 7EM |
| UXTAB16 | Zero extend two bytes with Addition | 6, 7EM |
| UXTAH | Zero extend Halfword with Addition | 6, 7EM |

**Table 10-1 Summary of ARM and Thumb instructions (continued)**

| Mnemonic | Brief description | *Arch.* on page 10-280 |
|---|---|---|
| UXTB | Zero extend Byte | 6 |
| UXTH | Zero extend Halfword | 6 |
| UXTB16 | Zero extend two bytes | 6, 7EM |
| V* | VFP instructions | |
| WFE | Wait For Event | T2, 6M |
| WFI | Wait For Interrupt | T2, 6M |
| YIELD | Yield | T2, 6M |

Entries in the Architecture column indicate that the instructions are available as follows:

**All**
: All versions of the ARM architecture.

**5**
: The ARMv5T*, ARMv6*, and ARMv7 architectures.

**5E**
: The ARMv5TE, ARMv6*, and ARMv7 architectures.

**6**
: The ARMv6* and ARMv7 architectures.

**6M**
: The ARMv6-M and ARMv7 architectures.

**x6M**
: Not available in the ARMv6-M architecture.

**7**
: The ARMv7 architectures.

**7M**
: The ARMv7-M architecture, including ARMv7E-M implementations.

**x7M**
: Not available in the ARMv6-M or ARMv7-M architecture, or any ARMv7E-M implementation.

**7EM**
: ARMv7E-M implementations but not in the ARMv7-M or ARMv6-M architecture.

**7R**
: The ARMv7-R architecture.

**7VE**
: The ARMv7 architectures that implement the Virtualization Extensions.

**J**
: The ARMv5TEJ, ARMv6*, and ARMv7 architectures.

**K**
: The ARMv6K, and ARMv7 architectures.

**T**
: The ARMv4T, ARMv5T*, ARMv6*, and ARMv7 architectures.

**T2**
: The ARMv6T2 and above architectures.

**Z**
: If Security Extensions are implemented.

## 10.2 Instruction width specifiers

The instruction width specifiers `.W` and `.N` control the size of Thumb instruction encodings for ARMv6T2 or later.

In Thumb code (ARMv6T2 or later) 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 ARM code.

In Thumb 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 ARM 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
```

## 10.3 Flexible second operand (Operand2)

Many ARM and Thumb 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 either of the following:

- A constant.
- A register with optional shift.

### Related concepts

*10.6 Shift operations* on page 10-285.

### Related references

*10.4 Syntax of Operand2 as a constant* on page 10-283.
*10.5 Syntax of Operand2 as a register with optional shift* on page 10-284.

## 10.4 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 ARM 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 Thumb 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

*10.6 Shift operations* on page 10-285.

### Related references

*10.3 Flexible second operand (Operand2)* on page 10-282.
*10.5 Syntax of Operand2 as a register with optional shift* on page 10-284.

## 10.5 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

*10.6 Shift operations* on page 10-285.

### Related references

*10.3 Flexible second operand (Operand2)* on page 10-282.
*10.4 Syntax of Operand2 as a constant* on page 10-283.

## 10.6 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`*.

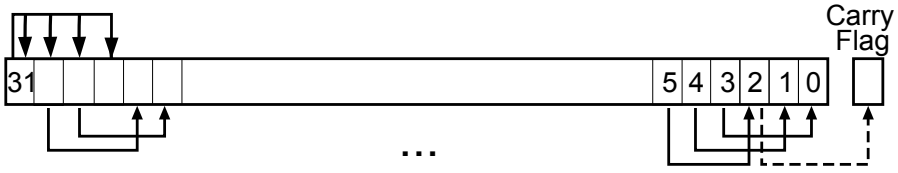


**Figure 10-1 ASR #3**

### 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.

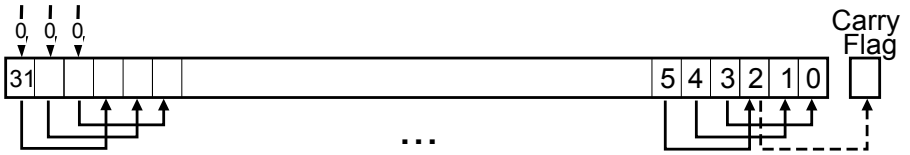


**Figure 10-2 LSR #3**

### 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.

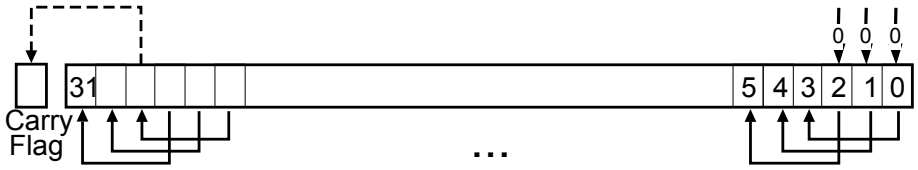


**Figure 10-3 LSL #3**

### 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.

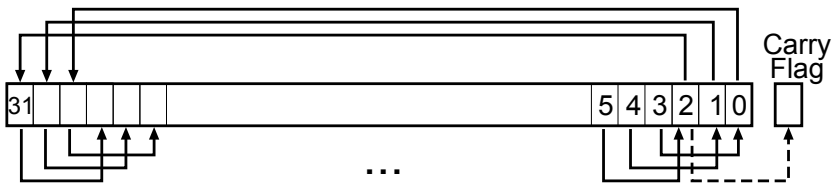


**Figure 10-4 ROR #3**

### 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`*.

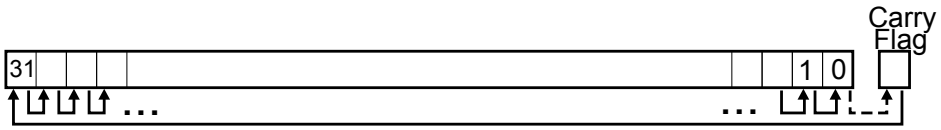


**Figure 10-5 RRX**

### Related references

*10.3 Flexible second operand (Operand2)* on page 10-282.

*10.4 Syntax of Operand2 as a constant* on page 10-283.

*10.5 Syntax of Operand2 as a register with optional shift* on page 10-284.

## 10.7 Saturating instructions

Some ARM and Thumb instructions perform saturating arithmetic.

The saturating instructions are:

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

Some of the parallel instructions are also saturating.

### Saturating arithmetic

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

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

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

**Note**

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

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

### Related references

*10.74 QADD* on page 10-398.
*10.81 QSUB* on page 10-405.
*10.78 QDADD* on page 10-402.
*10.79 QDSUB* on page 10-403.
*10.109 SMLAxy* on page 10-441.
*10.114 SMLAWy* on page 10-447.
*10.121 SMULxy* on page 10-454.
*10.123 SMULWy* on page 10-456.
*10.126 SSAT* on page 10-460.
*10.174 USAT* on page 10-521.
*10.63 MSR (general-purpose register to PSR)* on page 10-380.

## 10.8 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 10-2 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

*8.7 Conditional execution of VFP instructions* on page 8-170.

### Related references

*5.6 Comparison of condition code meanings in integer and floating-point code* on page 5-101.
*10.39 IT* on page 10-338.
*11.21 VMRS* on page 11-560.

## 10.9 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 Thumb 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 ARM 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 ARM instruction is deprecated.

——— **Note** ———

The deprecation of SP and PC in ARM instructions is only in ARMv6T2 and above.

———————————

### 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 Thumb 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

*10.3 Flexible second operand (Operand2)* on page 10-282.
*10.8 Condition code suffixes* on page 10-289.

## 10.10 ADD

Add without Carry.

### Syntax

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

`ADD{cond} {Rd}, Rn, #imm12 ; Thumb, 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 Thumb 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 Thumb `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 Thumb `ADD{cond} Rd, Rd, Rm` instructions, where both registers cannot be PC. However, the following 16-bit Thumb instructions are deprecated in ARMv6T2 and above:
  — `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 ARM 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 ARM instructions are deprecated.

**Note**

The deprecation of SP and PC in ARM instructions is only in ARMv6T2 and above.

### 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 Thumb 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.

`ADD Rd, Rd, Rm`
: ARMv6 and earlier: either *Rd* or *Rm*, or both, must be a Hi register. ARMv6T2 and above: this restriction does not apply.

`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

*10.3 Flexible second operand (Operand2)* on page 10-282.
*10.8 Condition code suffixes* on page 10-289.
*10.138 SUBS pc, lr* on page 10-481.

## 10.11 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 Thumb bit (bit 0) of the address if the target contains Thumb 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 10-3 PC-relative offsets**

| Instruction | Offset range | Architectures |
|---|---|---|
| ARM ADR | Any value that can be produced by rotating an 8-bit value right by any even number of bits within a 32-bit word. | All |
| Thumb ADR, 32-bit encoding | +/– 4095 | T2 |
| Thumb ADR, 16-bit encoding [b] | 0-1020 [c] | T |

### Notes about the Architectures column

Entries in the Architectures column indicate that the instructions are available as follows:

**All**
: All versions of the ARM architecture.

**T2**
: The ARMv6T2 and above architectures.

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

**T**

The ARMv4T, ARMv5T*, ARMv6*, and ARMv7 architectures.

### ADR in Thumb

You can use the `.W` width specifier to force `ADR` to generate a 32-bit instruction in Thumb 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 Thumb code, even if that results in failure for an address that could be generated in a 32-bit Thumb `ADD` instruction.

### Restrictions

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

In ARM code, `Rd` can be `PC` or `SP` but use of `SP` is deprecated in ARMv6T2 and above.

### Related concepts

*4.9 Load addresses to a register using ADR* on page 4-69.
*7.5 Register-relative and PC-relative expressions* on page 7-136.

### Related references

*10.4 Syntax of Operand2 as a constant* on page 10-283.
*10.13 ADRL pseudo-instruction* on page 10-299.
*12.6 AREA* on page 12-582.
*10.8 Condition code suffixes* on page 10-289.

## 10.12 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 Thumb 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 10-4 Register-relative offsets**

| Instruction | Offset range | Architectures |
|---|---|---|
| ARM `ADR` | Any value that can be produced by rotating an 8-bit value right by any even number of bits within a 32-bit word. | All |
| Thumb `ADR`, 32-bit encoding | +/– 4095 | T2 |
| Thumb `ADR`, 16-bit encoding, base register is SP [d] | 0-1020 [e] | T |

### Notes about the Architectures column

Entries in the Architectures column indicate that the instructions are available as follows:

**All**
: All versions of the ARM architecture.

**T2**
: The ARMv6T2 and above architectures.

[d] *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
[e] Must be a multiple of 4.

**T**

The ARMv4T, ARMv5T*, ARMv6*, and ARMv7 architectures.

### ADR in Thumb

You can use the `.W` width specifier to force `ADR` to generate a 32-bit instruction in Thumb 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 Thumb code, even if that results in failure for an address that could be generated in a 32-bit Thumb `ADD` instruction.

### Related concepts

*7.5 Register-relative and PC-relative expressions* on page 7-136.

### Related references

*10.4 Syntax of Operand2 as a constant* on page 10-283.
*10.13 ADRL pseudo-instruction* on page 10-299.
*12.53 MAP* on page 12-636.
*12.30 FIELD* on page 12-609.
*10.8 Condition code suffixes* on page 10-289.

## 10.13 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 Thumb bit (bit 0) of the address if the target contains Thumb instructions.

### Architectures and range

The available range depends on the instruction set in use:

**ARM**
: 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.

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

**Thumb, 16-bit encoding**
: `ADRL` is not available.

The given range is relative to a point four bytes (in Thumb code) or two words (in ARM code) after the address of the current instruction.

**Note**

When assembling Thumb instructions, `ADRL` is only available in ARMv6T2 and later.

### Related concepts

*7.5 Register-relative and PC-relative expressions* on page 7-136.

*4.3 Load immediate values* on page 4-60.

### Related references

*10.4 Syntax of Operand2 as a constant* on page 10-283.

*10.46 LDR pseudo-instruction* on page 10-356.
*12.6 AREA* on page 12-582.
*10.10 ADD* on page 10-292.
*10.8 Condition code suffixes* on page 10-289.

### Related information

*ARM Architecture Reference Manual.*

## 10.14 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 Thumb instructions

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

### Use of PC and SP in ARM instructions

You can use PC and SP with the `AND` ARM instruction but this is deprecated in ARMv6T2 and above.

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 Thumb 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

*10.3 Flexible second operand (Operand2)* on page 10-282.

*10.138 SUBS pc, lr* on page 10-481.

*10.8 Condition code suffixes* on page 10-289.

## 10.15 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 Thumb code

Thumb instructions must not use PC or SP.

### Use of SP and PC in ARM instructions

You can use SP in the `ASR` ARM instruction but this is deprecated in ARMv6T2 and above.

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 in ARMv6T2 and above.

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 ARM 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 Thumb 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

The `ASR` ARM instruction is available in all architectures.

The `ASR` 32-bit Thumb instruction is available in ARMv6T2 and above.

The `ASR` 16-bit Thumb instruction is available in ARMv4T and above.

### Example

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

### Related references

*10.55 MOV* on page 10-370.
*10.8 Condition code suffixes* on page 10-289.

## 10.16 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 Thumb.

*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 `B` instructions that are available in ARM and Thumb state. Instructions that are not shown in this table are not available. Notes in brackets show the first architecture version where the instruction is available.

**Table 10-5 B instruction availability and range**

| Instruction | ARM | | Thumb, 16-bit encoding | | Thumb, 32-bit encoding | |
|---|---|---|---|---|---|---|
| `B label` | ±32MB | (All) | ±2KB | (All T) | ±16MB [f] | (All T2) |
| `B{cond} label` | ±32MB | (All) | –252 to +258 | (All T) | ±1MB [f] | (All T2) |

### 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 Thumb

You can use the **.W** width specifier to force `B` to generate a 32-bit instruction in Thumb 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 Thumb code, even if that results in failure for a target that could be reached using a 32-bit Thumb instruction.

### Condition flags

The `B` instruction does not change the flags.

### Architectures

See the preceding table for details of availability of the `B` instruction in each architecture.

### Example

