

RealView[®] Compilation Tools

Version 2.2

Assembler Guide



RealView Compilation Tools

Assembler Guide

Copyright © 2002-2005 ARM Limited. All rights reserved.

Release Information

The following changes have been made to this book.

Change History

Date	Issue	Change
August 2002	A	Release 1.2
January 2003	B	Release 2.0
September 2003	C	Release 2.0.1 for RVDS v2.0
January 2004	D	Release 2.1 for RVDS v2.1
December 2004	E	Release 2.2 for RVDS v2.2
May 2005	F	Release 2.2 for RVDS v2.2 SP1

Proprietary Notice

Words and logos marked with ® or ™ are registered trademarks or trademarks owned by ARM Limited. Other brands and names mentioned herein may be the trademarks of their respective owners.

Neither the whole nor any part of the information contained in, or the product described in, this document may be adapted or reproduced in any material form except with the prior written permission of the copyright holder.

The product described in this document is subject to continuous developments and improvements. All particulars of the product and its use contained in this document are given by ARM in good faith. However, all warranties implied or expressed, including but not limited to implied warranties of merchantability, or fitness for purpose, are excluded.

This document is intended only to assist the reader in the use of the product. ARM Limited shall not be liable for any loss or damage arising from the use of any information in this document, or any error or omission in such information, or any incorrect use of the product.

Confidentiality Status

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

Product Status

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

Web Address

<http://www.arm.com>

Contents

RealView Compilation Tools Assembler Guide

Preface

About this book	viii
Feedback	xi

Chapter 1

Introduction

1.1 About the RealView Compilation Tools assemblers	1-2
---	-----

Chapter 2

Writing ARM Assembly Language

2.1 Introduction	2-2
2.2 Overview of the ARM architecture	2-3
2.3 Structure of assembly language modules	2-11
2.4 Conditional execution	2-17
2.5 Loading constants into registers	2-23
2.6 Loading addresses into registers	2-31
2.7 Load and store multiple register instructions	2-37
2.8 Using macros	2-43
2.9 Adding symbol versions	2-46
2.10 Using frame directives	2-47
2.11 Assembly language changes	2-48

Chapter 3

Assembler Reference

3.1 Command syntax	3-2
--------------------------	-----

3.2	Format of source lines	3-14
3.3	Predefined register and coprocessor names	3-15
3.4	Built-in variables and constants	3-16
3.5	Symbols	3-18
3.6	Expressions, literals, and operators	3-24
3.7	Diagnostic messages	3-36
3.8	Using the C preprocessor	3-37

Chapter 4

ARM and Thumb Instructions

4.1	Instruction summary	4-2
4.2	Memory access instructions	4-7
4.3	General data processing instructions	4-41
4.4	Multiply instructions	4-73
4.5	Saturating instructions	4-95
4.6	Parallel instructions	4-100
4.7	Packing and unpacking instructions	4-108
4.8	Branch instructions	4-115
4.9	Coprocessor instructions	4-121
4.10	Miscellaneous instructions	4-131
4.11	Pseudo-instructions	4-144

Chapter 5

Vector Floating Point Programming

5.1	The Vector Floating Point coprocessor	5-4
5.2	Floating-point registers	5-5
5.3	Vector and scalar operations	5-7
5.4	VFP and condition codes	5-8
5.5	VFP system registers	5-10
5.6	Flush-to-zero mode	5-13
5.7	VFP instructions	5-15
5.8	VFP pseudo-instruction	5-36
5.9	VFP directives and vector notation	5-37

Chapter 6

Directives Reference

6.1	Alphabetical list of directives	6-2
6.2	Symbol definition directives	6-4
6.3	Data definition directives	6-13
6.4	Assembly control directives	6-28
6.5	Frame directives	6-37
6.6	Reporting directives	6-52
6.7	Instruction set and syntax selection directives	6-57
6.8	Miscellaneous directives	6-61

Glossary

Preface

This preface introduces the *RealView Compilation Tools Assembler Guide*. It contains the following sections:

- *About this book* on page viii
- *Feedback* on page xi.

About this book

This book provides tutorial and reference information for the *RealView® Compilation Tools* (RVCT) assemblers. This includes *armasm*, the free-standing assembler, and inline assemblers in the C and C++ compilers. It describes the command-line options to the assembler, the assembly language mnemonics, the pseudo-instructions, the macros, and directives available to assembly language programmers, and the ARM®, Thumb®-2, Thumb, and *Vector Floating Point* (VFP) instruction sets.

Intended audience

This book is written for all developers who are producing applications using RVCT. It assumes that you are an experienced software developer and that you are familiar with the ARM development tools as described in *RealView Compilation Tools v2.2 Essentials Guide*.

Using this book

This book is organized into the following chapters:

Chapter 1 *Introduction*

Read this chapter for an introduction to the RVCT assemblers and assembly language.

Chapter 2 *Writing ARM Assembly Language*

Read this chapter for tutorial information to help you use the ARM assemblers and assembly language.

Chapter 3 *Assembler Reference*

Read this chapter for reference material about the syntax and structure of the language provided by the ARM assemblers.

Chapter 4 *ARM and Thumb Instructions*

Read this chapter for reference material on the ARM and Thumb instruction sets, covering both Thumb-2 and earlier Thumb.

Chapter 5 *Vector Floating Point Programming*

Read this chapter for reference material on the VFP instruction set, and other VFP-specific assembly language information.

Chapter 6 *Directives Reference*

Read this chapter for reference material on the assembler directives available in the ARM assembler, *armasm*.

Glossary An alphabetically arranged glossary defines the special terms used in this book.

This book assumes that you have installed your ARM software in the default location, for example, on Windows this might be *volume:\Program Files\ARM*. This is assumed to be the location of *install_directory* when referring to path names, for example *install_directory\Documentation\...* You might have to change this if you have installed your ARM software in a different location.

Typographical conventions

The following typographical conventions are used in this book:

<code>monospace</code>	Denotes text that can be entered at the keyboard, such as commands, file and program names, and source code.
<u><code>monospace</code></u>	Denotes a permitted abbreviation for a command or option. The underlined text can be entered instead of the full command or option name.
<i>monospace italic</i>	Denotes arguments to commands and functions where the argument is to be replaced by a specific value.
<code>monospace bold</code>	Denotes language keywords when used outside example code.
<i>italic</i>	Highlights important notes, introduces special terminology, denotes internal cross-references, and citations.
bold	Highlights interface elements, such as menu names. Also used for emphasis in descriptive lists, where appropriate, and for ARM processor signal names.

Further reading

This section lists publications from both ARM Limited and third parties that provide additional information on developing code for the ARM family of processors.

ARM periodically provides updates and corrections to its documentation. See <http://www.arm.com> for current errata sheets and addenda, and the ARM Frequently Asked Questions.

ARM publications

This book contains reference information that is specific to development tools supplied with RVCT. Other publications included in the suite are:

- *RealView Compilation Tools v2.2 Essentials Guide* (ARM DUI 0202)
- *RealView Compilation Tools v2.2 Developer Guide* (ARM DUI 0203)
- *RealView Compilation Tools v2.2 Compiler and Libraries Guide* (ARM DUI 0205)
- *RealView Compilation Tools v2.2 Linker and Utilities Guide* (ARM DUI 0206)

For full information about the base standard, software interfaces, and standards supported by ARM, see *install_directory\Documentation\Specifications\....*

In addition, see the following documentation for specific information relating to ARM products:

- *ARM Architecture Reference Manual* (ARM DDI 0100)
- *ARM Architecture Reference Manual, Thumb-2 supplement* (ARM DDI 0308)
- *ARM Architecture Reference Manual, Security Extensions supplement* (ARM DDI 0309)
- *ARM Reference Peripheral Specification* (ARM DDI 0062)
- the ARM datasheet or technical reference manual for your hardware device.

Other publications

For an introduction to ARM architecture, see Steve Furber, *ARM system-on-chip architecture* (2nd edition, 2000). Addison Wesley, ISBN 0-201-67519-6.

Feedback

ARM Limited welcomes feedback on both RealView Compilation Tools and the documentation.

Feedback on RealView Compilation Tools

If you have any problems with RVCT, contact your supplier. To help them provide a rapid and useful response, give:

- your name and company
- the serial number of the product
- details of the release you are using
- details of the platform you are running on, such as the hardware platform, operating system type and version
- a small standalone sample of code that reproduces the problem
- a clear explanation of what you expected to happen, and what actually happened
- the commands you used, including any command-line options
- sample output illustrating the problem
- the version string of the tools, including the version number and build numbers.

Feedback on this book

If you notice any errors or omissions in this book, send email to errata@arm.com giving:

- the document title
- the document number
- the page number(s) to which your comments apply
- a concise explanation of the problem.

General suggestions for additions and improvements are also welcome.

Chapter 1

Introduction

This chapter introduces the assemblers provided with *RealView® Compilation Tools* (RVCT). It contains the following section:

- *About the RealView Compilation Tools assemblers* on page 1-2.

1.1 About the RealView Compilation Tools assemblers

RVCT has:

- a freestanding assembler, `armasm`
- an optimizing inline assembler built into the C and C++ compilers.

The language that these assemblers take as input is basically the same. However, there are limitations on what features of the language you can use in the inline assemblers. See the *Mixing C, C++, and Assembly Language* chapter in *RealView Compilation Tools v2.2 Developer Guide* for more information on the inline assemblers.

If you are upgrading to RVCT from a previous release, ensure that you read *RealView Compilation Tools v2.2 Essentials Guide* for details about new features and enhancements in this release.

1.1.1 ARM assembly language

The RVCT v2.2 assembler implements a substantially enhanced version of the assembly language, while continuing to support the old version to enable the assembly of old code. The new version can be assembled to either ARM® or Thumb® instructions.

The new version is necessary to support Thumb-2, an upgraded version of the Thumb instruction set available on the latest processors from ARM. Thumb-2 has most of the functionality of the ARM instruction set. The old Thumb assembly language is insufficiently versatile to describe all the new instructions.

You can write source code that can be assembled for either ARM or Thumb-2.

You can also use the new ARM assembly language for writing Thumb code for earlier, pre-Thumb-2 processors. In this case, you must only use instructions that are available on the processor you are writing for. This manual provides the information necessary for you to do this. The assembler faults the use of unavailable instructions.

1.1.2 Using the examples

This book references examples provided with RealView Developer Suite in the main examples directory `install_directory\RVDs\Examples`. See *RealView Developer Suite Getting Started Guide* for a summary of the examples provided.

Chapter 2

Writing ARM Assembly Language

This chapter provides an introduction to the general principles of writing ARM® *Assembly Language*. It contains the following sections:

- *Introduction* on page 2-2
- *Overview of the ARM architecture* on page 2-3
- *Structure of assembly language modules* on page 2-11
- *Conditional execution* on page 2-17
- *Loading constants into registers* on page 2-23
- *Loading addresses into registers* on page 2-31
- *Load and store multiple register instructions* on page 2-37
- *Using macros* on page 2-43
- *Adding symbol versions* on page 2-46
- *Using frame directives* on page 2-47
- *Assembly language changes* on page 2-48.

2.1 Introduction

This chapter gives a basic, practical understanding of how to write ARM assembly language modules. It also gives information on the facilities provided by the ARM assembler (armasm).

This chapter does not provide a detailed description of the ARM, Thumb®-2, Thumb, or VFP instruction sets. For this information see:

- Chapter 4 *ARM and Thumb Instructions*
- Chapter 5 *Vector Floating Point Programming*.

For more information, see *ARM Architecture Reference Manual* and *ARM Architecture Reference Manual, Thumb-2 supplement*.

For the convenience of programmers who are already familiar with ARM and Thumb assembly languages, this chapter includes a section outlining the differences between the latest version of the ARM assembly language and earlier versions of the ARM and Thumb assembly languages, see *Assembly language changes* on page 2-48.

2.1.1 Code examples

There are a number of code examples in this chapter. Many of them are supplied in the *install_directory\RVD\Examples\...\asm* directory.

Follow these steps to build and link an assembly language file:

1. Type `armasm --debug filename.s` at the command prompt to assemble the file and generate debug tables.
2. Type `armlink filename.o -o filename` to link the object file and generate an ELF executable image.

To execute and debug the image, load it into a compatible debugger, for example RealView Debugger or *ARM eXtended Debugger (AXD)*, with an appropriate debug target.

To see how the assembler converts the source code, enter:

```
fromelf -c filename.o
```

See *RealView Compilation Tools v2.2 Linker and Utilities Guide* for details on `armlink` and `fromelf`.

2.2 Overview of the ARM architecture

This section gives a brief overview of the ARM architecture.

ARM processors are typical of RISC processors in that they implement a load/store architecture. Only load and store instructions can access memory. Data processing instructions operate on register contents only.

This section describes:

- *Architecture versions*
- *ARM, Thumb, and Thumb-2 instruction sets*
- *ARM and Thumb state* on page 2-4
- *Processor mode* on page 2-4
- *Registers* on page 2-5
- *Instruction set overview* on page 2-7
- *Instruction capabilities* on page 2-9.

2.2.1 Architecture versions

The information and examples in this book assume that you are using a processor that implements ARMv4 or above. RVCT also includes support for ARMv6 extensions, for example Security Extensions, and the ARM MPCore™.

All these processors have a 32-bit addressing range.

See *ARM Architecture Reference Manual* for details of the various architecture versions.

2.2.2 ARM, Thumb, and Thumb-2 instruction sets

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

ARMv4T and above define a 16-bit instruction set called the Thumb instruction set. 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 inferior performance.

ARMv6T2 defines Thumb-2, a major enhancement of the Thumb instruction set. Thumb-2 provides almost exactly the same functionality as the ARM instruction set. It has both 16-bit and 32-bit instructions, and achieves ARM-like performance with Thumb-like code density.