```
    B       loopA
```

[f] Use **.W** to instruct the assembler to use this 32-bit instruction.

### Related concepts

*7.5 Register-relative and PC-relative expressions* on page 7-136.

### Related references

*10.8 Condition code suffixes* on page 10-289.

### Related information

*Information about image structure and generation.*

## 10.17 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` ARM instruction but this is deprecated in ARMv6T2 and above. You cannot use SP in the `BFC` Thumb instruction.

### Condition flags

The `BFC` instruction does not change the flags.

### Architectures

This ARM instruction is available in ARMv6T2 and above.

This 32-bit Thumb instruction is available in ARMv6T2 and above.

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

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 10.18 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` ARM instruction but this is deprecated in ARMv6T2 and above. You cannot use SP in the `BFI` Thumb instruction.

### Condition flags

The `BFI` instruction does not change the flags.

### Architectures

This ARM instruction is available in ARMv6T2 and above.

This 32-bit Thumb instruction is available in ARMv6T2 and above.

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

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 10.19 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 Thumb instructions

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

### Use of PC and SP in ARM instructions

You can use PC and SP with the BIC instruction but they are deprecated in ARMv6T2 and above.

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 Thumb 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

*10.3 Flexible second operand (Operand2)* on page 10-282.
*10.138 SUBS pc, lr* on page 10-481.
*10.8 Condition code suffixes* on page 10-289.

## 10.20 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 ARM instruction.
- 0-255 (an 8-bit value) in a 16-bit Thumb 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 ARM state and Thumb 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 ARM code. In Thumb code, the `BKPT` instruction does not require a condition code suffix because `BKPT` always executes irrespective of its condition code suffix.

### Architectures

This ARM instruction is available in ARMv5T and above.

This 16-bit Thumb instruction is available in ARMv5T and above.

There is no 32-bit version of this instruction in Thumb.

## 10.21 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 Thumb.

*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 ARM and Thumb state. Instructions that are not shown in this table are not available. Notes in brackets show the first architecture version where the instruction is available.

**Table 10-6 BL instruction availability and range**

| Instruction | ARM | Thumb, 16-bit encoding | Thumb, 32-bit encoding |
|---|---|---|---|
| `BL label` | ±32MB (All) | ±4MB [g] (All T) | ±16MB (All T2) |
| `BL{cond} label` | ±32MB (All) | - | - |

### 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.

### Architectures

See the preceding table for details of availability of the `BL` instruction in each architecture.

### Examples

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

### Related concepts

*7.5 Register-relative and PC-relative expressions* on page 7-136.

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

### Related references

*10.8 Condition code suffixes* on page 10-289.

### Related information

*Information about image structure and generation.*

## 10.22 BLX

Branch with Link and exchange instruction set.

### Syntax

`BLX{cond}{.W} label`

`BLX{cond} Rm`

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 `BLX` instruction in Thumb.

*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 ARM state to Thumb state, or a processor in Thumb state to ARM 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, ARM state
  — if bit[0] of *Rm* is 1, the processor changes to, or remains in, Thumb state.

### Instruction availability and branch ranges

The following table shows the `BLX` instructions that are available in ARM and Thumb state. Instructions that are not shown in this table are not available. Notes in brackets show the first architecture version where the instruction is available.

**Table 10-7 BLX instruction availability and range**

| Instruction | ARM | | Thumb, 16-bit encoding | | Thumb, 32-bit encoding | |
|---|---|---|---|---|---|---|
| `BLX label` | ±32MB | (5) | ±4MB [h] | (5T) | ±16MB | (All T2 except ARMv7-M) |
| `BLX Rm` | Available | (5) | Available | (5T) | Use 16-bit | (All T2) |
| `BLX{cond} Rm` | Available | (5) | - | | - | - |

### Register restrictions

You can use PC for `Rm` in the ARM `BLX` instruction, but this is deprecated in ARMv6T2 and above. You cannot use PC in other ARM instructions.

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

You can use SP for `Rm` in this ARM instruction but this is deprecated in ARMv6T2 and above.

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

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

### Condition flags

This instruction does not change the flags.

### Architectures

See the preceding table for details of availability of the `BLX` instruction in each architecture.

### Related concepts

*7.5 Register-relative and PC-relative expressions* on page 7-136.

### Related references

*10.8 Condition code suffixes* on page 10-289.

### Related information

*Information about image structure and generation.*

## 10.23 BX

Branch and exchange instruction set.

### Syntax

`BX{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 `BX` instruction causes a branch to the address contained in *Rm* and exchanges the instruction set, if required:

- `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, ARM state.
  - If bit[0] of *Rm* is 1, the processor changes to, or remains in, Thumb state.

### Instruction availability and branch ranges

The following table shows the instructions that are available in ARM and Thumb state. Instructions that are not shown in this table are not available. Notes in brackets show the first architecture version where the instruction is available.

**Table 10-8 BX instruction availability and range**

| Instruction | ARM | Thumb, 16-bit encoding | Thumb, 32-bit encoding |
|---|---|---|---|
| `BX Rm` [i] | Available (4T, 5) | Available (All T) | Use 16-bit (All T2) |
| `BX{cond} Rm` [i] | Available (4T, 5) | - | - |

### Register restrictions

You can use PC for `Rm` in the ARM `BX` instruction, but this is deprecated in ARMv6T2 and above. You cannot use PC in other ARM instructions.

You can use PC for `Rm` in the Thumb `BX` instruction. You cannot use PC in other Thumb instructions.

You can use SP for `Rm` in the ARM `BX` instruction but this is deprecated in ARMv6T2 and above.

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

### Condition flags

The `BX` instruction does not change the flags.

### Architectures

See the preceding table for details of availability of the `BX` instruction in each architecture.

### Related concepts

*7.5 Register-relative and PC-relative expressions* on page 7-136.

[i] The assembler accepts `BX{cond} Rm` for code assembled for ARMv4 and converts it to `MOV{cond} PC, Rm` at link time, unless objects targeted for ARMv4T are present.

### Related references

*10.8 Condition code suffixes* on page 10-289.

### Related information

*Information about image structure and generation.*

## 10.24 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.

### Instruction availability and branch ranges

The following table shows the `BXJ` instructions that are available in ARM and Thumb state. Instructions that are not shown in this table are not available. Notes in brackets show the first architecture version where the instruction is available.

**Table 10-9 BXJ instruction availability and range**

| Instruction | ARM | Thumb, 16-bit encoding | Thumb, 32-bit encoding |
|---|---|---|---|
| `BXJ Rm` | Available (5J, 6) | - | Available (All T2 except ARMv7-M) |
| `BXJ{cond} Rm` | Available (5J, 6) | - | - - |

### Register restrictions

You can use SP for *Rm* in the `BXJ` ARM instruction but this is deprecated in ARMv6T2 and above.

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

### Condition flags

The `BXJ` instruction does not change the flags.

### Architectures

See the preceding table for details of availability of the `BXJ` instruction in each architecture.

### Related concepts

*7.5 Register-relative and PC-relative expressions* on page 7-136.

### Related references

*10.8 Condition code suffixes* on page 10-289.

### Related information

*Information about image structure and generation.*

## 10.25 CBZ and CBNZ

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

### Syntax

`CBZ Rn, label`

`CBNZ Rn, label`

where:

*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 within 4 to 130 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 Thumb instructions are available in ARMv6T2 and above.

There are no ARM or 32-bit Thumb encodings of these instructions.

## 10.26 CDP and CDP2

Coprocessor data operations.

### 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 ARM 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 to 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

The `CDP` ARM instruction is available in all versions of the ARM architecture.

The `CDP2` ARM instruction is available in ARMv5T and above.

These 32-bit Thumb instructions are available in ARMv6T2 and above.

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

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 10.27 CLREX

Clear Exclusive.

### Syntax

`CLREX{`*`cond`*`}`

where:

*`cond`*

is an optional condition code.

——— **Note** ———

*`cond`* is permitted only in Thumb code, using a preceding `IT` instruction. This is an unconditional instruction in ARM.

———————————

### 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 ARM instruction is available in ARMv6K and above.

This 32-bit Thumb instruction is available in ARMv7 and above.

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

### Related references

*10.8 Condition code suffixes* on page 10-289.

### Related information

*ARM Architecture Reference Manual.*

## 10.28 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 ARM instructions but this is deprecated in ARMv6T2 and above.

You cannot use SP in Thumb instructions.

### Condition flags

This instruction does not change the flags.

### Architectures

This ARM instruction is available in ARMv5T and above.

This 32-bit Thumb instruction is available in ARMv6T2 and above.

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

### Examples

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

Use the `CLZ` Thumb 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

*10.8 Condition code suffixes* on page 10-289.

## 10.29 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 ARM 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 ARM and Thumb 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 ARM instructions without register controlled shift but this is deprecated in ARMv6T2 and above.

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

You cannot use PC for any operand in these Thumb instructions.

### Use of SP in ARM and Thumb instructions

You can use SP for *Rn* in ARM and Thumb instructions.

You can use SP for *Rm* in ARM instructions but this is deprecated in ARMv6T2 and above.

You can use SP for *Rm* in a 16-bit Thumb `CMP Rn, Rm` instruction but this is deprecated in ARMv6T2 and above. Other uses of SP for *Rm* are not permitted in Thumb.

### 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 Thumb 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

*10.3 Flexible second operand (Operand2)* on page 10-282.
*10.8 Condition code suffixes* on page 10-289.

## 10.30 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 Thumb code, and are 16-bit instructions:

- `CPSIE `*`iflags`*.
- `CPSID `*`iflags`*.

You cannot specify a mode change in a 16-bit Thumb instruction.

### Architectures

This ARM instruction is available in ARMv6 and above.

This 32-bit Thumb instruction are available in ARMv6T2 and above.

This 16-bit Thumb instruction is available in T variants of ARMv6 and above.

### 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

*2.4 Processor modes, and privileged and unprivileged software execution* on page 2-33.

## 10.31 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 ARMv6 and above in ARM code and in T variants of ARMv6 and above in Thumb 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

*10.55 MOV* on page 10-370.

## 10.32 DBG

Debug.

### Syntax

`DBG{cond} {option}`

where:

*cond*
is an optional condition code.

*option*
is an optional limitation on the operation of the hint. The range is 0-15.

### Usage

`DBG` is a hint instruction. It is optional whether it is implemented or not. If it is not implemented, it behaves as a `NOP`. The assembler produces a diagnostic message if the instruction executes as `NOP` on the target.

`DBG` executes as a `NOP` instruction in ARMv6K and ARMv6T2.

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 ARM instruction is available in ARMv6K and above.

This 32-bit Thumb instruction is available in ARMv6T2 and above.

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

### Related references

*10.67 NOP* on page 10-387.
*10.8 Condition code suffixes* on page 10-289.

## 10.33 DMB

Data Memory Barrier.

### Syntax

`DMB{`*`cond`*`} {`*`option`*`}`

where:

*`cond`*
is an optional condition code.

**Note**

*`cond`* is permitted only in Thumb code. This is an unconditional instruction in ARM code.

*`option`*
is an optional limitation on the operation of the hint. Permitted values are:

**`SY`**
Full system `DMB` operation. This is the default and can be omitted.

**`ST`**
`DMB` operation that waits only for stores to complete.

**`ISH`**
`DMB` operation only to the inner shareable domain.

**`ISHST`**
`DMB` operation that waits only for stores to complete, and only to the inner shareable domain.

**`NSH`**
`DMB` operation only out to the point of unification.

**`NSHST`**
`DMB` operation that waits only for stores to complete and only out to the point of unification.

**`OSH`**
`DMB` operation only to the outer shareable domain.

**`OSHST`**
`DMB` operation that waits only for stores to complete, and only to the outer shareable domain.

### 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 ARM and 32-bit Thumb instruction is available in ARMv7.

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

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 10.34 DSB

Data Synchronization Barrier.

### Syntax

`DSB{`*`cond`*`} {`*`option`*`}`

where:

*`cond`*
is an optional condition code.

**Note**

*`cond`* is permitted only in Thumb code. This is an unconditional instruction in ARM.

*`option`*
is an optional limitation on the operation of the hint. Permitted values are:

**SY**
Full system `DSB` operation. This is the default and can be omitted.

**ST**
`DSB` operation that waits only for stores to complete.

**ISH**
`DSB` operation only to the inner shareable domain.

**ISHST**
`DSB` operation that waits only for stores to complete, and only to the inner shareable domain.

**NSH**
`DSB` operation only out to the point of unification.

**NSHST**
`DSB` operation that waits only for stores to complete and only out to the point of unification.

**OSH**
`DSB` operation only to the outer shareable domain.

**OSHST**
`DSB` operation that waits only for stores to complete, and only to the outer shareable domain.

### 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 ARM and 32-bit Thumb instruction is available in ARMv7.

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

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 10.35 EOR

Logical Exclusive OR.

### Syntax

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

where:

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

*cond*
is an optional condition code.

*Rd*
is the destination register.

*Rn*
is the register holding the first operand.

*Operand2*
is a flexible second operand.

### Operation

The EOR instruction performs bitwise Exclusive OR operations on the values in *Rn* and *Operand2*.

### Use of PC in Thumb instructions

You cannot use PC (R15) for *Rd* or any operand in an EOR instruction.

### Use of PC and SP in ARM instructions

You can use PC and SP with the EOR instruction but they are deprecated in ARMv6T2 and above.

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 Thumb 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

*10.3 Flexible second operand (Operand2)* on page 10-282.

*10.138 SUBS pc, lr* on page 10-481.

*10.8 Condition code suffixes* on page 10-289.

## 10.36 ERET

Exception Return.

### Syntax

`ERET{`*cond*`}`

where:

*cond*
is an optional condition code.

### Usage

In a processor that implements the Virtualization Extensions, you can use `ERET` to perform a return from an exception taken to Hyp mode.

### Operation

When executed in Hyp mode, `ERET` loads the PC from ELR_hyp and loads the CPSR from SPSR_hyp. When executed in any other mode, apart from User or System, it behaves as:

- `MOVS PC, LR` in the ARM instruction set.
- `SUBS PC, LR, #0` in the Thumb 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 Thumb instruction set.

### Architectures

This ARM instruction is available in ARMv7 architectures that include the Virtualization Extensions.

This 32-bit Thumb instruction is available in ARMv7 architectures that include the Virtualization Extensions.

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

### Related concepts

*2.4 Processor modes, and privileged and unprivileged software execution* on page 2-33.

### Related references

*10.55 MOV* on page 10-370.
*10.138 SUBS pc, lr* on page 10-481.
*10.8 Condition code suffixes* on page 10-289.
*10.37 HVC* on page 10-336.

## 10.37 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 ARM and 32-bit Thumb instruction is available in ARMv7 architectures that include the Virtualization Extensions.

There is no 16-bit version of this instruction in Thumb

### Related concepts

*2.4 Processor modes, and privileged and unprivileged software execution* on page 2-33.

### Related references

*10.36 ERET* on page 10-335.

## 10.38 ISB

Instruction Synchronization Barrier.

### Syntax

`ISB{cond} {option}`

where:

*cond*
is an optional condition code.

**Note**

*cond* is permitted only in Thumb code. This is an unconditional instruction in ARM code.

*option*
is an optional limitation on the operation of the hint. The permitted value is:

**SY**
Full system `DMB` 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 ARM and 32-bit Thumb instruction is available in ARMv7.

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

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 10.39 IT

If-Then.

### 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 switch for the second, third and fourth instruction 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` instruction makes up to four following instructions (the *IT block*) conditional. The conditions can be all the same, or some of them can be the logical inverse of the others.

The instructions (including branches) in the IT block, except the `BKPT` instruction, must specify the condition in the `{cond}` part of their 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 ARM 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.

**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**

The assembler shows a diagnostic message when any of these instructions are used in an IT block.

### 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.

### Architectures

This 16-bit Thumb instruction is available in ARMv6T2 and above.

In ARM code, IT is a pseudo-instruction that does not generate any code.

There is no 32-bit version of this instruction.

### Correct examples

```
ITTE   NE        ; IT can be omitted
ANDNE  r0,r0,r1  ; 16-bit AND, not ANDS
ADDSNE r2,r2,#1  ; 32-bit ADDS (16-bit ADDS does not set flags in
                 ; IT block)
MOVEQ  r2,r3     ; 16-bit MOV
ITT    AL        ; emit 2 non-flag setting 16-bit instructions
ADDAL  r0,r0,r1  ; 16-bit ADD, not ADDS
SUBAL  r2,r2,#1  ; 16-bit SUB, not SUB
ADD    r0,r0,r1  ; expands into 32-bit ADD, and is not in IT block
ITT    EQ
MOVEQ  r0,r1
BEQ    dloop     ; branch at end of IT block is permitted
ITT    EQ
MOVEQ  r0,r1
BKPT   #1        ; BKPT always executes
ADDEQ  r0,r0,#1
```

### Incorrect example

```
IT     NE
ADD    r0,r0,r1  ; syntax error: no condition code used in IT block
```

## 10.40 LDC and LDC2

Transfer Data from memory to Coprocessor.

### 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 ARM 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 in the range 0 to 15.

*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

LDC is available in all versions of the ARM architecture.

LDC2 is available in ARMv5T and above.

These 32-bit Thumb instructions are available in ARMv6T2 and above.

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

### 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

*7.5 Register-relative and PC-relative expressions* on page 7-136.
*6.16 Address alignment* on page 6-128.

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 10.41 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 (ARM only).

  **DA**
  : Decrement address After each transfer (ARM 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 ARM 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 ARM state, but there are some restrictions in Thumb state.

**^**
: is an optional suffix, available in ARM 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 Thumb instructions

In 32-bit Thumb 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 ARM instructions

ARM 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 in ARMv6T2 and above.

### 16-bit instructions

16-bit versions of a subset of these instructions are available in Thumb 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.

In ARMv4, bits[1:0] of the address loaded must be 0b00.

In ARMv5T and above:

- Bits[1:0] must not be 0b10.
- If bit[0] is 1, execution continues in Thumb state.
- If bit[0] is 0, execution continues in ARM state.

### Loading or storing the base register, with writeback

In ARM or 16-bit Thumb 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 in ARMv6T2 and above.
- Otherwise, the loaded or stored value of *Rn* cannot be relied on, so these instructions are not permitted.

32-bit Thumb 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

*4.15 Stack implementation using LDM and STM* on page 4-77.

*6.16 Address alignment* on page 6-128.

### Related references

*10.72 POP* on page 10-395.

*10.8 Condition code suffixes* on page 10-289.

## 10.42 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 10-10 Offsets and architectures, LDR, word, halfword, and byte**

| Instruction | Immediate offset | Pre-indexed | Post-indexed | Arch. |
|---|---|---|---|---|
| ARM, word or byte [j] | –4095 to 4095 | –4095 to 4095 | –4095 to 4095 | All |
| ARM, signed byte, halfword, or signed halfword | –255 to 255 | –255 to 255 | –255 to 255 | All |
| ARM, doubleword | –255 to 255 | –255 to 255 | –255 to 255 | 5E |

**Table 10-10 Offsets and architectures, LDR, word, halfword, and byte (continued)**

| Instruction | Immediate offset | Pre-indexed | Post-indexed | Arch. |
|---|---|---|---|---|
| Thumb 32-bit encoding, word, halfword, signed halfword, byte, or signed byte [j] | –255 to 4095 | –255 to 255 | –255 to 255 | T2 |
| Thumb 32-bit encoding, doubleword | –1020 to 1020 [k] | –1020 to 1020 [k] | –1020 to 1020 [k] | T2 |
| Thumb 16-bit encoding, word [l] | 0 to 124 [k] | Not available | Not available | T |
| Thumb 16-bit encoding, unsigned halfword [l] | 0 to 62 [m] | Not available | Not available | T |
| Thumb 16-bit encoding, unsigned byte [l] | 0 to 31 | Not available | Not available | T |
| Thumb 16-bit encoding, word, Rn is SP [n] | 0 to 1020 [k] | Not available | Not available | T |

### Notes about the Architectures column

Entries in the Architecture column indicate that the instructions are available as follows:

**All**
: All versions of the ARM architecture.

**5E**
: The ARMv5TE, ARMv6*, and ARMv7 architectures.

**T2**
: The ARMv6T2 and above architectures.

**T**
: The ARMv4T, ARMv5T*, ARMv6*, and ARMv7 architectures.

### Register restrictions

R*n* must be different from R*t* in the pre-index and post-index forms.

### Doubleword register restrictions

R*n* must be different from R*t2* in the pre-index and post-index forms.

For Thumb instructions, you must not specify SP or PC for either *Rt* or *Rt2*.

For ARM 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 ARM code you can use PC for R*t* in LDR word instructions and PC for R*n* in LDR instructions.

Other uses of PC are not permitted in these ARM instructions.

In Thumb code you can use PC for R*t* in LDR word instructions and PC for R*n* in LDR instructions. Other uses of PC in these Thumb instructions are not permitted.

### Use of SP

You can use SP for R*n*.

[j] 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 Thumb state, otherwise execution continues in ARM state.
[k] Must be divisible by 4.
[l] Rt and Rn must be in the range R0-R7.
[m] Must be divisible by 2.
[n] Rt must be in the range R0-R7.

In ARM code, you can use SP for `Rt` in word instructions. You can use SP for `Rt` in non-word instructions in ARM code but this is deprecated in ARMv6T2 and above.

In Thumb 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 Thumb 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

*6.16 Address alignment* on page 6-128.

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 10.43 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 ARM code but they are deprecated in ARMv6T2 and above.

### 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 10-11 PC-relative offsets**

| Instruction | Offset range | Architectures |
|---|---|---|
| ARM LDR, LDRB, LDRSB, LDRH, LDRSH [o] | +/– 4095 | All |
| ARM LDRD | +/– 255 | 5E |
| 32-bit Thumb LDR, LDRB, LDRSB, LDRH, LDRSH [o] | +/– 4095 | T2 |
| 32-bit Thumb LDRD [p] | +/– 1020 [q] | T2 |
| 16-bit Thumb LDR [r] | 0-1020 [q] | T |

### Notes about the Architectures column

Entries in the Architectures column indicate that the instructions are available as follows:

**All**
: All versions of the ARM architecture.

**5E**
: The ARMv5TE, ARMv6*, and ARMv7 architectures.

**T2**
: The ARMv6T2 and above architectures.

**T**
: The ARMv4T, ARMv5T*, ARMv6*, and ARMv7 architectures.

### LDR (PC-relative) in Thumb

You can use the `.W` width specifier to force `LDR` to generate a 32-bit instruction in Thumb 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 Thumb code, even if that results in failure for a target that could be reached using a 32-bit Thumb `LDR` instruction.

### Doubleword register restrictions

For 32-bit Thumb instructions, you must not specify SP or PC for either *`Rt`* or *`Rt2`*.

For ARM 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 ARM code, you can use SP for `Rt` in `LDR` word instructions. You can use SP for `Rt` in `LDR` non-word ARM instructions but this is deprecated in ARMv6T2 and above.

In Thumb code, you can use SP for `Rt` in `LDR` word instructions only. All other uses of SP in these instructions are not permitted in Thumb code.

### Related concepts

*7.5 Register-relative and PC-relative expressions* on page 7-136.
*6.16 Address alignment* on page 6-128.

[o] 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 Thumb state, otherwise execution continues in ARM state.
[p] In ARMv7-M, `LDRD` (PC-relative) instructions must be on a word-aligned address.
[q] Must be a multiple of 4.
[r] Rt must be in the range R0-R7. There are no byte, halfword, or doubleword 16-bit instructions.

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 10.44 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 ; ARM only
LDR{type}{cond} Rt, [Rn], ±Rm {, shift} ; post-indexed ; ARM only
LDRD{cond} Rt, Rt2, [Rn, ±Rm] ; register offset, doubleword ; ARM only
LDRD{cond} Rt, Rt2, [Rn, ±Rm]! ; pre-indexed, doubleword ; ARM only
LDRD{cond} Rt, Rt2, [Rn], ±Rm ; post-indexed, doubleword ; ARM 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 Thumb 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 10-12 Options and architectures, LDR (register offsets)**

| Instruction | +/–*Rm* [s] | shift | Arch. |
|---|---|---|---|
| ARM, word or byte [t] | +/–*Rm* | `LSL #0-31 LSR #1-32` | All |
| | | `ASR #1-32 ROR #1-31 RRX` | |
| ARM, signed byte, halfword, or signed halfword | +/–*Rm* | Not available | All |
| ARM, doubleword | +/–*Rm* | Not available | 5E |
| Thumb 32-bit encoding, word, halfword, signed halfword, byte, or signed byte [t] | +*Rm* | `LSL #0-3` | T2 |
| Thumb 16-bit encoding, all except doubleword [u] | +*Rm* | Not available | T |

### Notes about the Architecture column

Entries in the Architecture column indicate that the instructions are available as follows:

**All**
: All versions of the ARM architecture.

**5E**
: The ARMv5TE, ARMv6*, and ARMv7 architectures.

**T2**
: The ARMv6T2 and above architectures.

**T**
: The ARMv4T, ARMv5T*, ARMv6*, and ARMv7 architectures.

### Register restrictions

In the pre-index and post-index forms:

- `Rn` must be different from `Rt`.
- `Rn` must be different from `Rm` in architectures before ARMv6.

### Doubleword register restrictions

For ARM 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 ARM 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 ARM instructions.

In Thumb instructions you can use PC for `Rt` in `LDR` word instructions. Other uses of PC in these Thumb instructions are not permitted.

### Use of SP

You can use SP for `Rn`.

In ARM code, you can use SP for `Rt` in word instructions. You can use SP for `Rt` in non-word ARM instructions but this is deprecated in ARMv6T2 and above.

[s] Where +/–Rm is shown, you can use –Rm, +Rm, or Rm. Where +Rm is shown, you cannot use –Rm.
[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 Thumb state, otherwise execution continues in ARM state.
[u] Rt, Rn, and Rm must all be in the range R0-R7.

You can use SP for `Rm` in ARM instructions but this is deprecated in ARMv6T2 and above.

In Thumb 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 Thumb code.

Use of SP for `Rm` is not permitted in Thumb state.

### Related concepts

*6.16 Address alignment* on page 6-128.

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 10.45 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 10-13 Register-relative offsets**

| Instruction | Offset range | Architectures |
|---|---|---|
| ARM `LDR`, `LDRB` [v] | +/– 4095 | All |
| ARM `LDRSB`, `LDRH`, `LDRSH` | +/– 255 | All |
| ARM `LDRD` | +/– 255 | 5E |

[v] 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 Thumb state, otherwise execution continues in ARM state.
[w] Must be a multiple of 4.
[x] Rt and base register must be in the range R0-R7.
[y] Must be a multiple of 2.
[z] Rt must be in the range R0-R7.

**Table 10-13 Register-relative offsets (continued)**

| Instruction | Offset range | Architectures |
|---|---|---|
| Thumb, 32-bit LDR, LDRB, LDRSB, LDRH, LDRSH [v] | –255 to 4095 | T2 |
| Thumb, 32-bit LDRD | +/– 1020 [w] | T2 |
| Thumb, 16-bit LDR [x] | 0 to 124 [w] | T |
| Thumb, 16-bit LDRH [x] | 0 to 62 [y] | T |
| Thumb, 16-bit LDRB [x] | 0 to 31 | T |
| Thumb, 16-bit LDR, base register is SP [z] | 0 to 1020 [w] | T |

### Notes about the Architectures column

Entries in the Architectures column indicate that the instructions are available as follows:

**All**
All versions of the ARM architecture.

**5E**
The ARMv5TE, ARMv6*, and ARMv7 architectures.

**T2**
The ARMv6T2 and above architectures.

**T**
The ARMv4T, ARMv5T*, ARMv6*, and ARMv7 architectures.

### LDR (register-relative) in Thumb

You can use the `.W` width specifier to force `LDR` to generate a 32-bit instruction in Thumb 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 Thumb code, even if that results in failure for a target that could be reached using a 32-bit Thumb `LDR` instruction.

### Doubleword register restrictions

For 32-bit Thumb instructions, you must not specify SP or PC for either *`Rt`* or *`Rt2`*.

For ARM 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 ARM code, you can use SP for `Rt` in word instructions. You can use SP for `Rt` in non-word ARM instructions but this is deprecated in ARMv6T2 and above.

In Thumb 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 Thumb code.

### Related concepts

*7.5 Register-relative and PC-relative expressions* on page 7-136.
*6.16 Address alignment* on page 6-128.

### Related references

*12.30 FIELD* on page 12-609.
*12.53 MAP* on page 12-636.
*10.8 Condition code suffixes* on page 10-289.

## 10.46 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 Thumb code, the Thumb 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 ARM or 32-bit Thumb encoding) or in the range 0 to +1KB (16-bit Thumb encoding). You are responsible for ensuring that there is a literal pool within range.

If the label referenced is in Thumb code, the LDR pseudo-instruction sets the Thumb bit (bit 0) of *label_expr*.

**Note**

In *RealView Compilation Tools* (RVCT) v2.2, the Thumb 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 Thumb bit when referencing labels in Thumb code.

### LDR in Thumb code

You can use the `.W` width specifier to force LDR to generate a 32-bit instruction in Thumb code on ARMv6T2 and above processors. `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 Thumb 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

*7.3 Numeric constants* on page 7-134.

*7.5 Register-relative and PC-relative expressions* on page 7-136.

*7.10 Numeric local labels* on page 7-141.

### Related references

*9.67 --untyped_local_labels* on page 9-264.

*10.56 MOV32 pseudo-instruction* on page 10-372.

*10.8 Condition code suffixes* on page 10-289.

*12.51 LTORG* on page 12-632.

## 10.47 LDR, unprivileged

Unprivileged load byte, halfword, or word.

### Syntax

```
LDR{type}T{cond} Rt, [Rn {, #offset}] ; immediate offset (32-bit Thumb encoding only)
LDR{type}T{cond} Rt, [Rn] {, #offset} ; post-indexed (ARM only)
LDR{type}T{cond} Rt, [Rn], ±Rm {, shift} ; post-indexed (register) (ARM 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 10-14 Offsets and architectures, LDR (User mode)**

| Instruction | Immediate offset | Post-indexed | +/–*Rm* [aa] | shift | Arch. |
|---|---|---|---|---|---|
| ARM, word or byte | Not available | –4095 to 4095 | +/–*Rm* | LSL #0-31 | All |
| | | | | LSR #1-32 | |

[aa] You can use –Rm, +Rm, or Rm.

**Table 10-14 Offsets and architectures, LDR (User mode) (continued)**

| Instruction | Immediate offset | Post-indexed | +/–*Rm* [aa] | shift | Arch. |
|---|---|---|---|---|---|
| | | | | ASR #1-32 | |
| | | | | ROR #1-31 | |
| | | | | RRX | |
| ARM, signed byte, halfword, or signed halfword | Not available | –255 to 255 | +/–*Rm* | Not available | T2 |
| Thumb, 32-bit encoding, word, halfword, signed halfword, byte, or signed byte | 0 to 255 | Not available | Not available | | T2 |

### Notes on the Architecture column

Entries in the Architecture column indicate that the instructions are available as follows:

**All**
All versions of the ARM architecture.

**T2**
The ARMv6T2 and above architectures.

### Related concepts

*6.16 Address alignment* on page 6-128.

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 10.48 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 Thumb 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 ARM instructions:

- SP can be used but use of SP for *Rt* or *Rt2* is deprecated in ARMv6T2 and above.
- For LDREXD, *Rt* must be an even numbered register, and not LR.
- *Rt2* must be *R(t+1)*.
- *offset* is not permitted.

For Thumb instructions:

- SP can be used for R*n*, but must not be used for *Rt* or *Rt2*.
- For LDREXD, *Rt* and *Rt2* must not be the same register.
- The value of *offset* can be any multiple of four in the range 0-1020.

### Usage

Use LDREX and STREX to implement interprocess communication in multiple-processor and shared-memory systems.

For reasons of performance, keep the number of instructions between corresponding LDREX and STREX instructions to a minimum.

**Note**

The address used in a STREX instruction must be the same as the address in the most recently executed LDREX instruction.

### Architectures

ARM LDREX and STREX are available in ARMv6 and above.

ARM LDREXB, LDREXH, LDREXD, STREXB, STREXD, and STREXH are available in ARMv6K and above.

All these 32-bit Thumb instructions are available in ARMv6T2 and above, except that LDREXD and STREXD are not available in the ARMv7-M architecture.

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

*6.16 Address alignment* on page 6-128.

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 10.49 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 Thumb code

Thumb 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 ARM instructions

You can use SP in these ARM instructions but this is deprecated in ARMv6T2 and above.

You cannot use PC in instructions with the `LSL{S}{cond} Rd, Rm, Rs` syntax. You can use PC for `R`*`d`* and `R`*`m`* in the other syntax, but this is deprecated in ARMv6T2 and above.

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 ARM 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 this instruction are available in Thumb 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 ARM instruction is available in all architectures.

This 32-bit Thumb instruction is available in ARMv6T2 and above.

This 16-bit Thumb instruction is available in ARMv4T and above.

### Example

```
    LSLS    r1, r2, r3
```

### Related references

*10.55 MOV* on page 10-370.
*10.8 Condition code suffixes* on page 10-289.

## 10.50 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 Thumb code

Thumb instructions must not use PC or SP.

### Use of SP and PC in ARM instructions

You can use SP in these ARM instructions but they are deprecated in ARMv6T2 and above.

You cannot use PC in instructions with the `LSR{S}{cond} Rd, Rm, Rs` syntax. You can use PC for `R`*`d`* and `R`*`m`* in the other syntax, but this is deprecated in ARMv6T2 and above.

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 ARM 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 Thumb 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 ARM instruction is available in all architectures.

This 32-bit Thumb instruction is available in ARMv6T2 and above.

This 16-bit Thumb instruction is available in ARMv4T and above.

### Example

```
    LSR     r4, r5, r6
```

### Related references

*10.55 MOV* on page 10-370.
*10.8 Condition code suffixes* on page 10-289.

## 10.51 MCR and MCR2

Move to Coprocessor from ARM Register. Depending on the coprocessor, you might be able to specify various additional operations.

### 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 ARM 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 in the range 0 to 15.

*opcode1*
: is a 3-bit coprocessor-specific opcode.

*opcode2*
: is an optional 3-bit coprocessor-specific opcode.

*Rt*
: is an ARM source 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

The `MCR` ARM instruction is available in all versions of the ARM architecture.

The `MCR2` ARM instruction is available in ARMv5T and above.

These 32-bit Thumb instructions are available in ARMv6T2 and above.

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

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 10.52 MCRR and MCRR2

Move to Coprocessor from ARM Registers. Depending on the coprocessor, you might be able to specify various additional operations.

### Syntax

`MCRR{cond} coproc, #opcode, Rt, Rt2, CRn`

`MCRR2{cond} coproc, #opcode, Rt, Rt2, CRn`

where:

*cond*
: is an optional condition code. In ARM 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 in the range 0 to 15.

*opcode*
: is a 4-bit coprocessor-specific opcode.

*Rt, Rt2*
: are ARM source 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

The `MCRR` ARM instruction is available in ARMv6 and above, and E variants of ARMv5T.

The `MCRR2` ARM instruction is available in ARMv6 and above.

These 32-bit Thumb instructions are available in ARMv6T2 and above.

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

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 10.53 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

*Rn* must be different from *Rd* in architectures before ARMv6.

You cannot use PC for any register.

You can use SP in ARM instructions but this is deprecated in ARMv6T2 and above. You cannot use SP in Thumb instructions.

### Condition flags

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

- Updates the N and Z flags according to the result.
- Corrupts the C and V flag in ARMv4.
- Does not affect the C or V flag in ARMv5T and above.

### Architectures

The `MLA` ARM instruction is available in all versions of the ARM architecture.

This 32-bit Thumb instruction is available in ARMv6T2 and above.

### Example

```
MLA     r10, r2, r1, r5
```

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 10.54 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 ARM instructions but this is deprecated in ARMv6T2 and above. You cannot use SP in Thumb instructions.

### Architectures

The `MLS` ARM instruction is available in ARMv6T2 and above.

This 32-bit Thumb instruction is available in ARMv6T2 and above.

### Example

```
MLS     r4, r5, r6, r7
```

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 10.55 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 Thumb encodings

You cannot use PC (`R15`) for *Rd*, or in *Operand2*, in 32-bit Thumb `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 Thumb encodings

You can use PC or SP in 16-bit Thumb `MOV{cond} Rd, Rm` instructions but these instructions in which both *Rd* and *Rm* are SP or PC are deprecated in ARMv6T2 and above.

You cannot use PC or SP in any other `MOV{S}` 16-bit Thumb instructions.

### Use of PC and SP in ARM 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.
- The deprecation of PC and SP in ARM instructions only applies to ARMv6T2 and above.

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 Thumb 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`
: In architectures before ARMv6, either *Rd* or *Rm*, or both, must be a Hi register. In ARMv6 and above, this restriction does not apply.

### Architectures

The `#imm16` form of the ARM instruction is available in ARMv6T2 and above. The other forms of the ARM instruction are available in all versions of the ARM architecture.

This 32-bit Thumb instruction is available in ARMv6T2 and above.

This 16-bit Thumb instruction is available in all T variants of the ARM architecture.

### Related concepts

*4.4 Load immediate values using MOV and MVN* on page 4-61.

### Related references

*10.3 Flexible second operand (Operand2)* on page 10-282.
*10.138 SUBS pc, lr* on page 10-481.
*10.8 Condition code suffixes* on page 10-289.

## 10.56 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 Thumb bit (bit 0) of the address if the label referenced is in Thumb code.

### Architectures

This pseudo-instruction is available in ARMv6T2 and above in both ARM and Thumb.

### 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

*10.8 Condition code suffixes* on page 10-289.

## 10.57 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 ARM or Thumb instructions.

You can use SP for *Rd* in ARM instructions but this is deprecated.

You cannot use SP in Thumb instructions.

### Condition flags

This instruction does not change the flags.

### Architectures

This ARM instruction is available in ARMv6T2 and above.

This 32-bit Thumb instruction is available in ARMv6T2 and above.

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

### Related references

*10.56 MOV32 pseudo-instruction* on page 10-372.
*10.8 Condition code suffixes* on page 10-289.

## 10.58 MRC and MRC2

Move to ARM Register from Coprocessor. Depending on the coprocessor, you might be able to specify various additional operations.

### 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 ARM 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 in the range 0 to 15.

*opcode1*
: is a 3-bit coprocessor-specific opcode.

*opcode2*
: is an optional 3-bit coprocessor-specific opcode.

*Rt*
: is the ARM destination 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

The `MRC` ARM instruction is available in all versions of the ARM architecture.

The `MRC2` ARM instruction is available in ARMv5T and above.

These 32-bit Thumb instructions are available in ARMv6T2 and above.

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

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 10.59 MRRC and MRRC2

Move to ARM Registers from Coprocessor. Depending on the coprocessor, you might be able to specify various additional operations.

### Syntax

`MRRC{cond} coproc, #opcode, Rt, Rt2, CRm`

`MRRC2{cond} coproc, #opcode, Rt, Rt2, CRm`

where:

*cond*
: is an optional condition code. In ARM code, *cond* is not permitted for `MRRC2`.

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

*opcode*
: is a 4-bit coprocessor-specific opcode.

*Rt, Rt2*
: are ARM destination 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

The `MRRC` ARM instruction is available in ARMv6 and above, and E variants of ARMv5T.

The `MRRC2` ARM instruction is available in ARMv6 and above.

These 32-bit Thumb instructions are available in ARMv6T2 and above.

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

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 10.60 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 ARMv7-M and ARMv6-M, in privileged software execution only.

*Mpsr*
on ARMv7-M and ARMv6-M 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 ARM instructions. You can use SP for `Rd` in ARM instructions but this is deprecated in ARMv6T2 and above.

You cannot use PC or SP for `Rd` in Thumb instructions.

### Condition flags

This instruction does not change the flags.

### Architectures

This ARM instruction is available in all versions of the ARM architecture.

This 32-bit Thumb instruction is available in ARMv6T2 and above.

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

### Related concepts

*2.16 Current Program Status Register* on page 2-46.

### Related references

*10.61 MRS (system coprocessor register to ARM register)* on page 10-378.
*10.62 MSR (ARM register to system coprocessor register)* on page 10-379.
*10.63 MSR (general-purpose register to PSR)* on page 10-380.
*10.8 Condition code suffixes* on page 10-289.

## 10.61 MRS (system coprocessor register to ARM register)

Move to ARM 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 ARM destination 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 *ARMv7-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-R in ARM and 32-bit Thumb code.

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

### Related references

*10.60 MRS (PSR to general-purpose register)* on page 10-376.

*10.62 MSR (ARM register to system coprocessor register)* on page 10-379.

*10.63 MSR (general-purpose register to PSR)* on page 10-380.

*10.8 Condition code suffixes* on page 10-289.

### Related information

*ARM Architecture Reference Manual.*

## 10.62 MSR (ARM register to system coprocessor register)

Move to system coprocessor register from ARM 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 ARM source register. R*n* 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 *ARMv7-AR Architecture Reference Manual*. For example:

```
MSR SCTLR, R1 ; writes the contents of R1 into the CP15
              ;  coprocessor register SCTLR
```

### Architectures

This pseudo-instruction is available in ARMv7-R in ARM and 32-bit Thumb code.

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

### Related references

*10.60 MRS (PSR to general-purpose register)* on page 10-376.
*10.61 MRS (system coprocessor register to ARM register)* on page 10-378.
*10.63 MSR (general-purpose register to PSR)* on page 10-380.
*10.8 Condition code suffixes* on page 10-289.
*10.147 SYS* on page 10-492.

### Related information

*ARM Architecture Reference Manual.*

## 10.63 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 source register. R*m* must not be PC.

### Syntax

You can also use the following syntax on architectures other than ARMv7-M and ARMv6-M:

`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 Thumb.

*Rm*
is the source register. R*m* must not be PC.

*psr*
is one of:

**CPSR**
for use in Debug state, also deprecated synonym for APSR

**SPSR**
on any processor, in privileged software execution only.

*fields*
specifies the SPSR or CPSR fields to be moved. *fields* can be one or more of:

**c**
control field mask byte, PSR[7:0] (privileged software execution)

**x**
extension field mask byte, PSR[15:8] (privileged software execution)

`s`
: status field mask byte, PSR[23:16] (privileged software execution)

`f`
: flags field mask byte, PSR[31:24] (privileged software execution).

### Syntax

You can also use the following syntax on ARMv7-M and ARMv6-M only:

`MSR{`*cond*`} psr, Rm`

where:

*cond*
: is an optional condition code.

*Rm*
: is the source register. `R`*m* 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 ARM instructions. You can use SP for `R`*m* in ARM instructions but this is deprecated in ARMv6T2 and above.

You cannot use PC or SP in Thumb instructions.

### Condition flags

This instruction updates the flags explicitly if the `APSR_nzcvq` or `CPSR_f` field is specified.

### Architectures

This ARM instruction is available in all versions of the ARM architecture.

This 32-bit Thumb instruction is available in ARMv6T2 and above.

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

### Related references

*10.60 MRS (PSR to general-purpose register)* on page 10-376.
*10.61 MRS (system coprocessor register to ARM register)* on page 10-378.
*10.62 MSR (ARM register to system coprocessor register)* on page 10-379.
*10.8 Condition code suffixes* on page 10-289.

## 10.64 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

*Rn* must be different from *Rd* in architectures before ARMv6.

You cannot use PC for any register.

You can use SP in ARM instructions but this is deprecated in ARMv6T2 and above. You cannot use SP in Thumb instructions.

### Condition flags

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

- Updates the N and Z flags according to the result.
- Corrupts the C and V flag in ARMv4.
- Does not affect the C or V flag in ARMv5T and above.

### 16-bit instructions

The following forms of the `MUL` instruction are available in Thumb 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.

### Architectures

This ARM instruction is available in all versions of the ARM architecture.

`MUL` is available in a 32-bit encoding in Thumb in ARMv6T2 and above. `MULS` is not available in a 32-bit encoding in Thumb.

This 16-bit Thumb instruction is available in all T variants of the ARM architecture.

### Examples

```
MUL     r10, r2, r5
MULS    r0, r2, r2
MULLT   r2, r3, r2
```

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 10.65 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 Thumb MVN

You cannot use PC (`R15`) for *`Rd`*, or in *`Operand2`*, in 32-bit Thumb `MVN` instructions. You cannot use SP (`R13`) for *`Rd`*, or in *`Operand2`*.

### Use of PC and SP in 16-bit Thumb instructions

You cannot use PC or SP in any `MVN{S}` 16-bit Thumb instructions.

### Use of PC and SP in ARM 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**

The deprecation of PC and SP in ARM instructions only applies to ARMv6T2 and above.

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 Thumb 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 ARM instruction is available in all versions of the ARM architecture.

This 32-bit Thumb instruction is available in ARMv6T2 and above.

This 16-bit Thumb instruction is available in all T variants of the ARM architecture.

### Correct example

```
    MVNNE   r11, #0xF000000B ; ARM only. This immediate value is not available in Thumb.
```

### Incorrect example

```
    MVN     pc,r3,ASR r0     ; PC not permitted with register-controlled shift
```

### Related concepts

*4.4 Load immediate values using MOV and MVN* on page 4-61.

### Related references

*10.3 Flexible second operand (Operand2)* on page 10-282.
*10.138 SUBS pc, lr* on page 10-481.
*10.8 Condition code suffixes* on page 10-289.

## 10.66 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 ARM encoding of this pseudo-instruction is available in all versions of the ARM architecture.

The 32-bit Thumb encoding of this pseudo-instruction is available in ARMv6T2 and later.

### Register restrictions

In ARM instructions, using SP or PC for *Rd* or *Rm* is deprecated. In Thumb 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

*10.10 ADD* on page 10-292.