ARMv6T2 also defines several new instructions in the ARM instruction set.

In ARMv6 and above, all ARM and Thumb instructions are little-endian. In ARMv6T2 and above, all Thumb-2 instruction fetches are little-endian.

For more information, see *Instruction set overview* on page 2-7.

2.2.3 ARM and Thumb state

A processor that is executing Thumb instructions is operating in *Thumb state*. A processor that is executing ARM instructions is operating in *ARM state*.

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.

Each instruction set includes instructions to change processor state.

You must also switch the assembler mode to produce the correct opcodes using ARM (CODE32), THUMB, or CODE16 directives. See *Instruction set and syntax selection directives* on page 6-57 for details.

ARM processors always start executing code in ARM state.

2.2.4 Processor mode

ARM processors support different processor modes, depending on the architecture version (see Table 2-1).

Table 2-1 ARM processor modes

Processor mode	Architectures	Mode number
User	All	0b10000
FIQ - Fast Interrupt Request	All	0b10001
IRQ - Interrupt Request	All	0b10010
Supervisor	All	0b10011
Abort	All	0b10111
Undefined	All	0b11011
System	ARMv4 and above	0b11111
Monitor	Security Extensions only	0b10110

All modes except User mode are referred to as *privileged* modes. They have full access to system resources and can change mode freely.

Applications that require task protection usually execute in User mode. Some embedded applications might run entirely in Supervisor or System modes.

Modes other than User mode are entered to service exceptions, or to access privileged resources (see the *Handling Processor Exceptions* chapter in *RealView Compilation Tools v2.2 Developer Guide*).

2.2.5 Registers

ARM processors have 37 registers. The registers are arranged in partially overlapping banks. There is a different register bank for each processor mode. The banked registers give rapid context switching for dealing with processor exceptions and privileged operations. See *ARM Architecture Reference Manual* for a detailed description of how registers are banked.

The following registers are available:

- *Thirty general-purpose, 32-bit registers*
- *The Program Counter (PC)* on page 2-6
- *The Current Program Status Register (CPSR)* on page 2-6
- *Saved Program Status Registers (SPSRs)* on page 2-6.

Thirty general-purpose, 32-bit registers

Fifteen general-purpose registers are visible at any one time, depending on the current processor mode, as r0, r1, ... , r13, r14.

By convention, r13 is used as a *stack pointer* (sp) in ARM assembly language. The C and C++ compilers always use r13 as the stack pointer.

In User mode, r14 is used as a *link register* (lr) 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, r14 holds the return address for the exception, or a subroutine return address if subroutine calls are executed within an exception. r14 can be used as a general-purpose register if the return address is stored on the stack.

The Program Counter (PC)

The Program Counter is accessed as r15 (or pc). It is incremented by one word (four bytes) for each instruction in ARM state, or by two bytes in Thumb state. Branch instructions load the destination address into pc. You can also load the PC directly using data operation instructions. For example, to return from a subroutine, you can copy the link register into the PC using:

```
MOV    pc,lr
```

During execution, r15 (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.

The Current Program Status Register (CPSR)

The CPSR holds:

- copies of the *Arithmetic Logic Unit* (ALU) status flags
- the current processor mode
- interrupt disable flags.

The ALU status flags in the CPSR are used to determine whether conditional instructions are executed or not. See *Conditional execution* on page 2-17 for more information.

On Thumb-capable or Jazelle®-capable processors, the CPSR also holds the current processor state (ARM, Thumb, or Jazelle).

On ARMv5TE, and ARMv6 and above, the CPSR also holds the Q flag (see *The ALU status flags* on page 2-18).

On ARMv6 and above, the CPSR also holds the GE flags (see *Parallel add and subtract* on page 4-101) and the Endianness bit (see *SETEND* on page 4-140).

On ARMv6T2 and above, Thumb-2 introduces new state bits to the CPSR. These are used by the IT instruction to control conditional execution of the IT block (see *IT* on page 4-70).

Saved Program Status Registers (SPSRs)

The SPSRs are used to store the CPSR when an exception is taken. One SPSR is accessible in each of the exception-handling modes. User mode and System mode do not have an SPSR because they are not exception handling modes. See the chapter describing *Handling Processor Exceptions in RealView Compilation Tools v2.2 Developer Guide* for more information.

2.2.6 Instruction set overview

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 and Thumb-2 instructions are either 16 or 32 bits long. Instructions are stored half-word aligned, so the least significant bit of instruction addresses is always zero in Thumb state.

Some instructions use the least significant bit to determine whether the code being branched to is Thumb code or ARM code.

Before the introduction of Thumb-2, 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. The Thumb-2 instruction set functionality is almost identical to that of the ARM instruction set.

See Chapter 4 *ARM and Thumb Instructions* for detailed information on the syntax of ARM and Thumb instructions.

ARM and Thumb instructions can be classified into a number of functional groups:

- *Branch instructions*
- *Data processing instructions* on page 2-8
- *Single register load and store instructions* on page 2-8
- *Multiple register load and store instructions* on page 2-8
- *Status register access instructions* on page 2-8
- *Coprocessor instructions* on page 2-8.

Branch instructions

These instructions are used to:

- branch backwards to form loops
- branch forward in conditional structures
- branch to subroutines
- change the processor between ARM state and Thumb state.

Data processing instructions

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 a constant supplied within the instruction (an *immediate value*).

Long multiply instructions give a 64-bit result in two registers.

Single register load and store instructions

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.

Multiple register load and store instructions

These instructions load or store any subset of the general-purpose registers from or to memory. See *Load and store multiple register instructions* on page 2-37 for a detailed description of these instructions.

Status register access instructions

These instructions move the contents of the CPSR or an SPSR to or from a general-purpose register.

Coprocessor instructions

These instructions support a general way to extend the ARM architecture.

2.2.7 Instruction capabilities

This section contains the following subsections:

- *Conditional execution*
- *Register access*
- *Access to the inline barrel shifter* on page 2-10.

Conditional execution

Almost all ARM instructions can be executed conditionally on the value of the ALU status flags in the CPSR. You do not have to use branches to skip conditional instructions, although it can be better to do so when a series of instructions depend on the same condition.

Thumb-2 provides an alternative mechanism for conditional execution, using the IT (If-Then) instruction and the same ALU flags. IT is a 16-bit instruction that provides conditional execution of up to four following instructions. There are also several other instructions providing additional mechanisms for conditional execution.

In ARM and Thumb-2 code, you can specify whether a data processing instruction updates the ALU flags or not. You can use the flags set by one instruction to control execution of other instructions even if there are many instructions in between.

In Thumb state on processors without Thumb-2, the only mechanism for conditional execution is a conditional branch. Most data processing instructions update the ALU flags. You cannot generally specify whether instructions update the state of the ALU flags or not.

See *Conditional execution* on page 2-17 for a detailed description.

Register access

In ARM state, all instructions can access r0 to r14, and most can also access r15 (pc). The MRS and MSR instructions can move the contents of the CPSR and SPSRs to a general-purpose register, where they can be manipulated by normal data processing operations. See *MRS* on page 4-134 and *MSR* on page 4-135 for more information.

Thumb state on Thumb-2 processors provides the same facilities, except that some of the less useful accesses to r15 are not permitted.

In Thumb state on processors without Thumb-2, most instructions can only access r0 to r7. Only a small number of instructions can access r8 to r15. Registers r0-r7 are called Lo registers. Registers r8-r15 are called Hi registers.

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 all ARM and Thumb-2 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 a constant
- constructing constants.

See *Loading constants into registers* on page 2-23 for more information on using the barrel shifter to generate constants.

Thumb-2 instructions give almost the same access to the barrel shifter as ARM instructions.

The Thumb instruction set available on processors without Thumb-2 only allows access to the barrel shifter using separate instructions.

2.3 Structure of assembly language modules

Assembly language is the language that the ARM assembler (`armasm`) parses and assembles to produce object code. By default, the assembler expects source code to be written in ARM assembly language.

`armasm` accepts source code written in older versions of ARM assembly language. It does not have to be informed in this case.

`armasm` can also accept source code written in old Thumb assembly language. In this case, you must inform `armasm` using the `--16` command-line option, or the `CODE16` directive in the source code.

This section describes:

- *Layout of assembly language source files*
- *An example ARM assembly language module* on page 2-14
- *Calling subroutines* on page 2-16.

2.3.1 Layout of assembly language source files

The general form of source lines in assembly language is:

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

Note

Instructions, pseudo-instructions, and directives must be preceded by white space, such as a space or a tab, even if there is no label.

All three sections of the source line are optional. You can use blank lines to make your code more readable.

Case rules

Instruction mnemonics, directives, and symbolic register names can be written in uppercase or lowercase, but not mixed.

Line length

To make source files easier to read, a long line of source can be split onto 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/end-of-line sequence as white space.

Note

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

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

Labels

Labels are symbols that represent addresses. The address given by a label is calculated during assembly.

The assembler 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 *program-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.

Local labels

Local labels are a subclass of label. A local label begins with a number in the range 0-99. Unlike other labels, a local label can be defined many times. Local labels are useful when you are generating labels with a macro. When the assembler finds a reference to a local label, it links it to a nearby instance of the local label.

The scope of local labels is limited by the AREA directive. You can use the ROUT directive to limit the scope more tightly.

See *Local labels* on page 3-22 for details of:

- the syntax of local label declarations
- how the assembler associates references to local labels with their labels.

Comments

The first semicolon on a line marks the beginning of a comment, except where the semicolon appears inside a string constant. The end of the line is the end of the comment. A comment alone is a valid line. The assembler ignores all comments.

Constants

Constants can be:

- Numbers** Numeric constants are accepted in the following forms:
- decimal, for example, 123
 - hexadecimal, for example, 0x7B
 - *n_xxx* where:
 - n* is a base between 2 and 9
 - xxx* is a number in that base.
- Boolean** The Boolean constants TRUE and FALSE must be written as {TRUE} and {FALSE}.
- Characters** Character constants consist of opening and closing single quotes, enclosing either a single character or an escaped character, using the standard C escape characters.
- Strings** Strings consist of opening and closing double quotes, enclosing characters and spaces. If double quotes or dollar signs are used within a string as literal text characters, they must be represented by a pair of the appropriate character. For example, you must use \$\$ if you require a single \$ in the string. The standard C escape sequences can be used within string constants.

2.3.2 An example ARM assembly language module

Example 2-1 illustrates some of the core constituents of an assembly language module. The example is written in ARM assembly language. It is supplied as `armex.s` in the main examples directory `install_directory\RVD\Examples`. See *Code examples* on page 2-2 for instructions on how to assemble, link, and execute the example.

The constituent parts of this example are described in more detail in the following sections.

Example 2-1

```

        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
        SWI       0x123456     ; ARM semihosting SWI

        END
                                ; Mark end of file

```

ELF sections and the AREA directive

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. See the chapter describing basic linker functionality in *RealView Compilation Tools v2.2 Linker and Utilities Guide* for more information on how the linker places sections.

In a source file, the `AREA` directive marks the start of a section. This directive names the section and sets its attributes. The attributes are placed after the name, separated by commas. See *AREA* on page 6-64 for a detailed description of the syntax of the `AREA` directive.

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

Example 2-1 on page 2-14 defines a single section called `ARMex` that contains code and is marked as being `READONLY`.

The `ENTRY` directive

The `ENTRY` directive marks the first instruction to be executed. In applications containing C code, 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 in Example 2-1 on page 2-14 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. This is done using the ARM semihosting `SWI` (0x123456 by default), with the following parameters:

- `r0` equal to `angel_SWIreason_ReportException` (0x18)
- `r1` equal to `ADP_Stopped_ApplicationExit` (0x20026).

See the *Semihosting SWIs* chapter in *RealView Compilation Tools v2.2 Compiler and Libraries Guide* for additional information.

The `END` directive

This 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.

2.3.3 Calling subroutines

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 program-relative or register-relative expression. See *B*, *BL*, *BX*, *BLX*, and *BXJ* on page 4-116 for more information.

The BL instruction:

- places the return address in the link register
- sets the PC to the address of the subroutine.

After the subroutine code is executed you can use a MOV pc,lr instruction to return. By convention, registers r0 to r3 are used to pass parameters to subroutines, and to pass results back to the callers.

————— **Note** —————

Calls between separately assembled or compiled modules must comply with the restrictions and conventions defined by the procedure call standard. See the *Using the Procedure Call Standard* appendix in *RealView Compilation Tools v2.2 Developer Guide* for more information.

Example 2-2 shows a subroutine that adds the values of two parameters and returns a result in r0. It is supplied as subroun.s in the main examples directory *install_directory\RVDS\Examples*. See *Code examples* on page 2-2 for instructions on how to assemble, link, and execute the example.

Example 2-2

	AREA	subroun, 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
	SWI	0x123456	; ARM semihosting SWI
doadd	ADD	r0, r0, r1	; Subroutine code
	BX	lr	; Return from subroutine
	END		; Mark end of file

2.4 Conditional execution

In ARM state, and in Thumb state on processors with Thumb-2, most data processing instructions have an option to update ALU status flags in the *Current Program Status Register* (CPSR) according to the result of the operation.

In Thumb state on earlier architectures, most data processing instructions update the ALU status flags automatically. There is no option not to update the flags. Other instructions cannot update the flags.

Almost every ARM instruction can be executed conditionally on the state of the ALU status flags in the CPSR. See Table 2-2 on page 2-18 for a list of the suffixes to add to instructions to make them conditional.

Almost all Thumb-2 instructions can be made conditional by the use of a special IT (If-Then) instruction. In addition, there are several conditional branch instructions.

In Thumb state on pre-Thumb-2 processors, the only mechanism for conditional execution is a conditional branch.

Flags are preserved until updated. A conditional instruction that is not executed has no effect on the flags.