## 10.67 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` (ARM) or `MOV r8, r8` (Thumb).

`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 ARM, or a 32-bit boundary in Thumb.

### Architectures

This ARM instruction is available in ARMv6K and above.

This 32-bit Thumb instruction is available in ARMv6T2 and above.

This 16-bit Thumb instruction is available in ARMv6T2 and above.

`NOP` is available on all other ARM and Thumb architectures as a pseudo-instruction.

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 10.68 ORN (Thumb 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` Thumb 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 Thumb instruction is available in ARMv6T2 and above.

There is no ARM or 16-bit Thumb `ORN` instruction.

### Related references

*10.3 Flexible second operand (Operand2)* on page 10-282.
*10.138 SUBS pc, lr* on page 10-481.
*10.8 Condition code suffixes* on page 10-289.

## 10.69 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 Thumb instructions

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

### Use of PC and SP in ARM instructions

You can use PC and SP with the `ORR` instruction but this is deprecated in ARMv6T2 and above.

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 Thumb 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

*10.3 Flexible second operand (Operand2)* on page 10-282.
*10.138 SUBS pc, lr* on page 10-481.
*10.8 Condition code suffixes* on page 10-289.

## 10.70 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 ARM instructions but this is deprecated in ARMv6T2 and above. You cannot use SP in Thumb instructions.

### Condition flags

These instructions do not change the flags.

### Architectures

These ARM instructions are available in ARMv6 and above.

These 32-bit Thumb instructions are available in ARMv6T2 and above. For the ARMv7-M architecture, they are only available in an ARMv7E-M implementation.

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

### 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

*10.8 Condition code suffixes* on page 10-289.

## 10.71 PLD 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.

  `I`
  : Instruction address.

*`cond`*
: is an optional condition code.

  **Note**

  *`cond`* is permitted only in Thumb code, using a preceding `IT` instruction. This is an unconditional instruction in ARM 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 ARM instructions.
- –255 to +4095 for Thumb instructions, when *`Rn`* is not PC.
- –4095 to +4095 for Thumb 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 ARM code, the value in *`Rm`* is added to or subtracted from the value in *`Rn`*. In Thumb 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 Thumb instructions.
- Any one of the following for ARM 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 Thumb instructions *`Rm`* must also not be SP.

*`Rn`* must not be PC for Thumb instructions of the syntax `PL`*`type`*`{`*`cond`*`} [R`*`n`*`, ±R`*`m`*`{, #`*`shift`*`}]`.

### Architectures

ARM `PLD` is available in ARMv5TE and above.

The 32-bit Thumb encoding of `PLD` is available in ARMv6T2 and above.

`PLI` is available only in ARMv7 and above.

There are no 16-bit encodings of `PLD` or `PLI` in Thumb.

These are hint instructions, and their implementation is optional. If they are not implemented, they execute as `NOP`s.

### Related concepts

*7.5 Register-relative and PC-relative expressions* on page 7-136.

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 10.72 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.

In ARMv5T and above:
- Bits[1:0] must not be 0b10.
- If bit[0] is 1, execution continues in Thumb state.
- If bit[0] is 0, execution continues in ARM state.

In ARMv4, bits[1:0] of the address loaded must be 0b00.

### Thumb instructions

A subset of these instructions are available in the Thumb 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 ARM instructions

ARM `POP` instructions cannot have SP but can have PC in the *reglist*. These instructions that include both PC and LR in the *reglist* are deprecated in ARMv6T2 and above.

### Example

```
        POP     {r0,r10,pc} ; no 16-bit version available
```

### Related references

*10.41 LDM* on page 10-342.

*10.73 PUSH* on page 10-397.
*10.8 Condition code suffixes* on page 10-289.

## 10.73 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.

### Thumb 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 ARM instructions

ARM `PUSH` instructions can have SP and PC in the *reglist* but these instructions that include SP or PC in the *reglist* are deprecated in ARMv6T2 and above.

### Examples

```
    PUSH    {r0,r4-r7}
    PUSH    {r2,lr}
```

### Related references

*10.41 LDM* on page 10-342.
*10.72 POP* on page 10-395.
*10.8 Condition code suffixes* on page 10-289.

## 10.74 QADD

Signed saturating addition.

### Syntax

`QADD{cond} {Rd}, Rm, Rn`

where:

*cond*
: is an optional condition code.

*Rd*
: is the destination register.

*Rm, Rn*
: are the registers holding the operands.

### Operation

The `QADD` instruction adds the values in *Rm* and *Rn*. It saturates the result to the signed range $-2^{31} \leq x \leq 2^{31}-1$.

**Note**

All values are treated as two's complement signed integers by this instruction.

### Register restrictions

You cannot use PC for any operand.

You can use SP in ARM instructions but this is deprecated in ARMv6T2 and above. You cannot use SP in Thumb 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 ARM instruction is available in ARMv6 and above, and E variants of ARMv5T.

This 32-bit Thumb instruction is available in ARMv6T2 and above. For the ARMv7-M architecture, it is only available in an ARMv7E-M implementation.

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

### Example

```
    QADD    r0, r1, r9
```

### Related concepts

*2.15 The Q flag* on page 2-45.

### Related references

*10.60 MRS (PSR to general-purpose register)* on page 10-376.
*10.8 Condition code suffixes* on page 10-289.

## 10.75 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 ARM 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 register.

You can use SP in ARM instructions but this is deprecated in ARMv6T2 and above. You cannot use SP in Thumb instructions.

### Condition flags

This instruction does not affect the N, Z, C, V, Q, or GE flags.

### Architectures

This ARM instruction is available in ARMv6 and above.

This 32-bit Thumb instruction is available in ARMv6T2 and above. For the ARMv7-M architecture, it is only available in an ARMv7E-M implementation.

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

### Related concepts

*2.15 The Q flag* on page 2-45.

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 10.76 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 ARM 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 register.

You can use SP in ARM instructions but this is deprecated in ARMv6T2 and above. You cannot use SP in Thumb instructions.

### Condition flags

This instruction does not affect the N, Z, C, V, Q, or GE flags.

### Architectures

This ARM instruction is available in ARMv6 and above.

This 32-bit Thumb instruction is available in ARMv6T2 and above. For the ARMv7-M architecture, it is only available in an ARMv7E-M implementation.

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

### Related concepts

*2.15 The Q flag* on page 2-45.

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 10.77 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 ARM 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 register.

You can use SP in ARM instructions but this is deprecated in ARMv6T2 and above. You cannot use SP in Thumb instructions.

### Condition flags

This instruction does not affect the N, Z, C, V, Q, or GE flags.

### Architectures

This ARM instruction is available in ARMv6 and above.

This 32-bit Thumb instruction is available in ARMv6T2 and above. For the ARMv7-M architecture, it is only available in an ARMv7E-M implementation.

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

### Related concepts

*2.15 The Q flag* on page 2-45.

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 10.78 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 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 ARM instructions but this is deprecated in ARMv6T2 and above. You cannot use SP in Thumb 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 ARM instruction is available in ARMv6 and above, and E variants of ARMv5T.

This 32-bit Thumb instruction is available in ARMv6T2 and above. For the ARMv7-M architecture, it is only available in an ARMv7E-M implementation.

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

### Related concepts

*2.15 The Q flag* on page 2-45.

### Related references

*10.60 MRS (PSR to general-purpose register)* on page 10-376.
*10.8 Condition code suffixes* on page 10-289.

## 10.79 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 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 ARM instructions but this is deprecated in ARMv6T2 and above. You cannot use SP in Thumb 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 ARM instruction is available in ARMv6 and above, and E variants of ARMv5T.

This 32-bit Thumb instruction is available in ARMv6T2 and above. For the ARMv7-M architecture, it is only available in an ARMv7E-M implementation.

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

### Example

```
QDSUBLT r9, r0, r1
```

### Related concepts

*2.15 The Q flag* on page 2-45.

### Related references

*10.60 MRS (PSR to general-purpose register)* on page 10-376.
*10.8 Condition code suffixes* on page 10-289.

## 10.80 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 ARM 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 register.

You can use SP in ARM instructions but this is deprecated in ARMv6T2 and above. You cannot use SP in Thumb instructions.

### Condition flags

This instruction does not affect the N, Z, C, V, Q, or GE flags.

### Architectures

This ARM instruction is available in ARMv6 and above.

This 32-bit Thumb instruction is available in ARMv6T2 and above. For the ARMv7-M architecture, it is only available in an ARMv7E-M implementation.

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

### Related concepts

*2.15 The Q flag* on page 2-45.

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 10.81 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 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 ARM instructions but this is deprecated in ARMv6T2 and above. You cannot use SP in Thumb 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 ARM instruction is available in ARMv6 and above, and E variants of ARMv5T.

This 32-bit Thumb instruction is available in ARMv6T2 and above. For the ARMv7-M architecture, it is only available in an ARMv7E-M implementation.

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

### Related concepts

*2.15 The Q flag* on page 2-45.

### Related references

*10.60 MRS (PSR to general-purpose register)* on page 10-376.
*10.8 Condition code suffixes* on page 10-289.

## 10.82 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 ARM 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 register.

You can use SP in ARM instructions but this is deprecated in ARMv6T2 and above. You cannot use SP in Thumb instructions.

### Condition flags

This instruction does not affect the N, Z, C, V, Q, or GE flags.

### Architectures

This ARM instruction is available in ARMv6 and above.

This 32-bit Thumb instruction is available in ARMv6T2 and above. For the ARMv7-M architecture, it is only available in an ARMv7E-M implementation.

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

### Related concepts

*2.15 The Q flag* on page 2-45.

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 10.83 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 ARM 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 register.

You can use SP in ARM instructions but this is deprecated in ARMv6T2 and above. You cannot use SP in Thumb instructions.

### Condition flags

This instruction does not affect the N, Z, C, V, Q, or GE flags.

### Architectures

This ARM instruction is available in ARMv6 and above.

This 32-bit Thumb instruction is available in ARMv6T2 and above. For the ARMv7-M architecture, it is only available in an ARMv7E-M implementation.

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

### Related concepts

*2.15 The Q flag* on page 2-45.

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 10.84 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 ARM instructions but this is deprecated in ARMv6T2 and above. You cannot use SP in Thumb instructions.

### Condition flags

This instruction does not change the flags.

### Architectures

This ARM instruction is available in ARMv6T2 and above.

This 32-bit Thumb instruction is available in ARMv6T2 and above.

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

### Example

```
    RBIT    r7, r8
```

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 10.85 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 ARM instructions but this is deprecated in ARMv6T2 and above. You cannot use SP in Thumb instructions.

### Condition flags

This instruction does not change the flags.

### 16-bit instructions

The following form of this instruction is available in Thumb code, and is a 16-bit instruction:

`REV Rd, Rm`
: *Rd* and *Rm* must both be Lo registers.

### Architectures

This ARM instruction is available in ARMv6 and above.

This 32-bit Thumb instruction is available in ARMv6T2 and above.

This 16-bit Thumb instruction is available in ARMv6 and above.

### Example

```
    REV     r3, r7
```

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 10.86 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 ARM instructions but this is deprecated in ARMv6T2 and above. You cannot use SP in Thumb instructions.

### Condition flags

This instruction does not change the flags.

### 16-bit instructions

The following form of this instruction is available in Thumb code, and is a 16-bit instruction:

`REV16 Rd, Rm`
: *Rd* and *Rm* must both be Lo registers.

### Architectures

This ARM instruction is available in ARMv6 and above.

This 32-bit Thumb instruction is available in ARMv6T2 and above.

This 16-bit Thumb instruction is available in ARMv6 and above.

### Example

```
REV16   r0, r0
```

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 10.87 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 ARM instructions but this is deprecated in ARMv6T2 and above. You cannot use SP in Thumb instructions.

### Condition flags

This instruction does not change the flags.

### 16-bit instructions

The following form of this instruction is available in Thumb code, and is a 16-bit instruction:

`REVSH Rd, Rm`
: *Rd* and *Rm* must both be Lo registers.

### Architectures

This ARM instruction is available in ARMv6 and above.

This 32-bit Thumb instruction is available in ARMv6T2 and above.

This 16-bit Thumb instruction is available in ARMv6 and above.

### Example

```
    REVSH   r0, r5       ; Reverse Signed Halfword
```

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 10.88 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 (ARM only)

  **DA**
  : Decrement address After each transfer (ARM 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 Thumb code, using a preceding `IT` instruction. This is an unconditional instruction in ARM 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 ARM, the address written to the PC must be word-aligned.
- For a return to Thumb, 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 ARM instruction is available in ARMv6 and above.

This 32-bit Thumb instruction is available in ARMv6T2 and above, except the ARMv7-M architecture.

There is no 16-bit version of this instruction.

### Example

```
RFE sp!
```

### Related concepts

*2.4 Processor modes, and privileged and unprivileged software execution* on page 2-33.

### Related references

*10.125 SRS* on page 10-458.
*10.8 Condition code suffixes* on page 10-289.

## 10.89 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 Thumb code

Thumb instructions must not use PC or SP.

### Use of SP and PC in ARM instructions

You can use SP in these ARM instructions but this is deprecated in ARMv6T2 and above.

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 in ARMv6T2 and above.

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 ARM 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 Thumb 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 ARM instruction is available in all architectures.

This 32-bit Thumb instruction is available in ARMv6T2 and above.

This 16-bit Thumb instruction is available in ARMv4T and above.

### Example

```
    ROR     r4, r5, r6
```

### Related references

*10.55 MOV* on page 10-370.
*10.8 Condition code suffixes* on page 10-289.

## 10.90 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 Thumb code

Thumb instructions must not use PC or SP.

### Use of SP and PC in ARM instructions

You can use SP in this ARM instruction but this is deprecated in ARMv6T2 and above.

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 ARM 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

This ARM instruction is available in all architectures.

This 32-bit Thumb instruction is available in ARMv6T2 and above.

There is no 16-bit `RRX` instruction in Thumb.

### Related references

*10.55 MOV* on page 10-370.
*10.8 Condition code suffixes* on page 10-289.

## 10.91 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 Thumb 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 ARM 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 in RSB ARM instructions is deprecated.

——— **Note** ———

The deprecation of SP and PC in ARM instructions is only in ARMv6T2 and above.

———————————

### 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 Thumb 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

*10.3 Flexible second operand (Operand2)* on page 10-282.
*10.8 Condition code suffixes* on page 10-289.

## 10.92 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 Thumb code.

### Use of PC and SP

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

Use of PC for any operand in RSC 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 in RSC instructions is deprecated.

——— **Note** ———

The deprecation of SP and PC is only in ARMv6T2 and above.

———————————

### 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

*10.3 Flexible second operand (Operand2)* on page 10-282.

*10.8 Condition code suffixes* on page 10-289.

## 10.93 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 ARM 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 register.

You can use SP in ARM instructions but this is deprecated in ARMv6T2 and above. You cannot use SP in Thumb 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.

### Architectures

This ARM instruction is available in ARMv6 and above.

This 32-bit Thumb instruction is available in ARMv6T2 and above. For the ARMv7-M architecture, it is only available in an ARMv7E-M implementation.

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

### Related references

*10.99 SEL* on page 10-430.

*10.8 Condition code suffixes* on page 10-289.

## 10.94 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 ARM 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 register.

You can use SP in ARM instructions but this is deprecated in ARMv6T2 and above. You cannot use SP in Thumb 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.

### Architectures

This ARM instruction is available in ARMv6 and above.

This 32-bit Thumb instruction is available in ARMv6T2 and above. For the ARMv7-M architecture, it is only available in an ARMv7E-M implementation.

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

### Related references

*10.99 SEL* on page 10-430.
*10.8 Condition code suffixes* on page 10-289.

## 10.95 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 ARM 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 register.

You can use SP in ARM instructions but this is deprecated in ARMv6T2 and above. You cannot use SP in Thumb 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.

### Architectures

This ARM instruction is available in ARMv6 and above.

This 32-bit Thumb instruction is available in ARMv6T2 and above. For the ARMv7-M architecture, it is only available in an ARMv7E-M implementation.

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

### Related references

*10.99 SEL* on page 10-430.
*10.8 Condition code suffixes* on page 10-289.

## 10.96 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 Thumb 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 ARM 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 in SBC ARM instructions is deprecated.

——— **Note** ———

The deprecation of SP and PC in ARM instructions is only in ARMv6T2 and above.

———————————

### 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 Thumb 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

*10.3 Flexible second operand (Operand2)* on page 10-282.
*10.8 Condition code suffixes* on page 10-289.

## 10.97 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 register.

You can use SP in the ARM instruction but this is deprecated in ARMv6T2 and above. You cannot use SP in the Thumb instruction.

### Condition flags

This instruction does not alter any flags.

### Architectures

This ARM instruction is available in ARMv6T2 and above.

This 32-bit Thumb instruction is available in ARMv6T2 and above.

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

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 10.98 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 Thumb instruction is available in ARMv7-R and ARMv7-M.

This ARM instruction is optional in ARMv7-R.

There is no 16-bit Thumb `SDIV` instruction.

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 10.99 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 register.

You can use SP in ARM instructions but this is deprecated in ARMv6T2 and above. You cannot use SP in Thumb instructions.

### Condition flags

This instruction does not change the flags.

### Architectures

This ARM instruction is available in ARMv6 and above.

This 32-bit Thumb instruction is available in ARMv6T2 and above. For the ARMv7-M architecture, it is only available in an ARMv7E-M implementation.

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

### 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

*2.14 Application Program Status Register* on page 2-44.

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 10.100 SETEND

Set the endianness bit in the CPSR, without affecting any other bits in the CPSR.

### 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 ARM instruction is available in ARMv6 and above.

This 16-bit Thumb instruction is available in T variants of ARMv6 and above, except the ARMv6-M and ARMv7-M architectures.

There is no 32-bit version of this instruction in Thumb.

### 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
```

## 10.101 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` executes as a `NOP` instruction in ARMv6T2.

`SEV` causes an event to be signaled to all cores within a multiprocessor system. If `SEV` is implemented, `WFE` must also be implemented.

### Architectures

This ARM instruction is available in ARMv6K and above.

This 32-bit Thumb instruction is available in ARMv6T2 and above.

This 16-bit Thumb instruction is available in ARMv6T2 and above.

### Related references

*10.67 NOP* on page 10-387.
*10.8 Condition code suffixes* on page 10-289.

## 10.102 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 ARM 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 register.

You can use SP in ARM instructions but this is deprecated in ARMv6T2 and above. You cannot use SP in Thumb instructions.

### Condition flags

This instruction does not affect the N, Z, C, V, Q, or GE flags.

### Architectures

This ARM instruction is available in ARMv6 and above.

This 32-bit Thumb instruction is available in ARMv6T2 and above. For the ARMv7-M architecture, it is only available in an ARMv7E-M implementation.

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

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 10.103 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 ARM 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 register.

You can use SP in ARM instructions but this is deprecated in ARMv6T2 and above. You cannot use SP in Thumb instructions.

### Condition flags

This instruction does not affect the N, Z, C, V, Q, or GE flags.

### Architectures

This ARM instruction is available in ARMv6 and above.

This 32-bit Thumb instruction is available in ARMv6T2 and above. For the ARMv7-M architecture, it is only available in an ARMv7E-M implementation.

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

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 10.104 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 ARM 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 register.

You can use SP in ARM instructions but this is deprecated in ARMv6T2 and above. You cannot use SP in Thumb instructions.

### Condition flags

This instruction does not affect the N, Z, C, V, Q, or GE flags.

### Architectures

This ARM instruction is available in ARMv6 and above.

This 32-bit Thumb instruction is available in ARMv6T2 and above. For the ARMv7-M architecture, it is only available in an ARMv7E-M implementation.

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

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 10.105 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 ARM 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 register.

You can use SP in ARM instructions but this is deprecated in ARMv6T2 and above. You cannot use SP in Thumb instructions.

### Condition flags

This instruction does not affect the N, Z, C, V, Q, or GE flags.

### Architectures

This ARM instruction is available in ARMv6 and above.

This 32-bit Thumb instruction is available in ARMv6T2 and above. For the ARMv7-M architecture, it is only available in an ARMv7E-M implementation.

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

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 10.106 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 ARM 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 register.

You can use SP in ARM instructions but this is deprecated in ARMv6T2 and above. You cannot use SP in Thumb instructions.

### Condition flags

This instruction does not affect the N, Z, C, V, Q, or GE flags.

### Architectures

This ARM instruction is available in ARMv6 and above.

This 32-bit Thumb instruction is available in ARMv6T2 and above. For the ARMv7-M architecture, it is only available in an ARMv7E-M implementation.

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

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 10.107 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 ARM 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 register.

You can use SP in ARM instructions but this is deprecated in ARMv6T2 and above. You cannot use SP in Thumb instructions.

### Condition flags

This instruction does not affect the N, Z, C, V, Q, or GE flags.

### Architectures

This ARM instruction is available in ARMv6 and above.

This 32-bit Thumb instruction is available in ARMv6T2 and above. For the ARMv7-M architecture, it is only available in an ARMv7E-M implementation.

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

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 10.108 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 ARM assembly language. `SMI` instructions disassemble to `SMC`, with a comment to say that this was formerly `SMI`.

### Architectures

This ARM instruction is available in implementations of ARMv6 and above, if they have the Security Extensions.

This 32-bit Thumb instruction is available in implementations of ARMv6T2 and above, if they have the Security Extensions.

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

### Related references

*10.8 Condition code suffixes* on page 10-289.

### Related information

*ARM Architecture Reference Manual.*

## 10.109 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 register.

You can use SP in ARM instructions but this is deprecated in ARMv6T2 and above. You cannot use SP in Thumb 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.

### Architectures

This ARM instruction is available in ARMv6 and above, and E variants of ARMv5T.

This 32-bit Thumb instruction is available in ARMv6T2 and above. For the ARMv7-M architecture, it is only available in an ARMv7E-M implementation.

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

### Examples

```
    SMLABBNE    r0, r2, r1, r10
    SMLABT      r0, r0, r3, r5
```

### Related references

*10.60 MRS (PSR to general-purpose register)* on page 10-376.
*10.63 MSR (general-purpose register to PSR)* on page 10-380.
*10.8 Condition code suffixes* on page 10-289.

## 10.110 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 register.

You can use SP in ARM instructions but this is deprecated in ARMv6T2 and above. You cannot use SP in Thumb instructions.

### Condition flags

This instruction does not change the flags.

### Architectures

This ARM instruction is available in ARMv6 and above.

This 32-bit Thumb instruction is available in ARMv6T2 and above. For the ARMv7-M architecture, it is only available in an ARMv7E-M implementation.

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

### Example

```
        SMLADLT     r1, r2, r4, r1
```

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 10.111 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 ARM 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 ARM 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

*Rn* must be different from *RdLo* and *RdHi* in architectures before ARMv6.

You cannot use PC for any register.

You can use SP in ARM instructions but this is deprecated in ARMv6T2 and above. You cannot use SP in Thumb 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 ARM instruction is available in all versions of the ARM architecture.

This 32-bit Thumb instruction is available in ARMv6T2 and above.

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

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 10.112 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 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 register.

You can use SP in ARM instructions but this is deprecated in ARMv6T2 and above. You cannot use SP in Thumb instructions.

### Condition flags

This instruction does not change the flags.

### Architectures

This ARM instruction is available in ARMv6 and above.

This 32-bit Thumb instruction is available in ARMv6T2 and above. For the ARMv7-M architecture, it is only available in an ARMv7E-M implementation.

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

### Example

```
    SMLALD      r10, r11, r5, r1
```

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 10.113 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 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 register.

You can use SP in ARM instructions but this is deprecated in ARMv6T2 and above. You cannot use SP in Thumb 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.

### Architectures

This ARM instruction is available in ARMv6 and above, and E variants of ARMv5T.

This 32-bit Thumb instruction is available in ARMv6T2 and above. For the ARMv7-M architecture, it is only available in an ARMv7E-M implementation.

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

### Examples

```
SMLALTB     r2, r3, r7, r1
SMLALBTVS   r0, r1, r9, r2
```

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 10.114 SMLAWy

Signed Multiply-Accumulate Wide, with one 32-bit operand 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 register.

You can use SP in ARM instructions but this is deprecated in ARMv6T2 and above. You cannot use SP in Thumb 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.

### Architectures

This ARM instruction is available in ARMv6 and above, and E variants of ARMv5T.

This 32-bit Thumb instruction is available in ARMv6T2 and above. For the ARMv7-M architecture, it is only available in an ARMv7E-M implementation.

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

### Related references

*10.60 MRS (PSR to general-purpose register)* on page 10-376.
*10.8 Condition code suffixes* on page 10-289.

## 10.115 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 register.

You can use SP in ARM instructions but this is deprecated in ARMv6T2 and above. You cannot use SP in Thumb instructions.

### Condition flags

This instruction does not change the flags.

### Architectures

This ARM instruction is available in ARMv6 and above.

This 32-bit Thumb instruction is available in ARMv6T2 and above. For the ARMv7-M architecture, this instruction is only available in an ARMv7E-M implementation.

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

### Examples

```
SMLSD       r1, r2, r0, r7
SMLSDX      r11, r10, r2, r3
```

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 10.116 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 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 register.

You can use SP in ARM instructions but this is deprecated in ARMv6T2 and above. You cannot use SP in Thumb instructions.

### Condition flags

This instruction does not change the flags.

### Architectures

This ARM instruction is available in ARMv6 and above.

This 32-bit Thumb instruction is available in ARMv6T2 and above. For the ARMv7-M architecture, it is only available in an ARMv7E-M implementation.

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

### Example

```
    SMLSLD      r3, r0, r5, r1
```

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 10.117 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 register.

You can use SP in ARM instructions but this is deprecated in ARMv6T2 and above. You cannot use SP in Thumb instructions.

### Condition flags

This instruction does not change the flags.

### Architectures

This ARM instruction is available in ARMv6 and above.

This 32-bit Thumb instruction is available in ARMv6T2 and above. For the ARMv7-M architecture, it is only available in an ARMv7E-M implementation.

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

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 10.118 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 register.

You can use SP in ARM instructions but this is deprecated in ARMv6T2 and above. You cannot use SP in Thumb instructions.

### Condition flags

This instruction does not change the flags.

### Architectures

This ARM instruction is available in ARMv6 and above.

This 32-bit Thumb instruction is available in ARMv6T2 and above. For the ARMv7-M architecture, it is only available in an ARMv7E-M implementation.

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

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 10.119 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 register.

You can use SP in ARM instructions but this is deprecated in ARMv6T2 and above. You cannot use SP in Thumb instructions.

### Condition flags

This instruction does not change the flags.

### Architectures

This ARM instruction is available in ARMv6 and above.

This 32-bit Thumb instruction is available in ARMv6T2 and above. For the ARMv7-M architecture, it is only available in an ARMv7E-M implementation.

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

### Examples

```
    SMMULGE     r6, r4, r3
    SMMULR      r2, r2, r2
```

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 10.120 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 register.

You can use SP in ARM instructions but this is deprecated in ARMv6T2 and above. You cannot use SP in Thumb instructions.

### Q flag

The `SMUAD` instruction sets the Q flag if the addition overflows.

### Architectures

This ARM instruction is available in ARMv6 and above.

This 32-bit Thumb instruction is available in ARMv6T2 and above. For the ARMv7-M architecture, it is only available in an ARMv7E-M implementation.

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

### Examples

```
    SMUAD       r2, r3, r2
```

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 10.121 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 register.

You can use SP in ARM instructions but this is deprecated in ARMv6T2 and above. You cannot use SP in Thumb instructions.

### Condition flags

These instructions do not affect the N, Z, C, or V flags.

### Architectures

This ARM instruction is available in ARMv6 and above, and E variants of ARMv5T.

This 32-bit Thumb instruction is available in ARMv6T2 and above. For the ARMv7-M architecture, it is only available in an ARMv7E-M implementation.

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

### Examples

```
    SMULTBEQ    r8, r7, r9
```

### Related references

*10.60 MRS (PSR to general-purpose register)* on page 10-376.
*10.63 MSR (general-purpose register to PSR)* on page 10-380.
*10.8 Condition code suffixes* on page 10-289.

## 10.122 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 ARM 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 ARM 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

*`Rn`* must be different from *`RdLo`* and *`RdHi`* in architectures before ARMv6.

You cannot use PC for any register.

You can use SP in ARM instructions but this is deprecated in ARMv6T2 and above. You cannot use SP in Thumb 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 ARM instruction is available in all versions of the ARM architecture.

This 32-bit Thumb instruction is available in ARMv6T2 and above.

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

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 10.123 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 register.

You can use SP in ARM instructions but this is deprecated in ARMv6T2 and above. You cannot use SP in Thumb instructions.

### Condition flags

This instruction does not affect the N, Z, C, or V flags.

### Architectures

This ARM instruction is available in ARMv6 and above, and E variants of ARMv5T.

This 32-bit Thumb instruction is available in ARMv6T2 and above. For the ARMv7-M architecture, it is only available in an ARMv7E-M implementation.

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

### Related references

*10.60 MRS (PSR to general-purpose register)* on page 10-376.
*10.8 Condition code suffixes* on page 10-289.

## 10.124 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 register.

You can use SP in ARM instructions but this is deprecated in ARMv6T2 and above. You cannot use SP in Thumb instructions.

### Architectures

This ARM instruction is available in ARMv6 and above.

This 32-bit Thumb instruction is available in ARMv6T2 and above. For the ARMv7-M architecture, it is only available in an ARMv7E-M implementation.

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

### Example

```
    SMUSDXNE    r0, r1, r2
```

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 10.125 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 (ARM only)

  **DA**
  : Decrement address After each transfer (ARM 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 Thumb code, using a preceding `IT` instruction. This is an unconditional instruction in ARM.

**!**
: 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.

### Architectures

This ARM instruction is available in ARMv6 and above.

This 32-bit Thumb instruction is available in ARMv6T2 and above, except the ARMv7-M architecture.

There is no 16-bit version of this instruction.

### Example

```
R13_usr  EQU     16
         SRSFD   sp,#R13_usr
```

### Related concepts

*4.15 Stack implementation using LDM and STM* on page 4-77.
*2.4 Processor modes, and privileged and unprivileged software execution* on page 2-33.

### Related references

*10.41 LDM* on page 10-342.
*10.8 Condition code suffixes* on page 10-289.

## 10.126 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 (ARM) or 1-31 (Thumb)

  `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 register.

You can use SP in ARM instructions but this is deprecated in ARMv6T2 and above. You cannot use SP in Thumb 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 ARM instruction is available in ARMv6 and above.

This 32-bit Thumb instruction is available in ARMv6T2 and above.

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

### Example

```
    SSAT    r7, #16, r7, LSL #4
```

### Related references

*10.127 SSAT16* on page 10-461.
*10.60 MRS (PSR to general-purpose register)* on page 10-376.
*10.8 Condition code suffixes* on page 10-289.

## 10.127 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 register.

You can use SP in ARM instructions but this is deprecated in ARMv6T2 and above. You cannot use SP in Thumb 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.

### Architectures

This ARM instruction is available in ARMv6 and above.

This 32-bit Thumb instruction is available in ARMv6T2 and above. For the ARMv7-M architecture, it is only available in an ARMv7E-M implementation.

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

### Correct example

```
SSAT16  r7, #12, r7
```

### Incorrect example

```
SSAT16  r1, #16, r2, LSL #4 ; shifts not permitted with halfword
                            ; saturations
```

### Related references

*10.60 MRS (PSR to general-purpose register)* on page 10-376.
*10.8 Condition code suffixes* on page 10-289.

## 10.128 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 ARM 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 register.

You can use SP in ARM instructions but this is deprecated in ARMv6T2 and above. You cannot use SP in Thumb 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.

### Architectures

This ARM instruction is available in ARMv6 and above.

This 32-bit Thumb instruction is available in ARMv6T2 and above. For the ARMv7-M architecture, it is only available in an ARMv7E-M implementation.

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

### Related references

*10.99 SEL* on page 10-430.
*10.8 Condition code suffixes* on page 10-289.

## 10.129 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 ARM 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 register.

You can use SP in ARM instructions but this is deprecated in ARMv6T2 and above. You cannot use SP in Thumb 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.

### Architectures

This ARM instruction is available in ARMv6 and above.

This 32-bit Thumb instruction is available in ARMv6T2 and above. For the ARMv7-M architecture, it is only available in an ARMv7E-M implementation.

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

### Related references

*10.99 SEL* on page 10-430.

*10.8 Condition code suffixes* on page 10-289.

## 10.130 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 ARM 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 register.

You can use SP in ARM instructions but this is deprecated in ARMv6T2 and above. You cannot use SP in Thumb 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.

### Architectures

This ARM instruction is available in ARMv6 and above.

This 32-bit Thumb instruction is available in ARMv6T2 and above. For the ARMv7-M architecture, it is only available in an ARMv7E-M implementation.

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

### Related references

*10.99 SEL* on page 10-430.
*10.8 Condition code suffixes* on page 10-289.

## 10.131 STC and STC2

Transfer Data between memory and Coprocessor.

### 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 ARM 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 in the range 0 to 15.

*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

`STC` is available in all versions of the ARM architecture.

`STC2` is available in ARMv5T and above.

These 32-bit Thumb instructions are available in ARMv6T2 and above.

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

### 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 Thumb `STC` and `STC2` instructions.

ARM `STC` and `STC2` instructions where `Rn` is PC, are deprecated in ARMv6T2 and above.

### Related concepts

*7.5 Register-relative and PC-relative expressions* on page 7-136.
*6.16 Address alignment* on page 6-128.

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 10.132 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 (ARM only).

  **DA**
  : Decrement address After each transfer (ARM 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 ARM 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 ARM state, but there are some restrictions in Thumb state.

**^**
: is an optional suffix, available in ARM 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 Thumb instructions

In 32-bit Thumb 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 ARM instructions

ARM store instructions can have SP and PC in the *reglist* but these instructions that include SP or PC in the *reglist* are deprecated in ARMv6T2 and above.

### 16-bit instruction

A 16-bit version of this instruction is available in Thumb 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 Thumb STM instructions with writeback that specify *Rn* as the lowest register in the *reglist* are deprecated in ARMv6T2 and above.

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 ARM or 16-bit Thumb 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 in ARMv6T2 and above.
- Otherwise, the stored value of R*n* cannot be relied on, so these instructions are not permitted.

32-bit Thumb 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

*4.15 Stack implementation using LDM and STM* on page 4-77.
*6.16 Address alignment* on page 6-128.

### Related references

*10.72 POP* on page 10-395.
*10.8 Condition code suffixes* on page 10-289.

## 10.133 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 register to store.

*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 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 10-15 Offsets and architectures, STR, word, halfword, and byte**

| Instruction | Immediate offset | Pre-indexed | Post-indexed | Arch. |
|---|---|---|---|---|
| ARM, word or byte | –4095 to 4095 | –4095 to 4095 | –4095 to 4095 | All |
| ARM, halfword | –255 to 255 | –255 to 255 | –255 to 255 | All |
| ARM, doubleword | –255 to 255 | –255 to 255 | –255 to 255 | 5E |
| Thumb 32-bit encoding, word, halfword, or byte | –255 to 4095 | –255 to 255 | –255 to 255 | T2 |
| Thumb 32-bit encoding, doubleword | –1020 to 1020 [ab] | –1020 to 1020 [ab] | –1020 to 1020 [ab] | T2 |
| Thumb 16-bit encoding, word [ac] | 0 to 124 [ab] | Not available | Not available | T |

[ab] Must be divisible by 4.
[ac] Rt and Rn must be in the range R0-R7.

**Table 10-15 Offsets and architectures, STR, word, halfword, and byte (continued)**

| Instruction | Immediate offset | Pre-indexed | Post-indexed | Arch. |
|---|---|---|---|---|
| Thumb 16-bit encoding, halfword [ac] | 0 to 62 [ae] | Not available | Not available | T |
| Thumb 16-bit encoding, byte [ac] | 0 to 31 | Not available | Not available | T |
| Thumb 16-bit encoding, word, Rn is SP [ad] | 0 to 1020 [ab] | Not available | Not available | T |

### Notes about the Architecture column

Entries in the Architecture column indicate that the instructions are available as follows:

**All**
: All versions of the ARM architecture.

**5E**
: The ARMv5TE, ARMv6*, and ARMv7 architectures.

**T2**
: The ARMv6T2 and above architectures.

**T**
: The ARMv4T, ARMv5T*, ARMv6*, and ARMv7 architectures.

### Register restrictions

R*n* must be different from R*t* in the pre-index and post-index forms.

### Doubleword register restrictions

R*n* must be different from R*t2* in the pre-index and post-index forms.

For Thumb instructions, you must not specify SP or PC for either *Rt* or *Rt2*.

For ARM 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 ARM instructions you can use PC for R*t* in STR word instructions and PC for R*n* in STR instructions with immediate offset syntax (that is the forms that do not writeback to the R*n*). However, this is deprecated in ARMv6T2 and above.

Other uses of PC are not permitted in these ARM instructions.

In Thumb code, using PC in STR instructions is not permitted.

### Use of SP

You can use SP for R*n*.

In ARM code, you can use SP for R*t* in word instructions. You can use SP for R*t* in non-word instructions in ARM code but this is deprecated in ARMv6T2 and above.

In Thumb code, you can use SP for R*t* in word instructions only. All other use of SP for R*t* in this instruction is not permitted in Thumb code.

[ad] Rt must be in the range R0-R7.
[ae] Must be divisible by 2.

### Example

```
    STR     r2,[r9,#consta-struc]   ; consta-struc is an expression
                                    ; evaluating to a constant in
                                    ; the range 0-4095.
```

### Related concepts

*6.16 Address alignment* on page 6-128.

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 10.134 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 ; ARM only
STR{type}{cond} Rt, [Rn], ±Rm {, shift} ; post-indexed ; ARM only
STRD{cond} Rt, Rt2, [Rn, ±Rm] ; register offset, doubleword ; ARM only
STRD{cond} Rt, Rt2, [Rn, ±Rm]! ; pre-indexed, doubleword ; ARM only
STRD{cond} Rt, Rt2, [Rn], ±Rm ; post-indexed, doubleword ; ARM 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 store.

*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 Thumb 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 10-16 Options and architectures, STR (register offsets)**

| Instruction | +/–*Rm* [af] | shift | Arch. |
|---|---|---|---|
| ARM, word or byte | +/–*Rm* | `LSL #0-31 LSR #1-32` | All |
| | | `ASR #1-32 ROR #1-31 RRX` | |
| ARM, halfword | +/–*Rm* | Not available | All |
| ARM, doubleword | +/–*Rm* | Not available | 5E |

[af] Where +/–Rm is shown, you can use –Rm, +Rm, or Rm. Where +Rm is shown, you cannot use –Rm.
[ag] Rt, Rn, and Rm must all be in the range R0-R7.

**Table 10-16 Options and architectures, STR (register offsets) (continued)**

| Instruction | +/–*Rm* [af] | shift | Arch. |
|---|---|---|---|
| Thumb 32-bit encoding, word, halfword, or byte | +*Rm* | `LSL #0-3` | T2 |
| Thumb 16-bit encoding, all except doubleword [ag] | +*Rm* | Not available | T |

### Notes about the Architecture column

Entries in the Architecture column indicate that the instructions are available as follows:

**All**
All versions of the ARM architecture.

**5E**
The ARMv5TE, ARMv6*, and ARMv7 architectures.

**T2**
The ARMv6T2 and above architectures.

**T**
The ARMv4T, ARMv5T*, ARMv6*, and ARMv7 architectures.

### Register restrictions

In the pre-index and post-index forms:

- R*n* must be different from `Rt`.
- R*n* must be different from R*m* in architectures before ARMv6.

### Doubleword register restrictions

For ARM 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)`.
- R*n* must be different from *Rt2* in the pre-index and post-index forms.

### Use of PC

In ARM instructions you can use PC for `Rt` in `STR` word instructions, and you can use PC for R*n* in `STR` instructions with register offset syntax (that is, the forms that do not writeback to the R*n*). However, this is deprecated in ARMv6T2 and above.

Other uses of PC are not permitted in ARM instructions.

Use of PC in `STR` Thumb instructions is not permitted.

### Use of SP

You can use SP for R*n*.

In ARM code, you can use SP for `Rt` in word instructions. You can use SP for `Rt` in non-word ARM instructions but this is deprecated in ARMv6T2 and above.

You can use SP for R*m* in ARM instructions but this is deprecated in ARMv6T2 and above.

In Thumb 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 Thumb code.

Use of SP for R*m* is not permitted in Thumb state.

### Related concepts

*6.16 Address alignment* on page 6-128.

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 10.135 STR, unprivileged

Unprivileged Store, byte, halfword, or word.

### Syntax

```
STR{type}T{cond} Rt, [Rn {, #offset}] ; immediate offset (Thumb, 32-bit encoding only)
STR{type}T{cond} Rt, [Rn] {, #offset} ; post-indexed (ARM only)
STR{type}T{cond} Rt, [Rn], ±Rm {, shift} ; post-indexed (register) (ARM 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 10-17 Offsets and architectures, STR (User mode)**

| Instruction | Immediate offset | Post-indexed | +/–*Rm* [ah] | shift | Arch. |
|---|---|---|---|---|---|
| ARM, word or byte | Not available | –4095 to 4095 | +/–*Rm* | `LSL #0-31` | All |
| | | | | `LSR #1-32` | |
| | | | | `ASR #1-32` | |
| | | | | `ROR #1-31` | |

[ah] You can use –Rm, +Rm, or Rm.