Some instructions update a subset of the flags. The other flags are unchanged by these instructions. Details are specified in the descriptions of the instructions.

In ARM state, and in Thumb state on processors with Thumb-2, you can execute an instruction conditionally, based upon the flags set in another instruction, either:

- immediately after the instruction which updated the flags
- after any number of intervening instructions that have not updated the flags.

This section describes:

- *The ALU status flags* on page 2-18
- *Conditional execution in ARM state* on page 2-18
- *Using conditional execution in ARM state* on page 2-19
- *Example of the use of conditional execution* on page 2-20
- *The Q flag* on page 2-22.

2.4.1 The ALU status flags

The CPSR contains the following ALU status flags:

N	Set when the result of the operation was Negative.
Z	Set when the result of the operation was Zero.
C	Set when the operation resulted in a Carry.
V	Set when the operation caused oVerflow.

A carry occurs if the result of an addition is greater than or equal to 2^{32} , if the result of a subtraction is positive, or as the result of an inline barrel shifter operation in a move or logical instruction.

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

2.4.2 Conditional execution in ARM state

The ARM instructions that can be conditional have an optional condition code, shown in syntax descriptions as {*cond*}. This condition is encoded in the instruction. An instruction with a condition code is only executed if the condition code flags in the CPSR meet the specified condition. Table 2-2 shows the condition codes that you can use.

Table 2-2 also shows the relationship between condition code suffixes and the N, Z, C and V flags.

Table 2-2 Condition code suffixes

Suffix	Flags	Meaning
EQ	Z set	Equal
NE	Z clear	Not equal
CS/HS	C set	Higher or same (unsigned $\geq$)
CC/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 $>$)

Table 2-2 Condition code suffixes (continued)

Suffix	Flags	Meaning
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.

Example 2-3 shows an example of conditional execution.

Example 2-3

```

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.

```

2.4.3 Using conditional execution in ARM state

You can use conditional execution of ARM instructions to reduce the number of branch instructions in your code. This improves code density.

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 ARM10™ and StrongARM®, have branch prediction hardware. In systems using these processors, the pipeline only has to be flushed and refilled when there is a misprediction.

2.4.4 Example of the use of conditional execution

This example uses two implementations of the *Greatest Common Divisor* (gcd) algorithm (Euclid). It demonstrates how you can use conditional execution to improve code density and execution speed. The detailed analysis of execution speed only applies to an ARM7™ processor. The code density calculations apply to all ARM processors.

In C the 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;
}
```

You can implement the gcd function with conditional execution of branches only, in the following way:

```
gcd    CMP     r0, r1
        BEQ     end
        BLT     less
        SUB     r0, r0, r1
        B       gcd
less   SUB     r1, r1, r0
        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.

By using the conditional execution feature of the ARM instruction set, you can implement the gcd function in only four instructions:

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

In addition to improving code size, this code executes faster in most cases. Table 2-3 and Table 2-4 show the number of cycles used by each implementation for the case where r0 equals 1 and r1 equals 2. In this case, replacing branches with conditional execution of all instructions saves three cycles.

The conditional version of the code executes in the same number of cycles for any case where r0 equals r1. In all other cases, the conditional version of the code executes in fewer cycles.

Table 2-3 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
1	1	BEQ end	3
			Total = 13

Table 2-4 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

Converting to pre-Thumb-2 Thumb code

Because B is the only 16-bit Thumb instruction that can be executed conditionally, the gcd algorithm must be written with conditional branches in Thumb code.

Like the ARM conditional branch implementation, the Thumb code requires seven instructions. When using Thumb instructions, the overall code size is 14 bytes, compared to 16 bytes for the smaller ARM implementation.

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

Branch prediction and caches

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. See *ARM Architecture Reference Manual* and the technical reference manuals for individual processors for full information.

2.4.5 The Q flag

ARMv5TE, and ARMv6 and above, have a Q flag to record when saturation has occurred in saturating arithmetic instructions (see *QADD*, *QSUB*, *QDADD*, and *QDSUB* on page 4-96), or when overflow has occurred in certain multiply instructions (see *SMULxy* and *SMLAxy* on page 4-78 and *SMULWy* and *SMLAWy* on page 4-80).

The Q flag is a *sticky* flag. Although these 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 (see *MSR* on page 4-135).

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 (see *MRS* on page 4-134).

2.5 Loading constants into registers

You cannot load an arbitrary 32-bit immediate constant into a register in a single ARM instruction without performing a data load from memory. This is because ARM instructions are only 32 bits long. Thumb instructions have a similar limitation.

You can also include many commonly-used constants directly as operands within data processing instructions, without a separate load operation.

You can load any 32-bit value into a register with a data load, but there are more direct and efficient ways to load many commonly-used constants.

In ARMv6T2 and above, you can also load any 32-bit value into a register with two instructions, a MOV followed by a MOVT. You can use a pseudo-instruction, MOV32, to construct the instruction sequence for you.

The following sections describe:

- how to use the MOV and MVN instructions to load a range of immediate values.
See *Direct loading with MOV and MVN* on page 2-24.
- how to use the MOV32 pseudo-instruction to load any 32-bit constant.
See *Loading with MOV32* on page 2-28.
- how to use the LDR pseudo-instruction to load any 32-bit constant.
See *Loading with LDR Rd, =const* on page 2-28.
- how to load floating-point constants.
See *Loading floating-point constants* on page 2-30.

2.5.1 Direct loading with MOV and MVN

In ARM and Thumb-2, you can use the 32-bit MOV and MVN instructions to load a wide range of constant values directly into a register.

The 16-bit Thumb MOV instruction can load any constant in the range 0-255. You cannot use the 16-bit MVN instruction to load a constant.

ARM state immediate constants shows the range of values that can be loaded in a single instruction in ARM. *Thumb-2 immediate constants* on page 2-26 shows the range of values that can be loaded in a single instruction in Thumb-2.

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 a constant that is not available, the assembler reports the error: Immediate *n* out of range for this operation.

ARM state immediate constants

In ARM state:

- MOV can load any 8-bit constant 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 above, MOV can load any 16-bit number, giving a range of 0x0-0xFFFF (0-65535).

Table 2-5 on page 2-25 shows the range of 8-bit values that this provides (for data processing operations).

Table 2-6 on page 2-25 shows the range of 16-bit values that this provides (for MOV instructions only).

Table 2-5 ARM state immediate constants (8-bit)

Binary	Decimal	Step	Hexadecimal	MVN value ^a	Notes
0000000000000000000000abcde fgh	0-255	1	0–0xFF	−1 to −256	-
0000000000000000000000abcde fgh00	0-1020	4	0–0x3FC	−4 to −1024	-
0000000000000000000000abcde fgh0000	0-4080	16	0–0xFF0	−16 to −4096	-
0000000000000000000000abcde fgh00000	0-16320	64	0–0xFC0	−64 to −16384	-
	...	...	...	...	-
abcde fgh000000000000000000000000	0-255 x 2 ²⁴	2 ²⁴	0–0xFFFF0000	1-256 x −2 ²⁴	-
cde fgh0000000000000000000000ab	(bit pattern)	-	-	(bit pattern)	See b in <i>Notes</i>
e fgh0000000000000000000000abcd	(bit pattern)	-	-	(bit pattern)	See b in <i>Notes</i>
gh0000000000000000000000abcdef	(bit pattern)	-	-	(bit pattern)	See b in <i>Notes</i>

Table 2-6 ARM state immediate constants (16-bit)

Binary	Decimal	Step	Hexadecimal	MVN value	Notes
0000000000000000abcdefghijklmnop	0-65535	1	0-0xFFFF	-	See c in <i>Notes</i>

Notes

These notes give extra information on Table 2-5 and Table 2-6.

- | | |
|----------|---|
| a | The MVN values are not available directly as operands in other, non data processing, instructions. |
| b | These values are available in ARM state only. All the other values in this table are also available in Thumb-2, except where noted. |
| c | These values are only available in ARMv6T2 and above. They are not available directly as operands in other instructions. |

Thumb-2 immediate constants

In Thumb state, in ARMv6T2 and above:

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

Table 2-7 on page 2-27 shows the range of values that this provides (for data processing operations).

Table 2-8 on page 2-27 shows the range of 16-bit values that this provides (for MOV instructions only).

Table 2-7 Thumb state immediate constants

Binary	Decimal	Step	Hexadecimal	MVN value ^a	Notes
000000000000000000000000abcde fgh	0-255	1	0-0xFF	-1 to -256	-
000000000000000000000000abcde fgh0	0-510	2	0-0x1FE	-2 to -512	-
000000000000000000000000abcde fgh00	0-1020	4	0-0x3FC	-4 to -1024	-
	...	...	...	...	-
0abcde fgh000000000000000000000000	0-255 x 2 ²³	2 ²³	0-0xF800000	1-256 x -2 ²³	-
abcde fgh000000000000000000000000	0-255 x 2 ²⁴	2 ²⁴	0-0xFFFF0000	1-256 x -2 ²⁴	-
abcde fghabcde fghabcde fghabcde fgh	(bit pattern)	-	0xXYXYXYXY	-	-
00000000abcde fgh00000000abcde fgh	(bit pattern)	-	0x00XY00XY	0xFFXYFFXY	-
abcde fgh00000000abcde fgh00000000	(bit pattern)	-	0xXY00XY00	0xXYFFXYFF	-
000000000000000000000000abcde fghi jkl	0-4095	1	0-0xFFF	-	See b in Notes

Table 2-8 Thumb state immediate constants (16-bit)

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

Notes

These notes give extra information on Table 2-7 and Table 2-8.

- | | |
|----------|--|
| a | The MVN values are not available directly as operands in other instructions. |
| b | These values are available directly as operands in data processing instructions, such as ADD and SUB, but not MVN. |
| c | These values are only available with MOV instructions. |

2.5.2 Loading with MOV32

In ARMv6T2, both ARM and Thumb-2 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 constant in a register. Alternatively, you can use the MOV32 pseudo-instruction. The assembler generates the MOV, MOVT instruction pair for you. See *MOV32 pseudo-instruction* on page 4-149 for a description of the syntax of the MOV32 pseudo-instruction.

2.5.3 Loading with LDR Rd, =const

The LDR Rd,=const pseudo-instruction can construct any 32-bit numeric constant in a single instruction. 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 a specific constant:

- If the constant can be constructed with a MOV or MVN instruction, the assembler generates the appropriate instruction.
- If the constant cannot be constructed with a 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 program-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. See *Placing literal pools* on page 2-29 for more information.

See *LDR pseudo-instruction* on page 4-151 for a description of the syntax of the LDR pseudo-instruction.

Placing literal pools

The assembler places a literal pool at the end of each section. These are defined by the AREA directive at the start of the following section, or by the END directive at the end of the assembly. 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 state, but can be in either direction
- forward and less than 1KB in Thumb state.

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

- Checks if the constant is available and addressable in any previous literal pools. If so, it addresses the existing constant.
- Attempts to place the constant 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 ± 4 KB (ARM, 32-bit Thumb-2) or in the range 0 to +1KB (Thumb, 16-bit Thumb-2). See *LTORG* on page 6-15 for a detailed description.

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 2-4 shows how this works. It is supplied as loadcon.s in the main examples directory *install_directory*\RVDS\Examples. See *Code examples* on page 2-2 for instructions on how to assemble, link, and execute the example.

The instructions listed as comments are the ARM instructions generated by the assembler.

Example 2-4

	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
        SWI    0x123456              ; ARM semihosting 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 0x55555555

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 empty

```

2.5.4 Loading floating-point constants

You can load any single-precision or double-precision floating-point constant in a single instruction, using the FLD pseudo-instructions

See *FLD pseudo-instruction* on page 5-36 for details.

2.6 Loading addresses into registers

It is often necessary to load an address into a register. You might have to load the address of a variable, a string constant, or the start location of a jump table.

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

This section describes the following methods for loading an address into a register:

- load the register directly, see *Direct loading with ADR and ADRL*
- load the address from a literal pool, see *Loading addresses with LDR Rd, =label* on page 2-34.

2.6.1 Direct loading with ADR and ADRL

The ADR and ADRL pseudo-instructions enable you to generate an address, within a certain range, without performing a data load. ADR and ADRL accept a program-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 ADR or ADRL must be within the same code section. The assembler faults references to labels that are out of range in the same section. The linker faults references to labels that are out of range in other code sections.

In Thumb state, a 16-bit ADR instruction can generate word-aligned addresses only.

ADRL is not available in Thumb code. Use it only in ARM code.

ADR

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

- a single ADD or SUB instruction that loads the address, if it is in range
- an error message if the address cannot be reached in a single instruction.

The available range depends on the instruction set:

ARM	± 255 bytes to a byte or halfword-aligned address. ± 1020 bytes to a word-aligned address.
32-bit Thumb-2	± 4095 bytes to a byte, halfword, or word-aligned address.
16-bit Thumb	0 to 1020 bytes. <i>label</i> must be word-aligned. You can use the ALIGN directive to ensure this.

See *ADR pseudo-instruction* on page 4-145 for details.

ADRL

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 in use:

ARM	± 64 KB to a byte or halfword-aligned address. ± 256 KB bytes to a word-aligned address.
32-bit Thumb-2	± 1 MB bytes to a byte, halfword, or word-aligned address.
16-bit Thumb	ADRL is not available.

See *Loading addresses with LDR Rd, =label* on page 2-34 for information on loading addresses that are outside the range of the ADRL pseudo-instruction.

Implementing a jump table with ADR

Example 2-5 on page 2-33 shows ARM code that implements a jump table. Here the ADR pseudo-instruction loads the address of the jump table. It is supplied as `jump.s` in the main examples directory `install_directory\RVD5\Examples`. See *Code examples* on page 2-2 for instructions on how to assemble, link, and execute the example.

Example 2-5 Implementing a jump table (ARM)

```

        AREA    Jump, CODE, READONLY    ; Name this block of code
CODE32
num     EQU     2                        ; Following code is ARM code
        ENTRY   start                    ; Number of entries in jump table
start   ; Mark first instruction to execute
        MOV     r0, #0                    ; First instruction to call
        MOV     r1, #3                    ; Set up the three parameters
        MOV     r2, #2
        BL      arithfunc                  ; Call the function
stop    MOV     r0, #0x18                  ; angel_SWIreason_ReportException
        LDR     r1, =0x20026               ; ADP_Stopped_ApplicationExit
        SWI     0x123456                  ; ARM semihosting SWI
arithfunc ; Label the function
        CMP     r0, #num                  ; Treat function code as unsigned
integer
        BXHS    lr                        ; If code is >= num then simply 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 Example 2-5, 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. It is used to give a value to a symbol. In Example 2-5 on page 2-33 it assigns the value 2 to <i>num</i> . When <i>num</i> 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 Example 2-5 on page 2-33 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.