**Table 10-17 Offsets and architectures, STR (User mode) (continued)**

| Instruction | Immediate offset | Post-indexed | +/–*Rm* [ah] | shift | Arch. |
|---|---|---|---|---|---|
| | | | | RRX | |
| ARM, halfword | Not available | –255 to 255 | +/–*Rm* | Not available | T2 |
| Thumb 32-bit encoding, word, halfword, or byte | 0 to 255 | Not available | Not available | | T2 |

### Notes about the Architecture column

Entries in the Architecture column indicate that the instructions are available as follows:

**All**
All versions of the ARM architecture.

**T2**
The ARMv6T2 and above architectures.

### Related concepts

*6.16 Address alignment* on page 6-128.

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 10.136 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 Thumb 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 ARM instructions:

- SP can be used but use of SP for any of *Rd*, *Rt*, or *Rt2* is deprecated in ARMv6T2 and above.
- For STREXD, *Rt* must be an even numbered register, and not LR.
- *Rt2* must be *R(t+1)*.
- *offset* is not permitted.

For Thumb instructions:

- SP can be used for R*n*, 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.

### Architectures

ARM STREX is available in ARMv6 and above.

ARM STREXB, STREXD, and STREXH are available in ARMv6K and above.

All these 32-bit Thumb instructions are available in ARMv6T2 and above, except that STREXD is not available in the ARMv7-M architecture.

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

*6.16 Address alignment* on page 6-128.

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 10.137 SUB

Subtract without carry.

### Syntax

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

`SUB{cond} {Rd}, Rn, #imm12 ; Thumb, 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 Thumb 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 Thumb `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 ARM 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 `ARM SUB` instructions are deprecated.

**Note**

The deprecation of SP and PC in ARM instructions is only in ARMv6T2 and above.

### 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 Thumb 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

*10.3 Flexible second operand (Operand2)* on page 10-282.
*10.138 SUBS pc, lr* on page 10-481.
*10.8 Condition code suffixes* on page 10-289.

## 10.138 SUBS pc, lr

Exception return, without popping anything from the stack.

### Syntax

`SUBS{cond} pc, lr, #imm ; ARM and Thumb code`

`MOVS{cond} pc, lr ; ARM and Thumb code`

`op1S{cond} pc, Rn, #imm ; ARM code only and is deprecated`

`op1S{cond} pc, Rn, Rm {, shift} ; ARM code only and is deprecated`

`op2S{cond} pc, #imm ; ARM code only and is deprecated`

`op2S{cond} pc, Rm {, shift} ; ARM 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 Thumb code, it is limited to the range 0-255. In ARM code, it is a flexible second operand.

R*n*
: is the first operand register. ARM deprecates the use of any register except LR.

R*m*
: is the optionally shifted second or only operand 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 ARM, the address written to the PC must be word-aligned.
- For a return to Thumb, 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 Thumb, 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 ARM, only `SUBS{cond} pc, lr, #imm` and `MOVS{cond} pc, lr` are valid instructions. Other instructions are deprecated in ARMv6T2 and above.

**Caution**

Do not use these instructions in User mode or System mode. The assembler cannot warn you about this.

### Architectures

This ARM instruction is available in all versions of the ARM architecture.

This 32-bit Thumb instruction is available in ARMv6T2 and above, except the ARMv7-M architecture.

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

### Related references

*10.14 AND* on page 10-301.
*10.55 MOV* on page 10-370.
*10.3 Flexible second operand (Operand2)* on page 10-282.
*10.10 ADD* on page 10-292.
*10.8 Condition code suffixes* on page 10-289.

## 10.139 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 ARM instruction.
- 0-255 (an 8-bit value) in a Thumb 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 ARM 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.

### Architectures

This ARM instruction is available in all versions of the ARM architecture.

This 16-bit Thumb instruction is available in all T variants of the ARM architecture.

There is no 32-bit version of this instruction in Thumb.

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 10.140 SWP and SWPB

Swap data between registers and memory.

### 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 in ARMv6 and above. You can use `LDREX` and `STREX` instructions to implement more sophisticated semaphores in ARMv6 and above.

### Architectures

These ARM instructions are available in all versions of the ARM architecture.

There are no Thumb `SWP` or `SWPB` instructions.

### Related references

*10.48 LDREX* on page 10-360.
*10.8 Condition code suffixes* on page 10-289.

## 10.141 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 register.

You can use SP in ARM instructions but this is deprecated in ARMv6T2 and above. You cannot use SP in Thumb instructions.

### Condition flags

This instruction does not change the flags.

### Architectures

This ARM instruction is available in ARMv6 and above.

This 32-bit Thumb instruction is available in ARMv6T2 and above. For the ARMv7-M architecture, it is only available in an ARMv7E-M implementation.

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

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 10.142 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 register.

You can use SP in ARM instructions but this is deprecated in ARMv6T2 and above. You cannot use SP in Thumb instructions.

### Condition flags

This instruction does not change the flags.

### Architectures

This ARM instruction is available in ARMv6 and above.

This 32-bit Thumb instruction is available in ARMv6T2 and above. For the ARMv7-M architecture, it is only available in an ARMv7E-M implementation.

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

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 10.143 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 register.

You can use SP in ARM instructions but this is deprecated in ARMv6T2 and above. You cannot use SP in Thumb instructions.

### Condition flags

This instruction does not change the flags.

### Architectures

This ARM instruction is available in ARMv6 and above.

This 32-bit Thumb instruction is available in ARMv6T2 and above. For the ARMv7-M architecture, it is only available in an ARMv7E-M implementation.

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

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 10.144 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 register.

You can use SP in ARM instructions but this is deprecated in ARMv6T2 and above. You cannot use SP in Thumb instructions.

### Condition flags

This instruction does not change the flags.

### 16-bit instructions

The following form of this instruction is available in Thumb code, and is a 16-bit instruction:

`SXTB Rd, Rm`
: *Rd* and *Rm* must both be Lo registers.

### Architectures

This ARM instruction is available in ARMv6 and above.

This 32-bit Thumb instruction is available in ARMv6T2 and above. For the ARMv7-M architecture, it is only available in an ARMv7E-M implementation.

This 16-bit Thumb instruction is available in ARMv6 and above.

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 10.145 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 register.

You can use SP in ARM instructions but this is deprecated in ARMv6T2 and above. You cannot use SP in Thumb instructions.

### Condition flags

This instruction does not change the flags.

### Architectures

This ARM instruction is available in ARMv6 and above.

This 32-bit Thumb instruction is available in ARMv6T2 and above. For the ARMv7-M architecture, it is only available in an ARMv7E-M implementation.

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

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 10.146 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 register.

You can use SP in ARM instructions but this is deprecated in ARMv6T2 and above. You cannot use SP in Thumb instructions.

### Condition flags

This instruction does not change the flags.

### 16-bit instructions

The following form of this instruction is available in Thumb code, and is a 16-bit instruction:

`SXTH Rd, Rm`
: *Rd* and *Rm* must both be Lo registers.

### Architectures

This ARM instruction is available in ARMv6 and above.

This 32-bit Thumb instruction is available in ARMv6T2 and above. For the ARMv7-M architecture, it is only available in an ARMv7E-M implementation.

This 16-bit Thumb instruction is available in ARMv6 and above.

### Example

```
SXTH        r3, r9, r4
```

### Incorrect example

```
SXTH    r9, r3, r2, ROR #12 ; rotation must be by 0, 8, 16, or 24.
```

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 10.147 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 *ARMv7-AR Architecture Reference Manual*. For example:

```
SYS ICIALLUIS ; invalidates all instruction caches Inner Shareable
              ; to Point of Unification and also flushes branch
              ; target cache.
```

### Architectures

The `SYS` pseudo-instruction is available in ARMv7-R in ARM and 32-bit Thumb code.

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

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 10.148 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 Thumb instructions are available in ARMv6T2 and above.

There are no versions of these instructions in ARM or in 16-bit Thumb encodings.

### Related concepts

*6.16 Address alignment* on page 6-128.

## 10.149 TEQ

Test Equivalence.

### Syntax

`TEQ{cond} Rn, Operand2`

where:

*cond*
: is an optional condition code.

*Rn*
: is the ARM 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 Thumb instruction, you cannot use SP or PC for *Rn* or *Operand2*.

In this ARM instruction, use of SP or PC is deprecated in ARMv6T2 and above.

For ARM 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 ARM instruction is available in all architectures that support the ARM instruction set.

The `TEQ` Thumb instruction is available in ARMv6T2 and above.

### Correct example

```
    TEQEQ   r10, r9
```

### Incorrect example

```
    TEQ     pc, r1, ROR r0      ; PC not permitted with register
                                ; controlled shift
```

### Related references

*10.3 Flexible second operand (Operand2)* on page 10-282.
*10.8 Condition code suffixes* on page 10-289.

## 10.150 TST

Test bits.

### Syntax

`TST{cond} Rn, Operand2`

where:

*cond*
: is an optional condition code.

*Rn*
: is the ARM 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 Thumb instruction, you cannot use SP or PC for *Rn* or *Operand2*.

In this ARM instruction, use of SP or PC is deprecated in ARMv6T2 and above.

For ARM 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 Thumb code, and is a 16-bit instruction:

`TST Rn, Rm`
: *Rn* and *Rm* must both be Lo registers.

### Architectures

This ARM instruction is available in all architectures that support the ARM instruction set.

The `TST` Thumb instruction is available in all architectures that support the Thumb instruction set.

### Examples

```
    TST     r0, #0x3F8
    TSTNE   r1, r5, ASR r1
```

### Related references

*10.3 Flexible second operand (Operand2)* on page 10-282.
*10.8 Condition code suffixes* on page 10-289.

## 10.151 UADD8

Unsigned parallel byte-wise addition.

### Syntax

`UADD8{cond} {Rd}, Rn, Rm`

where:

*cond*
: is an optional condition code.

*Rd*
: is the destination register.

*Rm, Rn*
: are the ARM 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 register.

You can use SP in ARM instructions but this is deprecated in ARMv6T2 and above. You cannot use SP in Thumb 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.

### Architectures

This ARM instruction is available in ARMv6 and above.

This 32-bit Thumb instruction is available in ARMv6T2 and above. For the ARMv7-M architecture, it is only available in an ARMv7E-M implementation.

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

### Related references

*10.99 SEL* on page 10-430.
*10.8 Condition code suffixes* on page 10-289.

## 10.152 UADD16

Unsigned parallel halfword-wise addition.

### Syntax

`UADD16{cond} {Rd}, Rn, Rm`

where:

*cond*
: is an optional condition code.

*Rd*
: is the destination register.

*Rm, Rn*
: are the ARM 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 register.

You can use SP in ARM instructions but this is deprecated in ARMv6T2 and above. You cannot use SP in Thumb 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.

### Architectures

This ARM instruction is available in ARMv6 and above.

This 32-bit Thumb instruction is available in ARMv6T2 and above. For the ARMv7-M architecture, it is only available in an ARMv7E-M implementation.

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

### Related references

*10.99 SEL* on page 10-430.
*10.8 Condition code suffixes* on page 10-289.

## 10.153 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 register.

*Rm, Rn*
: are the ARM 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 register.

You can use SP in ARM instructions but this is deprecated in ARMv6T2 and above. You cannot use SP in Thumb 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.

### Architectures

This ARM instruction is available in ARMv6 and above.

This 32-bit Thumb instruction is available in ARMv6T2 and above. For the ARMv7-M architecture, it is only available in an ARMv7E-M implementation.

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

### Related references

*10.99 SEL* on page 10-430.

*10.8 Condition code suffixes* on page 10-289.

## 10.154 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 register.

You can use SP in the ARM instruction but this is deprecated in ARMv6T2 and above. You cannot use SP in the Thumb instruction.

### Condition flags

This instruction does not alter any flags.

### Architectures

This ARM instruction is available in ARMv6T2 and above.

This 32-bit Thumb instruction is available in ARMv6T2 and above.

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

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 10.155 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 Thumb instruction is available in ARMv7-R and ARMv7-M.

This ARM instruction is optional in ARMv7-R.

There is no 16-bit Thumb `UDIV` instruction.

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 10.156 UHADD8

Unsigned halving parallel byte-wise addition.

### Syntax

`UHADD8{cond} {Rd}, Rn, Rm`

where:

*cond*
: is an optional condition code.

*Rd*
: is the destination register.

*Rm, Rn*
: are the ARM 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 register.

You can use SP in ARM instructions but this is deprecated in ARMv6T2 and above. You cannot use SP in Thumb instructions.

### Condition flags

This instruction does not affect the N, Z, C, V, Q, or GE flags.

### Architectures

This ARM instruction is available in ARMv6 and above.

This 32-bit Thumb instruction is available in ARMv6T2 and above. For the ARMv7-M architecture, it is only available in an ARMv7E-M implementation.

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

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 10.157 UHADD16

Unsigned halving parallel halfword-wise addition.

### Syntax

`UHADD16{cond} {Rd}, Rn, Rm`

where:

*cond*
: is an optional condition code.

*Rd*
: is the destination register.

*Rm, Rn*
: are the ARM 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 register.

You can use SP in ARM instructions but this is deprecated in ARMv6T2 and above. You cannot use SP in Thumb instructions.

### Condition flags

This instruction does not affect the N, Z, C, V, Q, or GE flags.

### Architectures

This ARM instruction is available in ARMv6 and above.

This 32-bit Thumb instruction is available in ARMv6T2 and above. For the ARMv7-M architecture, it is only available in an ARMv7E-M implementation.

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

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 10.158 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 register.

*Rm, Rn*
: are the ARM 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 register.

You can use SP in ARM instructions but this is deprecated in ARMv6T2 and above. You cannot use SP in Thumb instructions.

### Condition flags

This instruction does not affect the N, Z, C, V, Q, or GE flags.

### Architectures

This ARM instruction is available in ARMv6 and above.

This 32-bit Thumb instruction is available in ARMv6T2 and above. For the ARMv7-M architecture, it is only available in an ARMv7E-M implementation.

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

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 10.159 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 register.

*Rm, Rn*
: are the ARM 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 register.

You can use SP in ARM instructions but this is deprecated in ARMv6T2 and above. You cannot use SP in Thumb instructions.

### Condition flags

This instruction does not affect the N, Z, C, V, Q, or GE flags.

### Architectures

This ARM instruction is available in ARMv6 and above.

This 32-bit Thumb instruction is available in ARMv6T2 and above. For the ARMv7-M architecture, it is only available in an ARMv7E-M implementation.

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

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 10.160 UHSUB8

Unsigned halving parallel byte-wise subtraction.

### Syntax

`UHSUB8{cond} {Rd}, Rn, Rm`

where:

*cond*
: is an optional condition code.

*Rd*
: is the destination register.

*Rm, Rn*
: are the ARM 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 register.

You can use SP in ARM instructions but this is deprecated in ARMv6T2 and above. You cannot use SP in Thumb instructions.

### Condition flags

This instruction does not affect the N, Z, C, V, Q, or GE flags.

### Architectures

This ARM instruction is available in ARMv6 and above.

This 32-bit Thumb instruction is available in ARMv6T2 and above. For the ARMv7-M architecture, it is only available in an ARMv7E-M implementation.

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

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 10.161 UHSUB16

Unsigned halving parallel halfword-wise subtraction.

### Syntax

`UHSUB16{cond} {Rd}, Rn, Rm`

where:

*cond*
: is an optional condition code.

*Rd*
: is the destination register.

*Rm, Rn*
: are the ARM 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 register.

You can use SP in ARM instructions but this is deprecated in ARMv6T2 and above. You cannot use SP in Thumb instructions.

### Condition flags

This instruction does not affect the N, Z, C, V, Q, or GE flags.

### Architectures

This ARM instruction is available in ARMv6 and above.

This 32-bit Thumb instruction is available in ARMv6T2 and above. For the ARMv7-M architecture, it is only available in an ARMv7E-M implementation.

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

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 10.162 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 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 register.

You can use SP in ARM instructions but this is deprecated in ARMv6T2 and above. You cannot use SP in Thumb instructions.

### Condition flags

This instruction does not change the flags.

### Architectures

This ARM instruction is available in ARMv6 and above.

This 32-bit Thumb instruction is available in ARMv6T2 and above. For the ARMv7-M architecture, it is only available in an ARMv7E-M implementation.

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

### Examples

```
UMAAL       r8, r9, r2, r3
UMAALGE     r2, r0, r5, r3
```

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 10.163 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 ARM 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 ARM 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

*`Rn`* must be different from *`RdLo`* and *`RdHi`* in architectures before ARMv6.

You cannot use PC for any register.

You can use SP in ARM instructions but this is deprecated in ARMv6T2 and above. You cannot use SP in Thumb 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 ARM instruction is available in all versions of the ARM architecture.

This 32-bit Thumb instruction is available in ARMv6T2 and above.

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

### Example

```
    UMLALS      r4, r5, r3, r8
```

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 10.164 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 ARM 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. *`RdLo`* and *`RdHi`* must be different registers.

*`Rn, Rm`*
: are ARM 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

*`Rn`* must be different from *`RdLo`* and *`RdHi`* in architectures before ARMv6.

You cannot use PC for any register.

You can use SP in ARM instructions but this is deprecated in ARMv6T2 and above. You cannot use SP in Thumb 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 ARM instruction is available in all versions of the ARM architecture.

This 32-bit Thumb instruction is available in ARMv6T2 and above.

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

### Example

```
    UMULL       r0, r4, r5, r6
```

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 10.165 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 10-18 Range and encoding of expr**

| Instruction | Encoding | Number of bits for *expr* | Range |
|---|---|---|---|
| ARM | 0xV7FYYYFY | 16 | 0-65535 |
| Thumb 32-bit encoding | 0xF7FYAYFY | 12 | 0-4095 |
| Thumb16-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 Thumb code

You can use the `.W` width specifier to force `UND` to generate a 32-bit instruction in Thumb code on ARMv6T2 and above processors. `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

*10.8 Condition code suffixes* on page 10-289.

## 10.166 UQADD8

Unsigned saturating parallel byte-wise addition.

### Syntax

`UQADD8{cond} {Rd}, Rn, Rm`

where:

*cond*
: is an optional condition code.

*Rd*
: is the destination register.

*Rm, Rn*
: are the ARM registers holding the operands.

### Operation

This instruction performs four unsigned integer additions on the corresponding bytes of the operands and writes the results into the corresponding bytes of the destination. It saturates the results to the unsigned range $0 \leq x \leq 2^8 - 1$. The Q flag is not affected even if this operation saturates.

### Register restrictions

You cannot use PC for any register.

You can use SP in ARM instructions but this is deprecated in ARMv6T2 and above. You cannot use SP in Thumb instructions.

### Condition flags

This instruction does not affect the N, Z, C, V, Q, or GE flags.

### Architectures

This ARM instruction is available in ARMv6 and above.

This 32-bit Thumb instruction is available in ARMv6T2 and above. For the ARMv7-M architecture, it is only available in an ARMv7E-M implementation.

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

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 10.167 UQADD16

Unsigned saturating parallel halfword-wise addition.

### Syntax

`UQADD16{cond} {Rd}, Rn, Rm`

where:

*cond*
: is an optional condition code.

*Rd*
: is the destination register.

*Rm, Rn*
: are the ARM registers holding the operands.

### Operation

This instruction performs two unsigned integer additions on the corresponding halfwords of the operands and writes the results into the corresponding halfwords of the destination. It saturates the results to the unsigned range $0 \le x \le 2^{16} - 1$. The Q flag is not affected even if this operation saturates.

### Register restrictions

You cannot use PC for any register.

You can use SP in ARM instructions but this is deprecated in ARMv6T2 and above. You cannot use SP in Thumb instructions.

### Condition flags

This instruction does not affect the N, Z, C, V, Q, or GE flags.

### Architectures

This ARM instruction is available in ARMv6 and above.

This 32-bit Thumb instruction is available in ARMv6T2 and above. For the ARMv7-M architecture, it is only available in an ARMv7E-M implementation.

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

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 10.168 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 register.

*Rm, Rn*
: are the ARM 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 $0 \le x \le 2^{16} - 1$. The Q flag is not affected even if this operation saturates.

### Register restrictions

You cannot use PC for any register.

You can use SP in ARM instructions but this is deprecated in ARMv6T2 and above. You cannot use SP in Thumb instructions.

### Condition flags

This instruction does not affect the N, Z, C, V, Q, or GE flags.

### Architectures

This ARM instruction is available in ARMv6 and above.

This 32-bit Thumb instruction is available in ARMv6T2 and above. For the ARMv7-M architecture, it is only available in an ARMv7E-M implementation.

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

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 10.169 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 register.

*Rm, Rn*
: are the ARM 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 $0 \le x \le 2^{16} - 1$. The Q flag is not affected even if this operation saturates.

### Register restrictions

You cannot use PC for any register.

You can use SP in ARM instructions but this is deprecated in ARMv6T2 and above. You cannot use SP in Thumb instructions.

### Condition flags

This instruction does not affect the N, Z, C, V, Q, or GE flags.

### Architectures

This ARM instruction is available in ARMv6 and above.

This 32-bit Thumb instruction is available in ARMv6T2 and above. For the ARMv7-M architecture, it is only available in an ARMv7E-M implementation.

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

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 10.170 UQSUB8

Unsigned saturating parallel byte-wise subtraction.

### Syntax

`UQSUB8{cond} {Rd}, Rn, Rm`

where:

*cond*
: is an optional condition code.

*Rd*
: is the destination register.

*Rm, Rn*
: are the ARM 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 $0 \le x \le 2^8 - 1$. The Q flag is not affected even if this operation saturates.

### Register restrictions

You cannot use PC for any register.

You can use SP in ARM instructions but this is deprecated in ARMv6T2 and above. You cannot use SP in Thumb instructions.

### Condition flags

This instruction does not affect the N, Z, C, V, Q, or GE flags.

### Architectures

This ARM instruction is available in ARMv6 and above.

This 32-bit Thumb instruction is available in ARMv6T2 and above. For the ARMv7-M architecture, it is only available in an ARMv7E-M implementation.

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

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 10.171 UQSUB16

Unsigned saturating parallel halfword-wise subtraction.

### Syntax

`UQSUB16{cond} {Rd}, Rn, Rm`

where:

*cond*
: is an optional condition code.

*Rd*
: is the destination register.

*Rm, Rn*
: are the ARM registers holding the operands.

### Operation

This instruction subtracts each halfword of the second operand from the corresponding halfword of the first operand and writes the results into the corresponding halfwords of the destination. It saturates the results to the unsigned range $0 \le x \le 2^{16} - 1$. The Q flag is not affected even if this operation saturates.

### Register restrictions

You cannot use PC for any register.

You can use SP in ARM instructions but this is deprecated in ARMv6T2 and above. You cannot use SP in Thumb instructions.

### Condition flags

This instruction does not affect the N, Z, C, V, Q, or GE flags.

### Architectures

This ARM instruction is available in ARMv6 and above.

This 32-bit Thumb instruction is available in ARMv6T2 and above. For the ARMv7-M architecture, it is only available in an ARMv7E-M implementation.

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

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 10.172 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 register.

You can use SP in ARM instructions but this is deprecated in ARMv6T2 and above. You cannot use SP in Thumb instructions.

### Condition flags

This instruction does not alter any flags.

### Architectures

This ARM instruction is available in ARMv6 and above.

This 32-bit Thumb instruction is available in ARMv6T2 and above. For the ARMv7-M architecture, it is only available in an ARMv7E-M implementation.

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

### Example

```
    USAD8       r2, r4, r6
```

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 10.173 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 register.

You can use SP in ARM instructions but this is deprecated in ARMv6T2 and above. You cannot use SP in Thumb instructions.

### Condition flags

This instruction does not alter any flags.

### Architectures

This ARM instruction is available in ARMv6 and above.

This 32-bit Thumb instruction is available in ARMv6T2 and above. For the ARMv7-M architecture, it is only available in an ARMv7E-M implementation.

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

### 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

*10.8 Condition code suffixes* on page 10-289.

## 10.174 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 (ARM) or 1-31 (Thumb).

  `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 register.

You can use SP in ARM instructions but this is deprecated in ARMv6T2 and above. You cannot use SP in Thumb 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 ARM instruction is available in ARMv6 and above.

This 32-bit Thumb instruction is available in ARMv6T2 and above.

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

### Example

```
USATNE  r0, #7, r5
```

### Related references

*10.127 SSAT16* on page 10-461.
*10.60 MRS (PSR to general-purpose register)* on page 10-376.
*10.8 Condition code suffixes* on page 10-289.

## 10.175 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 $0 \le x \le 2^{sat}-1$.

### Register restrictions

You cannot use PC for any register.

You can use SP in ARM instructions but this is deprecated in ARMv6T2 and above. You cannot use SP in Thumb 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.

### Architectures

This ARM instruction is available in ARMv6 and above.

This 32-bit Thumb instruction is available in ARMv6T2 and above. For the ARMv7-M architecture, it is only available in an ARMv7E-M implementation.

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

### Example

```
    USAT16  r0, #7, r5
```

### Related references

*10.60 MRS (PSR to general-purpose register)* on page 10-376.
*10.8 Condition code suffixes* on page 10-289.

## 10.176 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 register.

*Rm, Rn*
: are the ARM 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 register.

You can use SP in ARM instructions but this is deprecated in ARMv6T2 and above. You cannot use SP in Thumb 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.

### Architectures

This ARM instruction is available in ARMv6 and above.

This 32-bit Thumb instruction is available in ARMv6T2 and above. For the ARMv7-M architecture, it is only available in an ARMv7E-M implementation.

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

### Related references

*10.99 SEL* on page 10-430.

*10.8 Condition code suffixes* on page 10-289.

## 10.177 USUB8

Unsigned parallel byte-wise subtraction.

### Syntax

`USUB8{cond} {Rd}, Rn, Rm`

where:

*cond*
: is an optional condition code.

*Rd*
: is the destination register.

*Rm, Rn*
: are the ARM 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 register.

You can use SP in ARM instructions but this is deprecated in ARMv6T2 and above. You cannot use SP in Thumb 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.

### Architectures

This ARM instruction is available in ARMv6 and above.

This 32-bit Thumb instruction is available in ARMv6T2 and above. For the ARMv7-M architecture, it is only available in an ARMv7E-M implementation.

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

### Related references

*10.99 SEL* on page 10-430.
*10.8 Condition code suffixes* on page 10-289.

## 10.178 USUB16

Unsigned parallel halfword-wise subtraction.

### Syntax

`USUB16{cond} {Rd}, Rn, Rm`

where:

*cond*
: is an optional condition code.

*Rd*
: is the destination register.

*Rm, Rn*
: are the ARM 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 register.

You can use SP in ARM instructions but this is deprecated in ARMv6T2 and above. You cannot use SP in Thumb 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.

### Architectures

This ARM instruction is available in ARMv6 and above.

This 32-bit Thumb instruction is available in ARMv6T2 and above. For the ARMv7-M architecture, it is only available in an ARMv7E-M implementation.

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

### Related references

*10.99 SEL* on page 10-430.
*10.8 Condition code suffixes* on page 10-289.

## 10.179 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 register.

You can use SP in ARM instructions but this is deprecated in ARMv6T2 and above. You cannot use SP in Thumb instructions.

### Condition flags

This instruction does not change the flags.

### Architectures

This ARM instruction is available in ARMv6 and above.

This 32-bit Thumb instruction is available in ARMv6T2 and above. For the ARMv7-M architecture, it are only available in an ARMv7E-M implementation.

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

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 10.180 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 register.

You can use SP in ARM instructions but this is deprecated in ARMv6T2 and above. You cannot use SP in Thumb instructions.

### Condition flags

This instruction does not change the flags.

### Architectures

This ARM instruction is available in ARMv6 and above.

This 32-bit Thumb instruction is available in ARMv6T2 and above. For the ARMv7-M architecture, it is only available in an ARMv7E-M implementation.

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

### Example

```
    UXTAB16EQ    r0, r0, r4, ROR #16
```

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 10.181 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 register.

You can use SP in ARM instructions but this is deprecated in ARMv6T2 and above. You cannot use SP in Thumb instructions.

### Condition flags

This instruction does not change the flags.

### Architectures

This ARM instruction is available in ARMv6 and above.

This 32-bit Thumb instruction is available in ARMv6T2 and above. For the ARMv7-M architecture, it is only available in an ARMv7E-M implementation.

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

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 10.182 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 register.

You can use SP in ARM instructions but this is deprecated in ARMv6T2 and above. You cannot use SP in Thumb instructions.

### Condition flags

This instruction does not change the flags.

### 16-bit instruction

The following form of this instruction is available in Thumb code, and is a 16-bit instruction:

`UXTB Rd, Rm`
: *Rd* and *Rm* must both be Lo registers.

### Architectures

This ARM instruction is available in ARMv6 and above.

This 32-bit Thumb instruction is available in ARMv6T2 and above. For the ARMv7-M architecture, it is only available in an ARMv7E-M implementation.

This 16-bit Thumb instruction is available in ARMv6 and above.

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 10.183 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 register.

You can use SP in ARM instructions but this is deprecated in ARMv6T2 and above. You cannot use SP in Thumb instructions.

### Condition flags

This instruction does not change the flags.

### Architectures

This ARM instruction is available in ARMv6 and above.

This 32-bit Thumb instruction is available in ARMv6T2 and above. For the ARMv7-M architecture, it is only available in an ARMv7E-M implementation.

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

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 10.184 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 register.

You can use SP in ARM instructions but this is deprecated in ARMv6T2 and above. You cannot use SP in Thumb instructions.

### Condition flags

This instruction does not change the flags.

### 16-bit instructions

The following form of this instruction is available in Thumb code, and is a 16-bit instruction:

`UXTH Rd, Rm`
: *Rd* and *Rm* must both be Lo registers.

### Architectures

This ARM instruction is available in ARMv6 and above.

This 32-bit Thumb instruction is available in ARMv6T2 and above. For the ARMv7-M architecture, it is only available in an ARMv7E-M implementation.

This 16-bit Thumb instruction is available in ARMv6 and above.

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 10.185 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.

`WFE` executes as a `NOP` instruction in ARMv6T2.

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.

If the Event Register is set, `WFE` clears it and returns immediately.

If `WFE` is implemented, `SEV` must also be implemented.

### Architectures

This ARM instruction is available in ARMv6K and above.

This 32-bit Thumb instruction is available in ARMv6T2 and above.

This 16-bit Thumb instruction is available in ARMv6T2 and above.

### Related references

*10.67 NOP* on page 10-387.
*10.8 Condition code suffixes* on page 10-289.
*10.101 SEV* on page 10-433.
*10.186 WFI* on page 10-535.

## 10.186 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` executes as a `NOP` instruction in ARMv6T2.

`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.

### Architectures

This ARM instruction is available in ARMv6K and above.

This 32-bit Thumb instruction is available in ARMv6T2 and above.

This 16-bit Thumb instruction is available in ARMv6T2 and above.

### Related references

*10.67 NOP* on page 10-387.
*10.8 Condition code suffixes* on page 10-289.
*10.185 WFE* on page 10-534.

## 10.187 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` executes as a `NOP` instruction in ARMv6T2.

`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.

### Architectures

This ARM instruction is available in ARMv6K and above.

This 32-bit Thumb instruction is available in ARMv6T2 and above.

This 16-bit Thumb instruction is available in ARMv6T2 and above.

### Related references

*10.67 NOP* on page 10-387.
*10.8 Condition code suffixes* on page 10-289.

# Chapter 11
# VFP Instructions

Describes the assembly programming of the VFP hardware.

It contains the following sections:

## 11.1 Summary of VFP instructions

This table provides a summary of the VFP instructions and the VFP architectures that support them.

**Table 11-1 Summary of VFP instructions**

| Mnemonic | Brief description | Arch. |
|---|---|---|
| VABS | Absolute value | All |
| VADD | Add | All |
| VCMP, VCMPE | Compare | All |
| VCVT | Convert between single-precision and double-precision | All |
| | Convert between floating-point and integer | All |
| | Convert between floating-point and fixed-point | VFPv3, VFPv4 |
| VCVTB, VCVTT | Convert between half-precision and single-precision floating-point | Half-precision, VFPv4 |
| VDIV | Divide | All |
| VFMA, VFMS | Fused multiply accumulate, Fused multiply subtract | VFPv4 |
| VFNMA, VFNMS | Fused multiply accumulate with negation, Fused multiply subtract with negation | VFPv4 |
| VLDM | Load multiple | All |
| VLDR | Load (see also VLDR pseudo-instruction) | All |
| | Load (post-increment and pre-decrement) | All |
| VMLA | Multiply accumulate | All |
| VMLS | Multiply subtract | All |
| VMOV | Insert floating-point immediate in single-precision or double-precision register | VFPv3, VFPv4 |
| | Transfer from one single-precision or double-precision register to another | All |
| | Transfer from single-precision to ARM register | All |
| | Transfer from ARM register to single-precision | All |
| | Transfer from two ARM registers to two single-precision or one double-precision register | All |
| | Transfer from two single-precision registers or one double-precision register to two ARM registers | All |
| | Transfer from ARM register to half a double-precision register | All |
| | Transfer from half a double-precision register to ARM register | All |
| VMRS | Transfer from VFP system register to ARM register | All |
| VMSR | Transfer from ARM register to VFP system register | All |
| VMUL | Multiply | All |
| VNEG | Negate | All |
| VNMLA | Negated multiply accumulate | All |
| VNMLS | Negated multiply subtract | All |
| VNMUL | Negated multiply | All |
| VPOP | Pop VFP registers from full-descending stack | All |

**Table 11-1 Summary of VFP instructions (continued)**

| Mnemonic | Brief description | Arch. |
|---|---|---|
| VPUSH | Push VFP registers to full-descending stack | All |
| VSQRT | Square Root | All |
| VSTM | Store multiple | All |
| VSTR | Store | All |
| | Store (post-increment and pre-decrement) | All |
| VSUB | Subtract | All |

## 11.2 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

*10.8 Condition code suffixes* on page 10-289.

## 11.3 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

*10.8 Condition code suffixes* on page 10-289.

## 11.4 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

*10.8 Condition code suffixes* on page 10-289.

## 11.5 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

*10.8 Condition code suffixes* on page 10-289.

## 11.6 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

*10.8 Condition code suffixes* on page 10-289.

## 11.7 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

*10.8 Condition code suffixes* on page 10-289.

## 11.8 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

*10.8 Condition code suffixes* on page 10-289.

## 11.9 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

*10.8 Condition code suffixes* on page 10-289.

## 11.10 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

*11.23 VMUL (floating-point)* on page 11-562.
*10.8 Condition code suffixes* on page 10-289.

## 11.11 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 ARM 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

*4.15 Stack implementation using LDM and STM* on page 4-77.

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 11.12 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 ARM 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

*7.5 Register-relative and PC-relative expressions* on page 7-136.

### Related references

*11.14 VLDR pseudo-instruction* on page 11-553.
*10.8 Condition code suffixes* on page 10-289.

## 11.13 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 ARM 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

*11.11 VLDM (floating-point)* on page 11-550.
*11.12 VLDR (floating-point)* on page 11-551.
*10.8 Condition code suffixes* on page 10-289.

## 11.14 VLDR pseudo-instruction

Pseudo-instruction that loads a constant value into a VFP single-precision or double-precision register.

**Note**

This section describes the `VLDR` pseudo-instruction only.

### Syntax

`VLDR{`*cond*`}.`*datatype* *Dd*`,=`*constant*

`VLDR{`*cond*`}.`*datatype* *Sd*`,=`*constant*

where:

*datatype*
: must be either `F32` or `F64`.

*n*
: must be one of 8, 16, 32, or 64.

*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 *datatype*.

### Operation

If an instruction (for example, `VMOV`) is available that can generate the constant directly into the register, the assembler uses it. Otherwise, the assembler generates a doubleword literal pool entry containing the constant and loads the constant using a `VLDR` instruction.

### Related concepts

*8.9 VFP data types* on page 8-172.

### Related references

*11.12 VLDR (floating-point)* on page 11-551.
*10.8 Condition code suffixes* on page 10-289.

## 11.15 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

*10.8 Condition code suffixes* on page 10-289.

## 11.16 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

*10.8 Condition code suffixes* on page 10-289.

## 11.17 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 +/–*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

*10.8 Condition code suffixes* on page 10-289.

## 11.18 VMOV (between one ARM register and single precision VFP)

Transfer contents between a single-precision floating-point register and an ARM register.

### Syntax

`VMOV{cond} Rd, Sn`

`VMOV{cond} Sn, Rd`

where:

*cond*
: is an optional condition code.

*Sn*
: is the VFP single-precision register.

*Rd*
: is the ARM 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

*10.8 Condition code suffixes* on page 10-289.

## 11.19 VMOV (between two ARM registers and one or two extension registers)

Transfer contents between two ARM 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 ARM 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

*10.8 Condition code suffixes* on page 10-289.

## 11.20 VMOV (between an ARM register and half a double precision VFP register)

Transfer contents between an ARM register and half a double precision VFP register.

### Syntax

`VMOV{cond}{.size} Dn[x], Rd`

`VMOV{cond}{.size} Rd, Dn[x]`

where:

*cond*
: is an optional condition code.

*size*
: is the data size. Must be either `32` or omitted. If omitted, *size* is `32`.

*Dn*`[x]`
: if *x* is 0, is the least significant half, or if *x* is 1, the most significant half of a double precision VFP register.

*Rd*
: is the ARM 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*.

## 11.21 VMRS

Transfer the contents of a VFP system register to an ARM register.

### Syntax

`VMRS{cond} Rd, extsysreg`

where:

*cond*
: is an optional condition code.

*extsysreg*
: is the VFP system register, usually `FPSCR`, `FPSID`, or `FPEXC`.

*Rd*
: is the ARM 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 ARM APSR.

### Operation

The `VMRS` instruction transfers the contents of *extsysreg* into *Rd*.

**Note**

This instruction stalls the processor until all current VFP operations complete.

### Examples

```
VMRS    r2,FPCID
VMRS    APSR_nzcv, FPSCR  ; transfer FP status register to ARM APSR
```

### Related references

*8.11 VFP system registers* on page 8-174.
*10.8 Condition code suffixes* on page 10-289.

## 11.22 VMSR

Transfer the contents of an ARM register to a VFP system register.

### Syntax

`VMSR{cond} extsysreg, Rd`

where:

*cond*
: is an optional condition code.