2.6.2 Loading addresses with LDR Rd, =label

The LDR Rd, = pseudo-instruction can load any 32-bit constant into a register (see *Loading with LDR Rd, =const* on page 2-28). It also accepts program-relative expressions such as labels, and labels with offsets.

The assembler converts an LDR r0, =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 program-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 there is a literal pool within range (see *Placing literal pools* on page 2-29 for more information).

Unlike the ADR and ADRL pseudo-instructions, you can use LDR with labels that are outside the current section. If the label is 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.

Example 2-6 shows how this works. It is supplied as `ldrlabel.s` in the main examples directory `install_directory\RVDS\Examples`. See *Code examples* on page 2-2 for instructions on how to assemble, link, and execute the example.

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

Example 2-6

```

                                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
                                SWI      0x123456                    ; ARM semihosting 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]
                                MOV      pc, 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
                                                                ; as 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 out of range of
                                                                ; the LDR instructions above

```

An LDR Rd, =label example: string copying

Example 2-7 shows an ARM code routine that overwrites one string with another string. 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. See *DCB* on page 6-19 for more information.
- 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.

**Example 2-7 String copy**


---

|        |       |                                 |                                     |
|--------|-------|---------------------------------|-------------------------------------|
|        | 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    | strcpy                          | ; Call subroutine to do copy        |
| stop   | MOV   | r0, #0x18                       | ; angel_SWIreason_ReportException   |
|        | LDR   | r1, =0x20026                    | ; ADP_Stopped_ApplicationExit       |
|        | SWI   | 0x123456                        | ; ARM semihosting SWI               |
| strcpy | 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   | strcpy                          | ; 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   |                                 |                                     |

---

## 2.7 Load and store multiple register instructions

The ARM, Thumb-2, and Thumb instruction sets include instructions that load and store multiple registers to and from memory.

Multiple register transfer instructions provide an efficient way of moving the contents of several registers to and from memory. They 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 `--checkreglist` assembler command-line option to check that registers in register lists are specified in increasing order. See *Command syntax* on page 3-2 for more information.

---

This section describes:

- *Load and store multiple instructions available in ARM and Thumb* on page 2-38
- *Implementing stacks with LDM and STM* on page 2-39
- *Block copy with LDM and STM* on page 2-41.

### 2.7.1 Load and store multiple instructions available in ARM and Thumb

The following instructions are available in both ARM and Thumb instruction sets:

|      |                                                                       |
|------|-----------------------------------------------------------------------|
| 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 ARM instructions, any or all of r0-r15
  - in 32-bit Thumb-2 instructions, any or all of r0-r12, and optionally r14 or r15 with some restrictions
  - in 16-bit Thumb and Thumb-2 instructions, any or all of r0-r7.
- The address can be:
  - incremented after each transfer
  - incremented before each transfer (ARM instructions only)
  - decremented after each transfer (ARM instructions only)
  - decremented before each transfer (not in 16-bit Thumb).
- The base register can be either:
  - updated to point to the next block of data in memory
  - left as it was before the instruction (not in 16-bit Thumb).

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 (r13) 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 ARM instructions, any or all of r0-r15
  - in 32-bit Thumb-2 instructions, any or all of r0-r12, and optionally r14 or r15 with some restrictions
  - in 16-bit Thumb-2 and Thumb instructions, any or all of r0-r7, and optionally r14 (PUSH only) or r15 (POP only).

## 2.7.2 Implementing stacks with LDM and STM

The load and store multiple instructions can update the base register. For stack operations, the base register is usually the stack pointer, r13. 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. See Table 2-9 for a list of stack-oriented suffixes.

**Table 2-9 Suffixes for load and store multiple instructions**

| Stack type       | Push                            | Pop                             |
|------------------|---------------------------------|---------------------------------|
| 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 r13!, {r0-r5} ; Push onto a Full Descending Stack
LDMFD r13!, {r0-r5} ; Pop from a Full Descending Stack
```

### Note

The *Procedure Call Standard for the ARM Architecture* (AAPCS), and ARM and Thumb C and C++ compilers 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.

## Stacking registers for nested subroutines

Stack operations are very useful at subroutine entry and exit. 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 pc off the stack at exit, instead of popping lr and then moving that value into 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 ARMv5T and above, you can change state in this way.

See the *Interworking ARM and Thumb* chapter in *RealView Compilation Tools v2.2 Developer Guide* for more information on mixing ARM and Thumb.

---

### 2.7.3 Block copy with LDM and STM

Example 2-8 is an ARM code routine that copies a set of words from a source location to a destination by copying a single word at a time. It is supplied as `word.s` in the main examples directory `install_directory\RVDS\Examples`. See *Code examples* on page 2-2 for instructions on how to assemble, link, and execute the example.

#### Example 2-8 Block copy without LDM and STM

---

|          |       |                                         |                                     |
|----------|-------|-----------------------------------------|-------------------------------------|
|          | AREA  | Word, 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      |
| 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       |
|          | SWI   | 0x123456                                | ; ARM semihosting SWI               |
| src      | AREA  | BlockData, DATA, READWRITE              |                                     |
| dst      | DCD   | 1,2,3,4,5,6,7,8,1,2,3,4,5,6,7,8,1,2,3,4 |                                     |
|          | DCD   | 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0     |                                     |
|          | END   |                                         |                                     |

---

This module can be made 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 registers that the ARM has. The number of eight-word multiples in the block to be copied can be found (if `r2` = number of words to be copied) using:

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

This value can be used to control the number of iterations through a loop that copies eight words per iteration. When there are less than eight words left, the number of words left can be found (assuming that `r2` has not been corrupted) using:

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

Example 2-9 on page 2-42 lists the block copy module rewritten to use LDM and STM for copying.

**Example 2-9 Block copy using LDM and STM**


---

|           |       |                                         |                                                 |
|-----------|-------|-----------------------------------------|-------------------------------------------------|
| num       | AREA  | Block, CODE, READONLY                   | ; name this block of code                       |
|           | 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 (r13)                    |
| blockcopy | MOVS  | r3,r2, LSR #3                           | ; Number of eight word multiples                |
|           | BEQ   | copywords                               | ; Less 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 need these now - restore<br>; 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                   |
|           | SWI   | 0x123456                                | ; ARM semihosting SWI                           |
| src       | AREA  | BlockData, DATA, READWRITE              |                                                 |
| dst       | DCD   | 1,2,3,4,5,6,7,8,1,2,3,4,5,6,7,8,1,2,3,4 |                                                 |
|           | DCD   | 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0   |                                                 |
|           | END   |                                         |                                                 |

---



## 2.8 Using macros

A macro definition is a block of code enclosed between `MACRO` and `MEND` directives. It defines a name that can be used instead of repeating the whole 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.

See *MACRO and MEND* on page 6-29 for more details.

This section describes:

- *Test-and-branch macro example*
- *Unsigned integer division macro example* on page 2-44.

### 2.8.1 Test-and-branch macro example

Before ARMv6T2, a test-and-branch operation required two ARM instructions to implement.

You can define a macro definition 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`). You must give values to the parameters when you invoke the macro. 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

```

## 2.8.2 Unsigned integer division macro example

Example 2-10 shows a macro that performs an unsigned integer division. It takes four 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 2-10

---

```

$Lab MACRO
 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 local labels (90, 91). See *Local labels* on page 2-12 for more information.

Example 2-11 shows the code that this macro produces if it is invoked as follows:

```
ratio DivMod r0,r5,r4,r2
```

### Example 2-11

---

```

ratio
 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 ;
 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

```

---

## 2.9 Adding symbol versions

The ARM linker conforms to the *Base Platform ABI for the ARM Architecture* [BPABI] and supports the GNU-extended symbol versioning model.

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

- `name@ver` for a non default version *ver* of *name*
- `name@@ver` for a default *ver* of *name*.

The version symbols must be enclosed in vertical bars.

For example, to define a default version:

```
|my_versioned_symbol@@ver2| ; Default version
my_asm_function proc
 ...
 bx lr
endp
```

To define a non default version:

```
|my_versioned_symbol@ver1| ; Non default version
my_old_asm_function proc
 ...
 bx lr
endp
```

See the chapter describing how to access symbols in *RealView Compilation Tools v2.2 Linker and Utilities Guide* for a full description of symbol versioning in RVCT.

## 2.10 Using frame directives

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.

See *Frame directives* on page 6-37 for details of these directives.

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. See the *Using the Procedure Call Standard* appendix in *RealView Compilation Tools v2.2 Developer Guide* for more information about stack checking qualifiers.

Frame directives do not affect the code produced by `armasm`.

2.11 Assembly language changes

Table 2-10 shows the main differences between the RVCT v2.2 and earlier versions of the ARM assembly language. The old ARM syntax is still accepted by the assembler.

Table 2-10 Changes from earlier ARM assembly language

| Change                                                                                                                                                                                                                                               | Old ARM syntax                                                                                                                                                                                                                  | Preferred 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 as well as Thumb.                                                                                                                                                | STMFD <i>sp!</i> , { <i>reglist</i> }<br>LDMFD <i>sp!</i> , { <i>reglist</i> }                                                                                                                                                  | PUSH { <i>reglist</i> }<br>POP { <i>reglist</i> }                                                                                                                                                         |
| You can use the LSL, LSR, ASR, ROR, and RRX instruction mnemonics for instructions with rotations and no other operation, in ARM as well as Thumb.                                                                                                   | MOV <i>Rd</i> , <i>Rn</i> , LSL <i>shift</i><br>MOV <i>Rd</i> , <i>Rn</i> , LSR <i>shift</i><br>MOV <i>Rd</i> , <i>Rn</i> , ASR <i>shift</i><br>MOV <i>Rd</i> , <i>Rn</i> , ROR <i>shift</i><br>MOV <i>Rd</i> , <i>Rn</i> , RRX | LSL <i>Rd</i> , <i>Rn</i> , <i>shift</i><br>LSR <i>Rd</i> , <i>Rn</i> , <i>shift</i><br>ASR <i>Rd</i> , <i>Rn</i> , <i>shift</i><br>ROR <i>Rd</i> , <i>Rn</i> , <i>shift</i><br>RRX <i>Rd</i> , <i>Rn</i> |
| Use the <i>label</i> form for PC-relative addressing. Do not use the <i>offset</i> form in new code.                                                                                                                                                 | LDR <i>Rd</i> , [ <i>pc</i> , # <i>offset</i> ]                                                                                                                                                                                 | LDR <i>Rd</i> , <i>label</i>                                                                                                                                                                              |
| Specify both registers for doubleword memory accesses. You must still obey rules about the register combinations you can use.                                                                                                                        | LDRD <i>Rd</i> , <i>addr_mode</i>                                                                                                                                                                                               | LDRD <i>Rd</i> , <i>Rd2</i> , <i>addr_mode</i>                                                                                                                                                            |
| { <i>cond</i> }, if used, is always the last element of all instructions.                                                                                                                                                                            | ADD{ <i>cond</i> }S<br>LDR{ <i>cond</i> }SB                                                                                                                                                                                     | ADDS{ <i>cond</i> }<br>LDRSB{ <i>cond</i> }                                                                                                                                                               |
| You can use both ARM { <i>cond</i> } conditional forms and Thumb-2 IT instructions, in both ARM and Thumb-2 code. The assembler checks for consistency between the two, and assembles the appropriate code depending on the current instruction set. | ADDEQ <i>r1</i> , <i>r2</i> , <i>r3</i><br>LDRNE <i>r1</i> , [ <i>r2</i> , <i>r3</i> ]                                                                                                                                          | ITEQ TE<br>ADDEQ <i>r1</i> , <i>r2</i> , <i>r3</i><br>LDRNE <i>r1</i> , [ <i>r2</i> , <i>r3</i> ]                                                                                                         |

In addition, some flexibility is permitted in RVCT v2.2 that was not permitted in previous assemblers (see Table 2-11).

Table 2-11 Relaxation of requirements

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

In RVCT v2.2, you can write source code for Thumb processors using the ARM assembly language.

If you are writing Thumb code for a pre-Thumb-2 processor, 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 a Thumb-2 processor, you can minimize your code size by using 16-bit instructions wherever possible.

Table 2-12 shows the main differences between Thumb assembly language and ARM assembly language. The assembler accepts the old Thumb syntax only if it is preceded by a CODE16 directive, or if the source file is assembled with the --16 command-line option.

**Table 2-12 Differences between ARM and Thumb assembly language**

| Change                                                                                                                                         | Old Thumb syntax                                         | ARM 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-2 instructions. | ADD r1, r2, r3<br>SUB r4, r5, #6                         | ADDS r1, r2, r3<br>SUBS r4, r5, #6                         |
| The preferred form for ALU instructions specifies three registers, even if the destination register is the same as the first operand.          | ADD r7, r8<br>SUB r1, #80                                | ADD r7, r7, r8<br>SUBS r1, r1, #80                         |
| If <i>Rd</i> and <i>Rn</i> are both Lo registers, MOV <i>Rd</i> , <i>Rn</i> is disassembled as ADDS <i>Rd</i> , <i>Rn</i> , #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 <i>Rd</i> , <i>Rm</i> is disassembled as RSBS <i>Rd</i> , <i>Rm</i> , #0.                                                                  | NEG <i>Rd</i> , <i>Rm</i>                                | RSBS <i>Rd</i> , <i>Rm</i> , #0                            |
| NOP instructions replace MOV r8, r8 when available.                                                                                            | - NOP                                                    | NOP MOV r8, r8                                             |





# Chapter 3

## Assembler Reference

This chapter provides general reference material on the ARM® assemblers. It contains the following sections:

- *Command syntax* on page 3-2
- *Format of source lines* on page 3-14
- *Predefined register and coprocessor names* on page 3-15
- *Built-in variables and constants* on page 3-16
- *Symbols* on page 3-18
- *Expressions, literals, and operators* on page 3-24
- *Diagnostic messages* on page 3-36
- *Using the C preprocessor* on page 3-37.

This chapter does not explain how to write ARM assembly language. See Chapter 2 *Writing ARM Assembly Language* for tutorial information.

It also does not describe the instructions, directives, or pseudo-instructions. See the separate chapters for reference information on these.

### 3.1 Command syntax

This section relates only to `armasm`. The inline assemblers are part of the C and C++ compilers, and have no command syntax of their own.

The `armasm` command line is case-insensitive, except in filenames, and where specified.

Invoke the ARM assembler using this command:

```
armasm options {inputfile}
```

where *options* can be any combination of the following, separated by spaces:

- `--16`           Instructs the assembler to interpret instructions as Thumb® instructions, using the old Thumb syntax. This is equivalent to a `CODE16` directive at the head of the source file. Use the `--thumb` option to specify Thumb or Thumb-2 instructions using the ARM syntax.
- `--32`           Instructs the assembler to interpret instructions as ARM instructions. This is the default.
- `--apcs` , [*qualifiers*]           Specifies whether you are using the *Procedure Call Standard for the ARM Architecture* (AAPCS). It can also specify some attributes of code sections. See AAPCS on page 3-5 for details.
- `--arm`           Is a synonym for `--32`.
- `--bigend`       Instructs the assembler to assemble code suitable for a big-endian ARM. The default is `--littleend`.
- `--brief_diagnostics`           See *Controlling the output of diagnostic messages* on page 3-11.
- `--littleend`   Instructs the assembler to assemble code suitable for a little-endian ARM.
- `--checkreglist`           Instructs the assembler to check RLIST, LDM, and STM register lists to ensure that all registers are provided in increasing register number order. A warning is given if registers are not listed in order.
- `--cpu name`     Sets the target CPU. See *CPU names* on page 3-8.
- `--debug`       Instructs the assembler to generate DWARF debug tables.  
The default is DWARF2. For backwards compatibility, the `--dwarf2` option is permitted, but not required.

`--depend dependfile`

Instructs the assembler to save source file dependency lists to *dependfile*. These are suitable for use with make utilities.

`--diag_[error | remark | warning | suppress | style]`

See *Controlling the output of diagnostic messages* on page 3-11.

`--dllexport_all`

Instructs the assembler to give dynamic visibility of all global symbols, unless specified otherwise. Use this option when building DLLs.

`--dwarf2` Use with `--debug`, to instruct the assembler to generate DWARF2 debug tables. This is the default if `--debug` is specified.

`--dwarf3` Use with `--debug`, to instruct the assembler to generate DWARF3 debug tables.

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

`--md` Instructs the assembler to write source file dependency lists to *inputfile.d*.

`--errors errorfile`

Instructs the assembler to output error messages to *errorfile*.

`--exceptions` See *Controlling exception table generation* on page 3-13.

`--exceptions_unwind`

See *Controlling exception table generation* on page 3-13.

`--fpmode model`

Specifies the floating-point conformance, and sets library attributes and floating-point optimizations. See *Floating-point model* on page 3-7.

`--fpu name` Selects the target *floating-point unit* (FPU) architecture. See *FPU names* on page 3-8.

`-i{dir} [,dir]...`

Adds directories to the source file have to be fully qualified (see *GET or INCLUDE* on page 6-74).

`--keep` Instructs the assembler to keep local labels in the symbol table of the object file, for use by the debugger (see *KEEP* on page 6-78).

`--list[listingfile]`

Instructs the assembler to output a detailed listing of the assembly language produced by the assembler to *listingfile*. See *Listing output to a file* on page 3-10 for details.

`--maxcache n` Sets the maximum source cache size to *n* bytes. The default is 8MB.

`--memaccess attributes`

Specifies memory access attributes of the target memory system. See *Memory access attributes* on page 3-9.

`--no_cache` Turns off source caching. By default the assembler caches source files on the first pass and reads them from memory on the second pass.

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

`--no_exceptions`

See *Controlling exception table generation* on page 3-13.

`--no_exceptions_unwind`

See *Controlling exception table generation* on page 3-13.

`--no_hide_all`

Enables you to control symbol visibility when building SVr4 shared objects where all extern definitions are exported, and all undefined references are imported.

`--no_regs` Instructs the assembler not to predefine register names. See *Predefined register and coprocessor names* on page 3-15 for a list of predefined register names.

`--no_warn` Turns off warning messages.

`-o filename` Names the output object file. If this option is not specified, the assembler uses the second command-line argument that is not a valid command-line option as the name of the output file. If there is no such argument, the assembler creates an object filename of the form *inputfilename.o*.

`--predefine "directive"`

Instructs the assembler to pre-execute one of the SET directives. See *Pre-executing a SET directive* on page 3-9 for details.

`--split_ldm` Instructs the assembler to fault long LDM and STM instructions. See *Splitting long LDMs and STMs* on page 3-10 for details.

|                         |                                                                                                                                                                                                                                        |
|-------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>--thumb</code>    | Instructs the assembler to interpret instructions as Thumb instructions, using the ARM syntax. This is equivalent to a THUMB directive at the head of the source file.                                                                 |
| <code>--unsafe</code>   | Downgrades errors to warnings. See <i>Controlling the output of diagnostic messages</i> on page 3-11).                                                                                                                                 |
| <code>--via file</code> | Instructs the assembler to open <i>file</i> and read in command-line arguments to the assembler. For more information see the <i>Via File Syntax</i> appendix in <i>RealView Compilation Tools v2.2 Compiler and Libraries Guide</i> . |
| <i>inputfile</i>        | Specifies the input file for the assembler. Input files must be ARM or Thumb assembly language source files.                                                                                                                           |

### 3.1.1 Obtaining a list of available options

Enter the following command to obtain a summary of available assembler command-line options:

```
armasm --help
```

### 3.1.2 AAPCS

The *Procedure Call Standard for the ARM Architecture* (AAPCS) forms part of the *Application Binary Interface (ABI) for the ARM Architecture (base standard)* [BSABI] specification. By writing code that adheres to the AAPCS, you can ensure that separately compiled and assembled modules can work together.

The `--apcs` option specifies whether you are using the AAPCS. It can also specify some attributes of code sections.

For more information, see:

- the *Procedure Call Standard for the ARM Architecture* specification, `aapcs.pdf`, in `install_directory\Documentation\Specifications\...`
- the appendix describing *Using the Procedure Call Standard* in *RealView Compilation Tools v2.2 Developer Guide*.

#### ————— Note —————

AAPCS qualifiers do not affect the code produced by the assembler. They are an assertion by the programmer that the code in *inputfile* complies with a particular variant of AAPCS. They cause attributes to be set in the object file produced by the assembler. The linker uses these attributes to check compatibility of files, and to select appropriate library variants.

Values for *qualifier* are:

|                        |                                                                                                                                                                                          |
|------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| none                   | Specifies that <i>inputfile</i> does not use AAPCS. AAPCS registers are not set up. Other qualifiers are not permitted if you use none.                                                  |
| <u>/interwork</u>      | Specifies that the code in <i>inputfile</i> is suitable for ARM/Thumb interworking. See <i>RealView Compilation Tools v2.2 Developer Guide</i> for information on interworking.          |
| <u>/nointerwork</u>    | Specifies that the code in <i>inputfile</i> is not suitable for ARM/Thumb interworking. This is the default.                                                                             |
| /ropi                  | Specifies that the content of <i>inputfile</i> is read-only position-independent.                                                                                                        |
| /noropi                | Specifies that the content of <i>inputfile</i> is not read-only position-independent. This is the default.                                                                               |
| /pic                   | Is a synonym for /ropi.                                                                                                                                                                  |
| /nopic                 | Is a synonym for /noropi.                                                                                                                                                                |
| /rwpi                  | Specifies that the content of <i>inputfile</i> is read-write position-independent.                                                                                                       |
| /norwpi                | Specifies that the content of <i>inputfile</i> is not read-write position-independent. This is the default.                                                                              |
| /pid                   | Is a synonym for /rwpi.                                                                                                                                                                  |
| /nopid                 | Is a synonym for /norwpi.                                                                                                                                                                |
| /fpic                  | Specifies that the content of <i>inputfile</i> is read-only position-independent code that requires FPIC addressing.                                                                     |
| <u>/swstackcheck</u>   | Specifies that the code in <i>inputfile</i> carries out software stack-limit checking.                                                                                                   |
| <u>/noswstackcheck</u> | Specifies that the code in <i>inputfile</i> does not carry out software stack-limit checking. This is the default.                                                                       |
| /swstna                | Specifies that the code in <i>inputfile</i> is compatible both with code that carries out stack-limit checking, and with code that does not carry out stack-limit checking.              |
| /adsabi                | Specifies that the code in <i>inputfile</i> is compatible with the old ADS <i>Application Binary Interface</i> (ABI). This option is deprecated and will be removed in a future release. |

### 3.1.3 Floating-point model

There is an option to specify the floating-point model:

`--fpmode model`

Selects the target floating-point model.

*model* can be one of:

**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.

This defines the symbols:

```
__FP_IEEE
__FP_FENV_EXCEPTIONS
__FP_FENV_ROUNDING
__FP_INEXACT_EXCEPTION
```

**ieee\_fixed**

IEEE standard with round-to-nearest and no inexact exception.

This defines the symbols:

```
__FP_IEEE
__FP_FENV_EXCEPTIONS
```

**ieee\_no\_fenv**

IEEE standard with round-to-nearest and no exceptions. This mode is compatible with the Java floating-point arithmetic model.

This defines the symbol `__FP_IEEE`.

**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.

This defines the symbol `__FP_FAST`.

### 3.1.4 CPU names

There is an option to specify the CPU name:

`--cpu name` Sets the target CPU. Some instructions produce either errors or warnings if assembled for the wrong target CPU (see also *Controlling the output of diagnostic messages* on page 3-11).

Valid values for *name* are architecture names such as 4T, 5TE, or 6T2, or part numbers such as ARM7TDMI. See *ARM Architecture Reference Manual* for information about the architectures. The default is ARM7TDMI®.

See *RealView Compilation Tools v2.2 Compiler and Libraries Guide* for details of the effect on software library selection at link time.

#### Obtaining a list of valid CPU names

You can obtain a list of valid CPU names by invoking the assembler with the following command:

```
armasm --cpu list
```

### 3.1.5 FPU names

There is an option to specify the FPU name:

`--fpu name` Selects the target *floating-point unit* (FPU) architecture. If you specify this option it overrides any implicit FPU set by the `--cpu` option. Floating-point instructions produce either errors or warnings if assembled for the wrong target FPU.

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

Valid values for *name* are:

|         |                                                                                                                                                                                                    |
|---------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| none    | Selects no floating-point architecture. This makes your assembled object file compatible with any other object file.                                                                               |
| vfpv2   | Selects hardware vector floating-point unit conforming to architecture VFPv2.                                                                                                                      |
| softvfp | Selects software floating-point library (fplib). This is the default if you do not specify a <code>--fpu</code> option and the <code>--cpu</code> option selected does not imply a particular FPU. |



`softvfp+vfpv2` Selects a floating-point library with software floating-point linkage that uses vector floating-point instructions.

This is equivalent to using `--fpu vfpv2`.

See the *C and C++ Compilers* chapter in *RealView Compilation Tools v2.2 Compiler and Libraries Guide* for full details of the effect of these values on software library selection at link time.

### Obtaining a list of valid FPU names

You can obtain a list of valid FPU names by invoking the assembler with the following command:

```
armasm --fpu list
```

## 3.1.6 Memory access attributes

Use the following to specify memory access attributes of the target memory system:

`--memaccess attributes`

The default is to enable aligned loads and saves of bytes, halfwords and words. *attributes* modify the default. They can be any one of the following:

- `+L41` Enable unaligned LDRs.
- `-L22` Do not enable halfword loads.
- `-S22` Do not enable halfword stores.
- `-L22-S22` Do not enable halfword loads and stores.

## 3.1.7 Pre-executing a SET directive

You can instruct the assembler to pre-execute one of the SET directives using the following option:

`--predefine "directive"`

You must enclose *directive* in quotes. See *SETA*, *SETL*, and *SETS* on page 6-8. The assembler also executes a corresponding GBL, GBL, or GBLA directive to define the variable before setting its value.

The variable name is case-sensitive.

---

**Note**

---

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.

---

### 3.1.8 Splitting long LDMs and STMs

Use the following option to instruct the assembler to fault LDM and STM instructions with large numbers of registers:

`--split_ldm`

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

- five, for all STMs, and for LDMs that do not load the PC
- four, for LDMs that load the PC.

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.

### 3.1.9 Listing output to a file

Use the following option to list output to a file:

`--list [listingfile]`

This instructs the assembler to output a detailed listing of the assembly language produced by the assembler to *listingfile*. If `-` is given as *listingfile*, listing is sent to stdout. If no *listingfile* is given, listing is sent to *inputfile.lst*.

Use the following command-line options to control the behavior of `-list`:

- `--no_terse` Turns the terse flag off. When this option is on, lines skipped due to conditional assembly do not appear in the listing. If the terse option is off, these lines do appear in the listing. The default is on.
- `--width` Sets the listing page width. The default is 79 characters.
- `--length` Sets the listing page length. Length zero means an unpagged listing. The default is 66 lines.
- `--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.

### 3.1.10 Controlling the output of diagnostic messages

There are several options that control the output of diagnostic messages:

- `--brief_diagnostics`  
Enables or disables a mode that uses a shorter form of the diagnostic output. When enabled, 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`.
- `--diag_style {arm|ide}`  
Specifies the style used to display diagnostic messages:
  - `arm` Display messages using the ARM assembler style. This is the default if `--diag_style` is not specified.
  - `ide` Include the line number and character count for the line that is in error. These values are displayed in parentheses.
- `--diag_error tag[, tag, ...]`  
Sets the diagnostic messages that have the specified tag(s) to the error severity (see Table 3-1 on page 3-12).
- `--diag_remark tag[, tag, ...]`  
Sets the diagnostic messages that have the specified tag(s) to the remark severity (see Table 3-1 on page 3-12).
- `--diag_warning tag[, tag, ...]`  
Sets the diagnostic messages that have the specified tag(s) to the warning severity (see Table 3-1 on page 3-12).

`--diag_suppress tag[, tag, ...]`

Disables the diagnostic messages that have the specified tag(s).

`--unsafe` Enables assembly of a file containing instructions that are not available on the specified architecture and processor. It changes corresponding error messages to warning messages. It also suppresses warnings about operator precedence (see *Binary operators* on page 3-31).

Four of the `--diag_` options require a *tag*, that is the number of the message to be suppressed. More than one tag can be specified. For example, to suppress the warning messages that have numbers 1293 and 187, use the following command:

```
armasm --diag_suppress 1293,187 ...
```

A *tag* contains only digits. Omit alphabetic characters, if any, from message numbers.

Table 3-1 explains the meaning of the term *severity* used in the option descriptions.

**Table 3-1 Severity of diagnostic messages**

| Severity           | Description                                                                                                                                                                                                                                            |
|--------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Catastrophic error | Catastrophic errors indicate problems that cause the assembly to stop. These errors include command-line errors, internal errors, and missing include files. If multiple source files are being assembled, then no further source files are assembled. |
| 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.                         |

### 3.1.11 Controlling exception table generation

There are four options that control exception table generation:

- `--exceptions` Instructs the assembler to switch on exception table generation for all functions encountered.
- `--no_exceptions`  
Instructs the assembler to switch off exception table generation. No tables are generated.
- `--exceptions_unwind`  
Instructs the assembler to produce *unwind* tables for functions where possible.
- `--no_exceptions_unwind`  
Instructs the assembler to produce *nounwind* tables for every function.

For finer control, use `FRAME UNWIND ON` and `FRAME UNWIND OFF` directives, see *FRAME UNWIND ON* on page 6-49 and *FRAME UNWIND OFF* on page 6-49.

#### Unwind tables

A *function* is code encased by `PROC/ENDP` or `FUNC/ENDFUNC` directives.

An exception can propagate through a function with an *unwind* table. The assembler generates the unwinding information from debug frame information.

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 functions only if the function contains sufficient `FRAME` directives to describe the use of the stack within the function. 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.

## 3.2 Format of source lines

The general form of source lines in an ARM assembly language module is:

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

All three sections of the source line are optional.

Instructions cannot start in the first column. They must be preceded by white space even if there is no preceding symbol.

You can write directives in all uppercase, as in this manual. Alternatively, you can write directives in all lowercase. You must not write a directive in mixed upper and lowercase.

You can use blank lines to make your code more readable.

*symbol* is usually a label (see *Labels* on page 3-21). 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 and cannot contain any white space character such as a space or a tab (see *Symbol naming rules* on page 3-18).

### 3.3 Predefined register and coprocessor names

All register and coprocessor names are case-sensitive.

#### 3.3.1 Predeclared register names

The following register names are predeclared:

- `r0-r15` and `R0-R15`
- `a1-a4` (argument, result, or scratch registers, synonyms for `r0` to `r3`)
- `v1-v8` (variable registers, `r4` to `r11`)
- `sb` and `SB` (static base, `r9`)
- `s1` and `SL` (stack limit, `r10`)
- `fp` and `FP` (frame pointer, `r11`)
- `ip` and `IP` (intra-procedure-call scratch register, `r12`)
- `sp` and `SP` (stack pointer, `r13`)
- `lr` and `LR` (link register, `r14`)
- `pc` and `PC` (program counter, `r15`).

#### 3.3.2 Predeclared program status register names

The following program status register names are predeclared:

- `cpsr` and `CPSR` (current program status register)
- `spsr` and `SPSR` (saved program status register).

#### 3.3.3 Predeclared floating-point register names

The following floating-point register names are predeclared:

- `s0-s31` and `S0-S31` (VFP single-precision registers)
- `d0-d15` and `D0-D15` (VFP double-precision registers).

———— **Note** —————

Use of FPA registers `f0-f7` and `F0-F7` is obsolete.

---

#### 3.3.4 Predeclared coprocessor names

The following coprocessor names and coprocessor register names are predeclared:

- `p0-p15` (coprocessors 0-15)
- `c0-c15` (coprocessor registers 0-15).

### 3.4 Built-in variables and constants

Table 3-2 lists the built-in variables defined by the ARM assembler.

**Table 3-2 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. See also <i>Determining the armasm version at assembly time</i> on page 3-17.                                                    |
| 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 cpu. 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 value True if /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 value True if /inter is set. The default is False.                                                                                                                          |
| {LINENUM}        | Holds an integer indicating the line number in the current source file.                                                                                                             |
| {NOSWST}         | Has the value True if /noswst is set. The default is False.                                                                                                                         |
| {OPT}            | Value of the currently-set listing option. The OPT directive can be used 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 CPU or architecture specified. |
| {ROPI}           | Has the value True if /ropi is set. The default is False.                                                                                                                           |
| {RWPI}           | Has the value True if /rwpi is set. The default is False.                                                                                                                           |
| {SWST}           | Has the value True if /swst is set. The default is False.                                                                                                                           |
| {VAR} or @       | Current value of the storage area location counter.                                                                                                                                 |



Built-in variables cannot be set using the SETA, SETL, or SETS directives. They can be used in expressions or conditions, for example:

```
IF {ARCHITECTURE} = "4T"
```

|ads\$version| must be all lowercase. The other built-in variables can be uppercase, lowercase, or mixed.

Table 3-3 lists the built-in Boolean constants defined by the ARM assembler.

**Table 3-3 Built-in Boolean constants**

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

### 3.4.1 Determining the armasm version at assembly time

You can use the built-in variable {ARMASM\$VERSION} to distinguish between versions of armasm. However, previous (SDT) versions of armasm did not have this built-in variable.

If you have to build both RVCT and SDT versions of your code, you can test for the built-in variable |ads\$version|. Use code similar to the following:

```
IF :DEF: |ads$version|
 ; code for RVCT or ADS
ELSE
 ; code for SDT
ENDIF
```

## 3.5 Symbols

You can use symbols to represent variables, addresses, and numeric constants. Symbols representing addresses are also called *labels*. See:

- *Symbol naming rules*
- *Variables* on page 3-19
- *Numeric constants* on page 3-19
- *Assembly time substitution of variables* on page 3-19
- *Labels* on page 3-21
- *Local labels* on page 3-22.

### 3.5.1 Symbol naming rules

The following general rules apply to symbol names:

- You can use uppercase letters, lowercase letters, numeric characters, or the underscore character in symbol names.
- Do not use numeric characters for the first character of symbol names, except in local labels (see *Local labels* on page 3-22).
- Symbol names are case-sensitive.
- All characters in the symbol name are significant.
- Symbol names must be unique within their scope.
- Symbols must not use built-in variable names or predefined symbol names (see *Predefined register and coprocessor names* on page 3-15 and *Built-in variables and constants* on page 3-16).
- Symbols must not use the same name as instruction mnemonics or directives. 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.

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.

### 3.5.2 Variables

The value of a variable can be changed as assembly proceeds. Variables are of three types:

- numeric
- logical
- string.

The type of a variable cannot be changed.

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 (see *Numeric constants* and *Numeric expressions* on page 3-25).

The possible values of a logical variable are {TRUE} or {FALSE} (see *Logical expressions* on page 3-28).

The range of possible values of a string variable is the same as the range of values of a string expression (see *String expressions* on page 3-24).

Use the GBLA, GBL, GBLS, LCLA, LCLL, and LCLS directives to declare symbols representing variables, and assign values to them using the SETA, SETL, and SETS directives. See:

- *GBLA, GBL, and GBLS* on page 6-5
- *LCLA, LCLL, and LCLS* on page 6-7
- *SETA, SETL, and SETS* on page 6-8.

### 3.5.3 Numeric constants

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  $2^3-1$ . However, the assembler makes no distinction between  $-n$  and  $2^{32}-n$ . Relational operators such as  $\geq$  use the unsigned interpretation. This means that  $0 > -1$  is {FALSE}.

Use the EQU directive to define constants (see *EQU* on page 6-68). You cannot change the value of a numeric constant after you define it.

See also *Numeric expressions* on page 3-25 and *Numeric literals* on page 3-26.

### 3.5.4 Assembly time substitution of variables

You can use a string variable for a whole line of assembly language, or any part of a line. Use the variable with a \$ prefix in the places where the value is to be substituted for the variable. The dollar character instructs the assembler to substitute the string into the source code line before checking the syntax of the line.

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 (see *Symbol naming rules* on page 3-18). 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.

## Examples

```

; 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

```

### 3.5.5 Labels

Labels are symbols representing the addresses in memory of instructions or data. They can be program-relative, register-relative, or absolute.

#### Program-relative labels

These represent the PC, plus or minus a numeric constant. Use them as targets for branch instructions, or to access small items of data embedded in code sections. You can define program-relative labels using a label on an instruction or on one of the data definition directives. See:

- *DCB* on page 6-19
- *DCD and DCDU* on page 6-20
- *DCFD and DCFDU* on page 6-22
- *DCFS and DCFSU* on page 6-23
- *DCI* on page 6-24
- *DCQ and DCQU* on page 6-25
- *DCW and DCWU* on page 6-26.

#### Register-relative labels

These represent a named register plus a numeric constant. They are most often used to access data in data sections. You can define them with a storage map. You can use the *EQU* directive to define additional register-relative labels, based on labels defined in storage maps. See:

- *MAP* on page 6-16
- *SPACE* on page 6-18
- *DCDO* on page 6-21
- *EQU* on page 6-68.

#### Absolute addresses

These are numeric constants. They are integers in the range 0 to  $2^{32}-1$ . They address the memory directly.

### 3.5.6 Local labels

A local label is a number in the range 0-99, optionally followed by a name. The same number can be used for more than one local label in an ELF section.

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 in macros (see *MACRO* and *MEND* on page 6-29).

Use the *ROUT* directive to limit the scope of local labels (see *ROUT* on page 6-83). A reference to a local label refers to a matching label within the same scope. If there is no matching label within the scope in either direction, the assembler generates an error message and the assembly fails.

You can use the same number for more than one local label even within the same scope. By default, the assembler links a local label reference to:

- the most recent local label of the same number, if there is one within the scope
- the next following local label of the same number, if there is not a preceding one within the scope.

Use the optional parameters to modify this search pattern if required.

#### Syntax

The syntax of a local label is:

*n*{*rouname*}

The syntax of a reference to a local label is:

%{F|B}{A|T}*n*{*rouname*}

where:

|                |                                                           |
|----------------|-----------------------------------------------------------|
| <i>n</i>       | is the number of the local label.                         |
| <i>rouname</i> | is the name of the current scope.                         |
| %              | introduces the reference.                                 |
| F              | instructs the assembler to search forwards only.          |
| B              | instructs the assembler to search backwards only.         |
| A              | instructs the assembler to search all macro levels.       |
| T              | instructs the assembler to look at this macro level only. |

If neither F nor B is specified, the assembler searches backwards first, then forwards.

If neither A nor T is specified, the assembler searches all macros from the current level to the top level, but does not search lower level macros.

If *route*name is specified in either a label or a reference to a label, the assembler checks it against the name of the nearest preceding ROUT directive. If it does not match, the assembler generates an error message and the assembly fails.

## 3.6 Expressions, literals, and operators

This section contains the following subsections:

- *String expressions*
- *String literals* on page 3-25
- *Numeric expressions* on page 3-25
- *Numeric literals* on page 3-26
- *Floating-point literals* on page 3-27
- *Register-relative and program-relative expressions* on page 3-28
- *Logical expressions* on page 3-28
- *Logical literals* on page 3-28
- *Operator precedence* on page 3-28
- *Unary operators* on page 3-30
- *Binary operators* on page 3-31.

### 3.6.1 String expressions

String expressions consist of combinations of string literals, string variables, string manipulation operators, and parentheses. See:

- *Variables* on page 3-19
- *String literals* on page 3-25
- *Unary operators* on page 3-30
- *String manipulation operators* on page 3-32
- *SETA, SETL, and SETS* on page 6-8.

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 512 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
```



### 3.6.2 String literals

String literals consist of a series of characters contained between double quote characters. The length of a string literal is restricted by the length of the input line (see *Format of source lines* on page 3-14).

To include a double quote character or a dollar character in a string, use two of the character.