*extsysreg*
: is the VFP system register, usually `FPSCR`, `FPSID`, or `FPEXC`.

*Rd*
: is the ARM register. *Rd* must not be PC.

### Operation

The `VMSR` instruction transfers the contents of *Rd* into *extsysreg*.

**Note**

This instruction stalls the processor until all current VFP operations complete.

### Example

```
    VMSR    FPSCR, r4
```

### Related references

*8.11 VFP system registers* on page 8-174.
*10.8 Condition code suffixes* on page 10-289.

## 11.23 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

*10.8 Condition code suffixes* on page 10-289.

## 11.24 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

*10.8 Condition code suffixes* on page 10-289.

## 11.25 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

*10.8 Condition code suffixes* on page 10-289.

## 11.26 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

*10.8 Condition code suffixes* on page 10-289.

## 11.27 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

*10.8 Condition code suffixes* on page 10-289.

## 11.28 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

*4.15 Stack implementation using LDM and STM* on page 4-77.

### Related references

*10.8 Condition code suffixes* on page 10-289.
*11.29 VPUSH (floating-point)* on page 11-568.

## 11.29 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

*4.15 Stack implementation using LDM and STM* on page 4-77.

### Related references

*10.8 Condition code suffixes* on page 10-289.
*11.28 VPOP (floating-point)* on page 11-567.

## 11.30 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

*10.8 Condition code suffixes* on page 10-289.

## 11.31 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 ARM 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

*4.15 Stack implementation using LDM and STM* on page 4-77.

### Related references

*10.8 Condition code suffixes* on page 10-289.

## 11.32 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 ARM 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

*7.5 Register-relative and PC-relative expressions* on page 7-136.

### Related references

*11.14 VLDR pseudo-instruction* on page 11-553.
*10.8 Condition code suffixes* on page 10-289.

## 11.33 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 ARM 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

*11.32 VSTR (floating-point)* on page 11-571.
*11.31 VSTM (floating-point)* on page 11-570.
*10.8 Condition code suffixes* on page 10-289.

## 11.34 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

*10.8 Condition code suffixes* on page 10-289.

# Chapter 12
# Directives Reference

Describes the directives that are provided by the ARM assembler, `armasm`.

**Note**

None of these directives are available in the inline assemblers in the ARM C and C++ compilers.

It contains the following sections:

## 12.1 Alphabetical list of directives

The ARM assembler, armasm, provides various directives.

The following table lists them:

**Table 12-1 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 | QN |
| 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 | THUMBX |
| ENDFUNC or ENDP | INFO | TTL |
| ENDIF (see IF) | KEEP | WHILE and WEND |
| ENTRY | LCLA, LCLL, and LCLS | |

## 12.2 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

*12.52 MACRO and MEND* on page 12-633.
*12.54 MEXIT* on page 12-637.
*12.45 IF, ELSE, ENDIF, and ELIF* on page 12-624.
*12.68 WHILE and WEND* on page 12-652.

## 12.3 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

*12.31 FRAME ADDRESS* on page 12-610.
*12.32 FRAME POP* on page 12-611.
*12.33 FRAME PUSH* on page 12-612.
*12.34 FRAME REGISTER* on page 12-613.
*12.35 FRAME RESTORE* on page 12-614.
*12.36 FRAME RETURN ADDRESS* on page 12-615.
*12.37 FRAME SAVE* on page 12-616.
*12.38 FRAME STATE REMEMBER* on page 12-617.
*12.39 FRAME STATE RESTORE* on page 12-618.
*12.40 FRAME UNWIND ON* on page 12-619.
*12.41 FRAME UNWIND OFF* on page 12-620.
*12.42 FUNCTION or PROC* on page 12-621.
*12.25 ENDFUNC or ENDP* on page 12-603.

## 12.4 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

*12.28 EXPORT or GLOBAL* on page 12-606.

## 12.5 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 ARM or Thumb 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 Thumb code, or words in ARM 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 Thumb pseudo-instruction can only load addresses that are word aligned, but a label within Thumb code might not be word aligned. Use ALIGN 4 to ensure four-byte alignment of an address within Thumb 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.
- In ARMv5TE, or in ARMv6 when SCTLR.U is 0, LDRD and STRD doubleword data transfers must be eight-byte aligned. Use ALIGN 8 before memory allocation directives such as DCQ if the data is to be accessed using LDRD or STRD. This is not required in ARMv6 when SCTLR.U is 1, or in ARMv7, because in these versions, doubleword data transfers can be word-aligned.
- A label on a line by itself can be arbitrarily aligned. Following ARM code is word-aligned (Thumb code is halfword aligned). The label therefore does not address the code correctly. Use ALIGN 4 (or ALIGN 2 for Thumb) 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

*12.6 AREA* on page 12-582.

## 12.6 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 ARM code sections.

Do not use `ALIGN=0` for Thumb 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 ARM or Thumb 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 ARM or Thumb 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

*3.3 ELF sections and the AREA directive* on page 3-54.

### Related references

*12.5 ALIGN* on page 12-580.
*12.57 RELOC* on page 12-641.
*12.16 DCD and DCDU* on page 12-594.

### Related information

*Execute-only memory.*
*Building applications for execute-only memory.*
*Information about image structure and generation.*

## 12.7 ARM or CODE32

The ARM directive instructs the assembler to interpret subsequent instructions as ARM instructions, using either the UAL or the pre-UAL ARM assembly language syntax. CODE32 is a synonym for ARM.

### Syntax

```
ARM
```

### Usage

In files that contain code using different instruction sets, ARM must precede any ARM 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 the assembler to assemble ARM instructions, and inserts padding if necessary.

### ARM and THUMB directives

This example shows how you can use ARM and THUMB directives to switch state and assemble both ARM and Thumb instructions in a single area.

```
     AREA ToThumb, CODE, READONLY  ; Name this block of code
     ENTRY                         ; Mark first instruction to execute
     ARM                           ; Subsequent instructions are ARM
start
     ADR     r0, into_thumb + 1    ; Processor starts in ARM state
     BX      r0                    ; Inline switch to Thumb state
     THUMB                         ; Subsequent instructions are Thumb
into_thumb
     MOVS    r0, #10               ; New-style Thumb instructions
```

### Related references

*12.11 CODE16* on page 12-589.
*12.65 THUMB* on page 12-649.
*12.66 THUMBX* on page 12-650.

## 12.8 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

*12.48 INFO* on page 12-629.

## 12.9 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.*

## 12.10 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

*2.10 Predeclared core register names* on page 2-40.
*2.11 Predeclared extension register names* on page 2-41.

## 12.11 CODE16

The `CODE16` directive instructs the assembler to interpret subsequent instructions as Thumb instructions, using the pre-UAL assembly language syntax.

### Syntax

```
CODE16
```

### Usage

In files that contain code using different instruction sets, `CODE16` must precede Thumb 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 the assembler to assemble Thumb instructions, and inserts padding if necessary.

### Related references

*12.7 ARM or CODE32* on page 12-585.
*12.65 THUMB* on page 12-649.
*12.66 THUMBX* on page 12-650.

## 12.12 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
```

## 12.13 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

*2.10 Predeclared core register names* on page 2-40.
*2.11 Predeclared extension register names* on page 2-41.

## 12.14 DATA

The DATA directive is no longer required. It is ignored by the assembler.

## 12.15 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, ARM 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

*7.14 Numeric expressions* on page 7-145.

### Related references

*12.16 DCD and DCDU* on page 12-594.
*12.21 DCQ and DCQU* on page 12-599.
*12.22 DCW and DCWU* on page 12-600.
*12.64 SPACE or FILL* on page 12-648.
*12.5 ALIGN* on page 12-580.

## 12.16 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

*7.14 Numeric expressions* on page 7-145.

### Related references

*12.15 DCB* on page 12-593.
*12.21 DCQ and DCQU* on page 12-599.
*12.22 DCW and DCWU* on page 12-600.
*12.64 SPACE or FILL* on page 12-648.
*12.20 DCI* on page 12-598.

## 12.17 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.
```

## 12.18 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

*12.19 DCFS and DCFSU* on page 12-597.
*7.16 Syntax of floating-point literals* on page 7-147.

## 12.19 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

*12.18 DCFD and DCFDU* on page 12-596.
*7.16 Syntax of floating-point literals* on page 7-147.

## 12.20 DCI

The DCI directive allocates two or four-byte aligned memory and defines the initial runtime contents of the memory.

### Syntax

{*label*} DCI{.W} *expr*{,*expr*}

where:

*expr*
is a numeric expression.

*.W*
if present, indicates that four bytes must be inserted in Thumb 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 ARM code, DCI allocates one or more words of memory, aligned on four-byte boundaries. It inserts up to three bytes of padding before the first defined word, if necessary, to achieve four-byte alignment.

In Thumb code, DCI allocates one or more halfwords of memory, aligned on two-byte boundaries. It 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 Thumb 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 Thumb example

```
DCI.W   0xf3af8000   ; inserts 32-bit NOP, 2-byte aligned.
```

### Related concepts

*7.14 Numeric expressions* on page 7-145.

### Related references

*12.16 DCD and DCDU* on page 12-594.
*12.22 DCW and DCWU* on page 12-600.

## 12.21 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*…}

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*.

### 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
        DCQU    number          ; DCQ and DCQU only accept literals not
                                ; expressions.
```

### Related concepts

*7.14 Numeric expressions* on page 7-145.

### Related references

*12.15 DCB* on page 12-593.
*12.16 DCD and DCDU* on page 12-594.
*12.22 DCW and DCWU* on page 12-600.
*12.64 SPACE or FILL* on page 12-648.

## 12.22 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

*7.14 Numeric expressions* on page 7-145.

### Related references

*12.15 DCB* on page 12-593.
*12.16 DCD and DCDU* on page 12-594.
*12.21 DCQ and DCQU* on page 12-599.
*12.64 SPACE or FILL* on page 12-648.

## 12.23 DN and SN

The DN and SN directives define names for VFP registers.

### Syntax

*name directive expr{.type}*

where:

*name*
: is the name to be assigned to the extension register. *name* cannot be the same as any of the predefined names.

*directive*
: is DN or SN.

*expr*
: Can be:
- An expression that evaluates to a number in the range:
  — 0-15 if you are using DN in VFPv2.
  — 0-31 otherwise.
- A predefined register name, or a register name that has already been defined in a previous directive.

*type*
: is any VFP datatype.

*type* is *Extended notation*.

### Usage

Use DN or SN to allocate convenient names to extension registers, to help you to remember what you use each one for.

The DN directive defines a name for a specified 64-bit extension register.

The SN directive defines a name for a specified single-precision VFP 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 VFP double-precision register 6
mass    SN  16  ; defines mass as a symbol for VFP single-precision register 16
```

### Related concepts

*8.9 VFP data types* on page 8-172.
*8.22 Overview of VFP directives and vector notation* on page 8-186.
*8.10 Extended notation extension for VFP* on page 8-173.

### Related references

*2.10 Predeclared core register names* on page 2-40.
*2.11 Predeclared extension register names* on page 2-41.
*2.12 Predeclared coprocessor names* on page 2-42.

## 12.24 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

*12.44 GET or INCLUDE* on page 12-623.

## 12.25 ENDFUNC or ENDP

The `ENDFUNC` directive marks the end of an AAPCS-conforming function. `ENDP` is a synonym for `ENDFUNC`.

### Related references

*12.42 FUNCTION or PROC* on page 12-621.

## 12.26 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.*

## 12.27 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

*12.49 KEEP* on page 12-630.
*12.28 EXPORT or GLOBAL* on page 12-606.

## 12.28 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 ARM symbol.

  **THUMB**
  : *symbol* is treated as a Thumb 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

*12.46 IMPORT and EXTERN* on page 12-626.

### Related information

*ELF for the ARM Architecture.*

## 12.29 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

*12.28 EXPORT or GLOBAL* on page 12-606.

## 12.30 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 1-25.

### Related references

*12.53 MAP* on page 12-636.

*1.4 Directives that can be omitted in pass 2 of the assembler* on page 1-27.

## 12.31 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

*12.32 FRAME POP* on page 12-611.
*12.33 FRAME PUSH* on page 12-612.

## 12.32 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 ARM 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

*12.31 FRAME ADDRESS* on page 12-610.
*12.35 FRAME RESTORE* on page 12-614.

## 12.33 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 ARM 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

*12.31 FRAME ADDRESS* on page 12-610.
*12.37 FRAME SAVE* on page 12-616.

## 12.34 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.

## 12.35 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

*12.32 FRAME POP* on page 12-611.

## 12.36 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.

## 12.37 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

*12.33 FRAME PUSH* on page 12-612.

## 12.38 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

*12.39 FRAME STATE RESTORE* on page 12-618.
*12.42 FUNCTION or PROC* on page 12-621.

## 12.39 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

*12.38 FRAME STATE REMEMBER* on page 12-617.
*12.42 FUNCTION or PROC* on page 12-621.

## 12.40 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

*9.29 --exceptions, --no_exceptions* on page 9-225.
*9.30 --exceptions_unwind, --no_exceptions_unwind* on page 9-226.

## 12.41 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

*9.29 --exceptions, --no_exceptions* on page 9-225.
*9.30 --exceptions_unwind, --no_exceptions_unwind* on page 9-226.

## 12.42 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 ARM 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 ARM 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 Thumb). 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

*12.39 FRAME STATE RESTORE* on page 12-618.
*12.31 FRAME ADDRESS* on page 12-610.
*12.5 ALIGN* on page 12-580.

## 12.43 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 --predefine "objectsize SETA 0xFF" sourcefile -o objectfile
```

### Related references

*12.50 LCLA, LCLL, and LCLS* on page 12-631.
*12.63 SETA, SETL, and SETS* on page 12-647.
*9.57 --predefine "directive"* on page 9-254.

## 12.44 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, `EQU`s, 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

*12.47 INCBIN* on page 12-628.
*12.2 About assembly control directives* on page 12-577.

## 12.45 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 --predefine "NEWVERSION SETL {TRUE}" test.s
```

Invoking armasm as follows leaves NEWVERSION undefined, so the second set of instructions and directives are assembled:

```
armasm 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 --predefine "NEWVERSION SETL {TRUE}" test.s
```

Invoking armasm as follows causes the second set of instructions and directives to be assembled:

```
armasm --predefine "NEWVERSION SETL {FALSE}" test.s
```

## Related references

*7.25 Relational operators* on page 7-156.
*12.2 About assembly control directives* on page 12-577.

## 12.46 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

*12.28 EXPORT or GLOBAL* on page 12-606.

### Related information

*ELF for the ARM Architecture.*

## 12.47 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.
```

## 12.48 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

*7.12 String expressions* on page 7-143.
*7.14 Numeric expressions* on page 7-145.

### Related references

*12.8 ASSERT* on page 12-586.

## 12.49 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

*7.10 Numeric local labels* on page 7-141.

### Related references

*12.53 MAP* on page 12-636.

## 12.50 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

*12.43 GBLA, GBLL, and GBLS* on page 12-622.
*12.63 SETA, SETL, and SETS* on page 12-647.
*12.52 MACRO and MEND* on page 12-633.

## 12.51 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

*10.46 LDR pseudo-instruction* on page 10-356.
*11.14 VLDR pseudo-instruction* on page 11-553.

## 12.52 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

*4.21 Use of macros* on page 4-85.

*7.4 Assembly time substitution of variables* on page 7-135.

### Related references

*12.54 MEXIT* on page 12-637.

*12.43 GBLA, GBLL, and GBLS* on page 12-622.

*12.50 LCLA, LCLL, and LCLS* on page 12-631.

## 12.53 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 1-25.

### Related references

*12.30 FIELD* on page 12-609.
*1.4 Directives that can be omitted in pass 2 of the assembler* on page 1-27.

## 12.54 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

*12.52 MACRO and MEND* on page 12-633.

## 12.55 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.

## 12.56 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 12-2 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

*9.43 --list=file* on page 9-240.

## 12.57 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 ARM or Thumb 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.*

## 12.58 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.

## 12.59 REQUIRE8 and PRESERVE8

The REQUIRE8 and PRESERVE8 directives specify that the current file requires or preserves eight-byte alignment of the stack.

### 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

*9.22 --diag_warning=tag[,tag,...]* on page 9-218.

### Related information

*Eight-byte Stack Alignment.*

## 12.60 RLIST

The RLIST (register list) directive gives a name to a set of general-purpose registers.

### 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

*2.10 Predeclared core register names* on page 2-40.
*2.11 Predeclared extension register names* on page 2-41.
*2.12 Predeclared coprocessor names* on page 2-42.

## 12.61 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

*2.10 Predeclared core register names* on page 2-40.
*2.11 Predeclared extension register names* on page 2-41.
*2.12 Predeclared coprocessor names* on page 2-42.

## 12.62 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

*7.10 Numeric local labels* on page 7-141.

### Related references

*12.6 AREA* on page 12-582.

## 12.63 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.

### Examples

```
                GBLA    VersionNumber
VersionNumber   SETA    21
                GBLL    Debug
Debug           SETL    {TRUE}
                GBLS    VersionString
VersionString   SETS    "Version 1.0"
```

### Related concepts

*7.12 String expressions* on page 7-143.
*7.14 Numeric expressions* on page 7-145.
*7.17 Logical expressions* on page 7-148.

### Related references

*12.43 GBLA, GBLL, and GBLS* on page 12-622.
*12.50 LCLA, LCLL, and LCLS* on page 12-631.
*9.57 --predefine "directive"* on page 9-254.

## 12.64 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

*7.14 Numeric expressions* on page 7-145.

### Related references

*12.5 ALIGN* on page 12-580.
*12.15 DCB* on page 12-593.
*12.16 DCD and DCDU* on page 12-594.
*12.21 DCQ and DCQU* on page 12-599.
*12.22 DCW and DCWU* on page 12-600.

## 12.65 THUMB

The THUMB directive instructs the assembler to interpret subsequent instructions as Thumb instructions, using the UAL syntax.

### Syntax

```
THUMB
```

### Usage

In files that contain code using different instruction sets, THUMB must precede Thumb 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 the assembler to assemble Thumb instructions, and inserts padding if necessary.

### ARM and THUMB directives

This example shows how you can use ARM and THUMB directives to switch state and assemble both ARM and Thumb instructions in a single area.

```
     AREA ToThumb, CODE, READONLY  ; Name this block of code
     ENTRY                         ; Mark first instruction to execute
     ARM                           ; Subsequent instructions are ARM
start
     ADR     r0, into_thumb + 1    ; Processor starts in ARM state
     BX      r0                    ; Inline switch to Thumb state
     THUMB                         ; Subsequent instructions are Thumb
into_thumb
     MOVS    r0, #10               ; New-style Thumb instructions
```

### Related references

*12.7 ARM or CODE32* on page 12-585.
*12.11 CODE16* on page 12-589.
*12.66 THUMBX* on page 12-650.

## 12.66 THUMBX

The `THUMBX` directive instructs the assembler to interpret subsequent instructions as ThumbEE instructions, using the UAL syntax.

### Syntax

```
THUMBX
```

### Usage

In files that contain code using different instruction sets, `THUMBX` must precede ThumbEE 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 the assembler to assemble ThumbEE instructions, and inserts padding if necessary.

**Note**

- ARM deprecates the use of ThumbEE instructions.
- For descriptions of ThumbEE instructions, see the ARM Architecture Reference Manual.

### Related references

*12.7 ARM or CODE32* on page 12-585.
*12.11 CODE16* on page 12-589.
*12.65 THUMB* on page 12-649.

### Related information

*ARM Architecture Reference Manual.*

## 12.67 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.
```

## 12.68 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

*7.17 Logical expressions* on page 7-148.

### Related references

*12.2 About assembly control directives* on page 12-577.

# Chapter 13
# **Via File Syntax**

Describes the syntax of via files accepted by `armasm`.

It contains the following sections:

## 13.1 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

*13.2 Via file syntax rules* on page 13-655.
*9.69 --via=filename* on page 9-266.

## 13.2 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

*13.1 Overview of via files* on page 13-654.

### Related references

*9.69 --via=filename* on page 9-266.