C string escape sequences are also enabled, unless `--no_esc` is specified (see *Command syntax* on page 3-2).

#### Examples

```
abc SETS "this string contains only one "" double quote"
def SETS "this string contains only one $$ dollar symbol"
```

### 3.6.3 Numeric expressions

Numeric expressions consist of combinations of numeric constants, numeric variables, ordinary numeric literals, binary operators, and parentheses. See:

- *Numeric constants* on page 3-19
- *Variables* on page 3-19
- *Numeric literals* on page 3-26
- *Binary operators* on page 3-31
- *SETA, SETL, and SETS* on page 6-8.

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, 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}.

#### Example

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

### 3.6.4 Numeric literals

Numeric literals can take any of the following forms:

*decimal-digits*

*0xhexadecimal-digits*

*&hexadecimal-digits*

*n\_base-n-digits*

*'character'*

where:

*decimal-digits* Is a sequence of characters using only the digits 0 to 9.

*hexadecimal-digits* Is a sequence of characters using only the digits 0 to 9 and the letters A to F or a to f.

*n\_* Is a single digit between 2 and 9 inclusive, followed by an underscore character.

*base-n-digits* Is a sequence of characters using only the digits 0 to (*n*−1)

*character* Is any single character except a single quote. Use \ if you require a single quote. 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
```

### 3.6.5 Floating-point literals

Floating-point literals can take any of the following forms:

*{-}digitsE{-}digits*

*{-}{digits}.digits{E{-}digits}*

*0xhexdigits*

*&hexdigits*

where:

- |                  |                                                                                                                                                                                                                                                                                                   |
|------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>digits</i>    | 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.                                                                                                                                    |
| <i>hexdigits</i> | 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 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.

#### Examples

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

### 3.6.6 Register-relative and program-relative expressions

A register-relative expression evaluates to a named register plus or minus a numeric constant (see *MAP* on page 6-16).

A program-relative expression evaluates to the *Program Counter* (PC), plus or minus a numeric constant. It is normally a label combined with a numeric expression.

#### Example

```

 LDR r4,=data+4*n ; n is an assembly-time variable
 ; code
 MOV pc,lr
data DCD value0
 ; n-1 DCD directives
 DCD valuen ; data+4*n points here
 ; more DCD directives

```

### 3.6.7 Logical expressions

Logical expressions consist of combinations of logical literals ({TRUE} or {FALSE}), logical variables, Boolean operators, relations, and parentheses (see *Boolean operators* on page 3-35).

Relations consist of combinations of variables, literals, constants, or expressions with appropriate relational operators (see *Relational operators* on page 3-34).

### 3.6.8 Logical literals

The logical literals are:

- {TRUE}
- {FALSE}.

### 3.6.9 Operator precedence

The assembler includes an extensive set of operators for use in expressions. Many of the operators resemble their counterparts in high-level languages such as C (see *Unary operators* on page 3-30 and *Binary operators* on page 3-31).

There is a strict order of precedence in their evaluation:

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.

## Operator precedence in armasm and C

The assembler order of precedence is not exactly the same as in C.

For example,  $(1 + 2 :SHR; 3)$  evaluates as  $(1 + (2 :SHR; 3)) = 1$  in armasm. The equivalent expression in C evaluates as  $((1 + 2) >> 3) = 0$ .

You are recommended to use brackets to make the precedence explicit.

If your code contains an expression that would parse differently in C, armasm normally gives a warning:

A1466W: Operator precedence means that expression would evaluate differently in C

However, the warning is not given if you use the `-unsafe` command-line option.

Table 3-4 shows the order of precedence of operators in armasm, and a comparison with the order in C (see Table 3-5 on page 3-30).

From these 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.

**Table 3-4 Operator precedence in armasm**

| armasm precedence       | equivalent C operators |
|-------------------------|------------------------|
| unary operators         | unary operators        |
| * / :MOD:               | * / %                  |
| string manipulation     | n/a                    |
| :SHL: :SHR: :ROR: :ROL: | << >>                  |
| + - :AND: :OR: :EOR:    | + - &                  |
| = > >= < <= /= <>       | == > >= < <= !=        |
| :LAND: :LOR: :LEOR:     | &&                     |

Table 3-5 Operator precedence in C

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

3.6.10 Unary operators

Unary operators have the highest precedence and are evaluated first. A unary operator precedes its operand. Adjacent operators are evaluated from right to left.

Table 3-6 lists the unary operators that return strings.

Table 3-6 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.                                                                                |
| :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.                                                        |

Table 3-7 lists the unary operators that return numeric values.

**Table 3-7 Unary operators that return numeric or logical values**

| Operator      | Usage                  | Description                                                                                                                                  |
|---------------|------------------------|----------------------------------------------------------------------------------------------------------------------------------------------|
| ?             | ?A                     | Number of bytes of executable code generated by line defining symbol A.                                                                      |
| + and -       | +A<br>-A               | Unary plus. Unary minus. + and - can act on numeric and program-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.                                                                                |
| :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. <sup>a</sup>                                                                                                        |
| :RCONST:      | :RCONST:Rn             | Number of register, 0-15 corresponding to r0-r15.                                                                                            |

a. ~ is an alias for :NOT:, for example, ~A.

### 3.6.11 Binary operators

Binary operators are written between the pair of subexpressions they operate on.

Binary operators have lower precedence than unary operators. Binary operators appear in this section in order of precedence.

#### **Note**

The order of precedence is not the same as in C, see *Operator precedence in armasm and C* on page 3-29.

## Multiplicative operators

Multiplicative operators have the highest precedence of all binary operators. They act only on numeric expressions.

Table 3-8 shows the multiplicative operators.

**Table 3-8 Multiplicative operators**

| Operator | Alias | Usage   | Explanation |
|----------|-------|---------|-------------|
| *        |       | A*B     | Multiply    |
| /        |       | A/B     | Divide      |
| :MOD:    | %     | A:MOD:B | A modulo B  |

## String manipulation operators

Table 3-9 shows the string manipulation operators.

In the slicing operators LEFT and RIGHT:

- A must be a string
- B must be a numeric expression.

In CC, A and B must both be strings.

**Table 3-9 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 |



## Shift operators

Shift operators act on numeric expressions, shifting or rotating the first operand by the amount specified by the second.

Table 3-10 shows the shift operators.

**Table 3-10 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.

## Addition, subtraction, and logical operators

Addition and subtraction operators act on numeric expressions.

Logical operators act on numeric expressions. The operation is performed *bitwise*, that is, independently on each bit of the operands to produce the result.

Table 3-11 shows addition, subtraction, and logical operators.

**Table 3-11 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           |

## Relational operators

Table 3-12 shows the relational operators. These act on two operands of the same type to produce a logical value.

The operands can be one of:

- numeric
- program-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}.

**Table 3-12 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             |

## Boolean operators

These are the operators with the lowest precedence. They perform the standard logical operations on their operands.

In all three cases both A and B must be expressions that evaluate to either {TRUE} or {FALSE}.

Table 3-13 shows the Boolean operators.

**Table 3-13 Boolean operators**

| Operator | 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           |

## 3.7 Diagnostic messages

The assembler can give a range of diagnostic messages. You can control what messages the assembler gives using command-line options. See *Controlling the output of diagnostic messages* on page 3-11 for details.

### 3.7.1 Interlocks

You can obtain warning messages about possible interlocks in your code caused by the pipeline of the core chosen by the `--cpu` option. To do this, use the following command-line option when invoking the assembler:

```
armasm --diag_warning 1563
```

This warning is off by default.

## 3.8 Using the C preprocessor

You can include the C preprocessor command `#include` in your assembly language source file. If you do this, you must preprocess the file using the C preprocessor, before using `armasm` to assemble it. See *RealView Compilation Tools v2.2 Compiler and Libraries Guide*.

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

Example 3-1 shows the commands you write to preprocess and assemble a file, `source.s`. In this example, the preprocessor outputs a file called `preprocessed.s`, and `armasm` assembles `preprocessed.s`.

### Example 3-1 Preprocessing an assembly language source file

---

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

---



# Chapter 4

## ARM and Thumb Instructions

This chapter describes the ARM®, Thumb®-2, and Thumb instructions (32-bit and 16-bit) supported by the ARM assembler. It contains the following sections:

- *Instruction summary* on page 4-2
- *Memory access instructions* on page 4-7
- *General data processing instructions* on page 4-41
- *Multiply instructions* on page 4-73
- *Saturating instructions* on page 4-95
- *Parallel instructions* on page 4-100
- *Packing and unpacking instructions* on page 4-108
- *Branch instructions* on page 4-115
- *Coprocessor instructions* on page 4-121
- *Miscellaneous instructions* on page 4-131
- *Pseudo-instructions* on page 4-144.

## 4.1 Instruction summary

Table 4-1 gives an overview of the instructions available in the ARM, Thumb, and Thumb-2 instruction sets. Use it to locate individual instructions and pseudo-instructions described in the rest of this chapter.

The § column in Table 4-1 shows the ARM architecture that each instruction first appeared in. 6T2 indicates that an instruction first appeared in 6T2, but is available in the ARM instruction set as well as the Thumb-2 instruction set. T2 indicates that the instruction is only available in the Thumb-2 instruction set. XScale indicates that the instruction is available only on XScale processors.

**Table 4-1 Location of instructions**

| <b>Mnemonic</b>         | <b>Brief description</b>                                 | <b>Page</b> | <b>§</b> |
|-------------------------|----------------------------------------------------------|-------------|----------|
| ADC, ADD                | Add with Carry, Add                                      | page 4-45   | All      |
| ADR pseudo-instruction  | Load program or register-relative address (short range)  | page 4-145  | All      |
| ADRL pseudo-instruction | Load program or register-relative address (medium range) | page 4-147  | All      |
| AND                     | Logical AND                                              | page 4-50   | All      |
| ASR                     | Arithmetic Shift Right                                   | page 4-67   | All      |
| B                       | Branch                                                   | page 4-116  | All      |
| BFC, BFI                | Bit Field Clear and Insert                               | page 4-53   | 6T2      |
| BIC                     | Bit Clear                                                | page 4-50   | All      |
| BKPT                    | Breakpoint                                               | page 4-132  | 5        |
| BL                      | Branch with Link                                         | page 4-116  | All      |
| CDP, CDP2               | Coprocessor Data Processing operation                    | page 4-122  | All, 5   |
| CLREX                   | Clear Exclusive                                          | page 4-39   | T2       |
| CLZ                     | Count leading zeros                                      | page 4-54   | 5        |
| CMN, CMP                | Compare Negative, Compare                                | page 4-55   | All      |
| CPS                     | Change Processor State                                   | page 4-137  | 6        |
| CPY                     | Copy                                                     | page 4-57   | 6        |
| CBZ, CBNZ               | Compare and Branch if {Non}Zero                          | page 4-119  | T2       |
| EOR                     | Exclusive OR                                             | page 4-50   | All      |



**Table 4-1 Location of instructions (continued)**

| <b>Mnemonic</b>          | <b>Brief description</b>                   | <b>Page</b> | <b>§</b>      |
|--------------------------|--------------------------------------------|-------------|---------------|
| IT                       | If-Then                                    | page 4-70   | T2            |
| LDC, LDC2                | Load Coprocessor                           | page 4-127  | All, 5        |
| LDM                      | Load Multiple registers                    | page 4-27   | All           |
| LDR                      | Load Register instructions                 | page 4-7    | All           |
| LDR pseudo-instruction   | Load Register pseudo-instruction           | page 4-151  | All           |
| LDREX                    | Load Register Exclusive                    | page 4-36   | 6             |
| LSL, LSR                 | Logical Shift Left, Logical Shift Right    | page 4-67   | All           |
| MAR                      | Move from Registers to 40-bit Accumulator  | page 4-143  | XScale        |
| MCR, MCR2, MCRR, MCRR2   | Move from Register(s) to Coprocessor       | page 4-123  | All, 5, 5E, 6 |
| MIA, MIAPH, MIAxy        | Multiply with Internal 40-bit Accumulate   | page 4-93   | XScale        |
| MLA, MLS                 | Multiply Accumulate, Multiply and Subtract | page 4-74   | All, 6T2      |
| MOV                      | Move                                       | page 4-57   | All           |
| MOVT                     | Move Top                                   | page 4-60   | 6T2           |
| MOV32 pseudo-instruction | Move 32-bit constant to register           | page 4-149  | T2            |
| MRA                      | Move from 40-bit Accumulator to Registers  | page 4-143  | XScale        |
| MRC, MRC2                | Move from Coprocessor to Register          | page 4-125  | All, 5        |
| MRS                      | Move from PSR to register                  | page 4-134  | All           |
| MSR                      | Move from register to PSR                  | page 4-135  | All           |
| MUL                      | Multiply                                   | page 4-74   | All           |
| MVN                      | Move Not                                   | page 4-57   | All           |
| NOP                      | No Operation                               | page 4-141  | All           |
| ORN                      | Logical OR NOT                             | page 4-50   | T2            |
| ORR                      | Logical OR                                 | page 4-50   | All           |
| PKHBT, PKHTB             | Pack Halfwords                             | page 4-113  | 6             |

Table 4-1 Location of instructions (continued)

| Mnemonic                                          | Brief description                                                                               | Page       | §      |
|---------------------------------------------------|-------------------------------------------------------------------------------------------------|------------|--------|
| PUSH, POP                                         | PUSH, POP registers                                                                             | page 4-30  | All T  |
| QADD, QDADD, QDSUB, QSUB                          | Saturating Arithmetic                                                                           | page 4-96  | 5ExP   |
| QADD8, QADD16, QASX,<br>QSUB8, QSUB16, QSAX       | Parallel signed Saturating Arithmetic                                                           | page 4-101 | 6      |
| REV, REV16, REVSH, RBIT                           | Reverse byte order, Reverse Bits                                                                | page 4-65  | 6, 6T2 |
| RFE                                               | Return From Exception                                                                           | page 4-32  | 6T2    |
| ROR                                               | Rotate Right Register                                                                           | page 4-67  | All    |
| RSB, RSC, SBC                                     | Reverse Sub, Reverse Sub with Carry, Sub with Carry                                             | page 4-45  | All    |
| SADD8, SADD16, SASX                               | Parallel signed arithmetic                                                                      | page 4-101 | 6      |
| SBFX, UBFX                                        | Signed, Unsigned Bit Field extract                                                              | page 4-109 | 6T2    |
| SEL                                               | Select bytes according to CPSR GE flags                                                         | page 4-63  | 6      |
| SEV                                               | Set Event                                                                                       | page 4-141 | K      |
| SETEND                                            | Set Endianness for memory accesses                                                              | page 4-140 | 6      |
| SHADD8, SHADD16, SHASX,<br>SHSUB8, SHSUB16, SHSAX | Parallel signed Halving arithmetic                                                              | page 4-101 | 6      |
| SMTI                                              | Secure Monitor Interrupt                                                                        | page 4-139 | Z      |
| SMLAD                                             | Dual Signed Multiply Accumulate ( $32 \leq 32 + 16 \times 16 + 16 \times 16$ )                  | page 4-88  | 6      |
| SMLAL                                             | Signed Multiply Accumulate ( $64 \leq 64 + 32 \times 32$ )                                      | page 4-76  | M      |
| SMLALxy                                           | Signed Multiply Accumulate ( $64 \leq 64 + 16 \times 16$ )                                      | page 4-82  | 5ExP   |
| SMLALD                                            | Dual Signed Multiply Accumulate Long<br>( $64 \leq 64 + 16 \times 16 + 16 \times 16$ )          | page 4-90  | 6      |
| SMLSD                                             | Dual Signed Multiply Subtract Accumulate<br>( $32 \leq 32 + 16 \times 16 - 16 \times 16$ )      | page 4-88  | 6      |
| SMLSXD                                            | Dual Signed Multiply Subtract Accumulate Long<br>( $64 \leq 64 + 16 \times 16 - 16 \times 16$ ) | page 4-90  | 6      |
| SMMUL                                             | Signed top word Multiply ( $32 \leq \text{TopWord}(32 \times 32)$ )                             | page 4-86  | 6      |

**Table 4-1 Location of instructions (continued)**

| <b>Mnemonic</b>                                   | <b>Brief description</b>                                                             | <b>Page</b> | <b>§</b> |
|---------------------------------------------------|--------------------------------------------------------------------------------------|-------------|----------|
| SMUAD, SMUSD                                      | Dual Signed Multiply, and Add or Subtract products                                   | page 4-84   | 6        |
| SMULL                                             | Signed Multiply ( $64 \leq 32 \times 32$ )                                           | page 4-76   | M        |
| SMULxy                                            | Signed Multiply ( $32 \leq 16 \times 16$ )                                           | page 4-78   | 5ExP     |
| SMULWy                                            | Signed Multiply ( $32 \leq 32 \times 16$ )                                           | page 4-80   | 5ExP     |
| SRS                                               | Store Return State                                                                   | page 4-34   | 6T2      |
| SSAT                                              | Signed Saturate                                                                      | page 4-98   | 6        |
| SSAT16                                            | Signed Saturate, parallel halfwords                                                  | page 4-106  | 6        |
| SSUB8, SSUB16, SSAX                               | Parallel signed arithmetic                                                           | page 4-101  | 6        |
| STC                                               | Store Coprocessor                                                                    | page 4-127  | All      |
| STC2                                              | Store Coprocessor                                                                    | page 4-129  | 5ExP     |
| STM                                               | Store Multiple registers                                                             | page 4-27   | All      |
| STR                                               | Store Register instructions                                                          | page 4-7    | All      |
| STREX                                             | Store Register Exclusive                                                             | page 4-36   | 6        |
| SUB                                               | Subtract                                                                             | page 4-45   | All      |
| SUBS PC, LR                                       | Exception return, no stack                                                           | page 4-49   | T2       |
| SWI                                               | Software Interrupt                                                                   | page 4-133  | All      |
| SWP, SWPB                                         | Swap registers and memory (ARM only)                                                 | page 4-40   | All      |
| SXT, SXTA                                         | Signed extend, with optional Addition                                                | page 4-110  | 6        |
| TBB, TBH                                          | Table Branch Byte, Halfword                                                          | page 4-120  | T2       |
| TEQ, TST                                          | Test Equivalence, Test                                                               | page 4-61   | All      |
| UADD8, UADD16, UASX                               | Parallel Unsigned Arithmetic                                                         | page 4-101  | 6        |
| UHADD8, UHADD16, UHASX,<br>UHSUB8, UHSUB16, UHSAX | Parallel Unsigned Halving Arithmetic                                                 | page 4-101  | 6        |
| UMAAL                                             | Unsigned Multiply Accumulate Accumulate Long<br>( $64 \leq 32 + 32 + 32 \times 32$ ) | page 4-92   | 6        |

Table 4-1 Location of instructions (continued)

| Mnemonic                                          | Brief description                                                                                     | Page       | §   |
|---------------------------------------------------|-------------------------------------------------------------------------------------------------------|------------|-----|
| UMLAL, UMULL                                      | Unsigned Multiply Accumulate, Multiply<br>( $64 \leq 32 \times 32 + 64$ ), ( $64 \leq 32 \times 32$ ) | page 4-76  | M   |
| UQADD8, UQADD16, UQASX,<br>UQSUB8, UQSUB16, UQSAX | Parallel Unsigned Saturating Arithmetic                                                               | page 4-101 | 6   |
| USAD8                                             | Unsigned Sum of Absolute Differences                                                                  | page 4-104 | 6   |
| USADA8                                            | Accumulate Unsigned Sum of Absolute Differences                                                       | page 4-104 | 6   |
| USAT                                              | Unsigned Saturate                                                                                     | page 4-98  | 6   |
| USAT16                                            | Unsigned Saturate, parallel halfwords                                                                 | page 4-106 | 6   |
| USUB8, USUB16, USAX                               | Parallel unsigned arithmetic                                                                          | page 4-101 | 6   |
| UXT, UXTA                                         | Unsigned extend, with optional Addition                                                               | page 4-110 | 6   |
| WFE, WFI, YIELD                                   | Wait For Event, Wait For Interrupt, Yield                                                             | page 4-141 | 6T2 |

## 4.2 Memory access instructions

This section contains the following subsections:

- *Address alignment* on page 4-8  
Alignment considerations that apply to all memory access instructions.
- *LDR and STR (zero, immediate, or pre-indexed immediate offset)* on page 4-9  
Load and store bytes, halfwords, words and doublewords.
- *LDR and STR (post-indexed immediate offset)* on page 4-13  
Load and store bytes, halfwords, words and doublewords.
- *LDR and STR (register or pre-indexed register offset)* on page 4-16  
Load and store bytes, halfwords, words and doublewords.
- *LDR and STR (post-indexed register offset)* on page 4-19  
Load and store bytes, halfwords, words and doublewords.
- *LDR (PC-relative)* on page 4-22  
Load bytes, halfwords, words and doublewords.
- *PLD* on page 4-25  
Preload an address for the future.
- *LDM and STM* on page 4-27  
Load and Store Multiple Registers.
- *PUSH and POP* on page 4-30  
Push low registers, and optionally the LR, onto the stack.  
Pop low registers, and optionally the PC, off the stack.
- *RFE* on page 4-32  
Return From Exception.
- *SRS* on page 4-34  
Store Return State.
- *LDREX and STREX* on page 4-36  
Load and Store Register Exclusive.
- *CLREX* on page 4-39  
Clear Exclusive.

- *SWP and SWPB* on page 4-40  
Swap data between registers and memory.

---

**Note**

---

There is also an LDR pseudo-instruction (see *LDR pseudo-instruction* on page 4-151). This pseudo-instruction either assembles to an LDR instruction, or to a MOV or MVN instruction.

---

## 4.2.1 Address alignment

In most circumstances, you must ensure that addresses for 4-byte transfers are 4-byte word-aligned, and addresses for 2-byte transfers are 2-byte aligned. In ARMv7, there is a configuration option to permit unaligned accesses.

In ARMv6 and below, if your system has a system coprocessor (cp15), you can enable alignment checking. Non word-aligned 32-bit transfers cause an alignment exception if alignment checking is enabled.

If your system does not have a system coprocessor (cp15), or alignment checking is disabled:

- For STR, the specified address is rounded down to a multiple of four.
- For LDR:
  1. The specified address is rounded down to a multiple of four.
  2. Four bytes of data are loaded from the resulting address.
  3. 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.

## 4.2.2 LDR and STR (zero, immediate, or pre-indexed immediate offset)

Load and Store. Byte and halfword loads are zero-extended or sign-extended to 32 bits.

### ———— Note ————

Also, see *Pseudo-instructions* on page 4-144.

### Syntax

*op*{*type*}{*T*}{*cond*} *Rd*, {*Rd2*,} [*Rn* {, #*offset*}]{!}

where:

*op* can be any one of:

|     |                 |
|-----|-----------------|
| LDR | Load Register   |
| STR | Store Register. |

*type* can be any one of:

|    |                            |
|----|----------------------------|
| B  | unsigned Byte              |
| SB | Signed Byte (LDR only)     |
| H  | unsigned Halfword          |
| SH | Signed Halfword (LDR only) |
| -  | omitted, for Word          |
| D  | Doubleword.                |

*T* is an optional suffix. If *T* is present, the memory system treats the access as though the processor was in User mode, even if it is in a privileged mode (see *Processor mode* on page 2-4). *T* has no effect in User mode.

*cond* is an optional condition code (see *Conditional execution* on page 2-17).

*Rd* is the ARM register to load or save.

*Rd2* is the second ARM register to load or save (*type* == D only).

*Rn* is the register on which the memory address is based.

*offset* is an immediate offset. If *offset* is omitted, the instruction is a zero offset instruction.

! is an optional suffix. If ! is present, the instruction is a pre-indexed instruction, and *Rn* must not be the same register as *Rd* or *Rd2*.

### Zero offset

The value in *Rn* is used as the address for the transfer.

The T suffix is not available for Doubleword instructions.

### Immediate offset

The offset is applied to the value in *Rn* before the data transfer takes place. The result is used as the memory address for the transfer. The range of offsets allowed is:

- –4095 to +4095 for ARM Word or Byte instructions.
- –255 to +255 for ARM Signed Byte, Halfword, Signed Halfword, and Doubleword instructions.
- –255 to +4095 for all Thumb-2 instructions without the T suffix, except Doubleword instructions.
- –1020 to +1020 for Thumb-2 Doubleword instructions. Must be a multiple of 4.
- 0 to +255 for Thumb-2 instructions with the T suffix.

The T suffix is not available for ARM instructions, or for Thumb-2 Doubleword instructions. It is available for all other Thumb-2 instructions.

### Pre-indexed immediate offset

The offset is applied to the value in *Rn* before the data transfer takes place. The result is used as the memory address for the transfer. The result is written back into *Rn*.

The range of offsets allowed is:

- –4095 to +4095 for ARM Word or Byte instructions.
- –255 to +255 for ARM Signed Byte, Halfword, Signed Halfword, and Doubleword instructions.
- –255 to +255 for all Thumb-2 instructions except Doubleword instructions.
- –1020 to +1020 for Thumb-2 Doubleword instructions. Must be a multiple of 4.

The T suffix is not available for ARM instructions or Thumb-2 instructions.



## Doubleword register restrictions

For Thumb-2 instructions, you must not specify r15 for either *Rd* or *Rd2*.

For ARM instructions:

- *Rd* must be an even-numbered register
- *Rd* must not be r14
- *Rd2* must be  $R(d + 1)$ .

## 16-bit instructions

16-bit versions of a subset of these instructions are available in Thumb-2 code, and in Thumb code on other Thumb-capable processors.

The following restrictions apply to 16-bit instructions:

- Only zero offset and immediate offset instructions are available. Pre-indexed immediate offset instructions are not available.
- Use of the T suffix is not allowed.
- If *Rn* is not *r13*:
  - *Rd* and *Rn* must both be Lo registers
  - for a Word instruction, offset must be in the range 0 to +124, and must be divisible by 4
  - for a Halfword instruction, offset must be in the range 0 to +62, and must be divisible by 2
  - for a Byte instruction, offset must be in the range 0 to +31
  - Signed Byte, Signed Halfword, and Doubleword instructions are not available.
- If *Rn* is *r13*:
  - *Rd* must be a Lo register
  - the instruction must be a Word instruction
  - offset must be in the range 0 to +1020, and must be divisible by 4.

## Loading to r15

*Rd* can be the PC, in either ARM or Thumb-2 code. In this case, *type* must be omitted.

A load to r15 (pc) causes a branch to the instruction at the address loaded.

In ARMv4, bits[1:0] of the value loaded must be zero.

In ARMv5 and above:

- bits[1:0] of a value loaded to r15 must not have the value 0b10
- if bit[0] of a value loaded to r15 is set, the processor changes to Thumb state.

You cannot use the T suffix when loading to r15.

## Architectures

These ARM instructions are available in all versions of the ARM architecture.

These 32-bit Thumb-2 instructions are available in all T2 variants of the ARM architecture.

These 16-bit Thumb instructions are available in all T variants of the ARM architecture.

In T and T2 variants of ARMv5 and above:

- in ARM state, a load to r15 causes a change to Thumb state if bit[0] of the value loaded is 1
- in Thumb state, a load to r15 causes a change to ARM state if bit[0] of the value loaded is 0.

## 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.

STR r2,[r9,#consta-struct] ; consta-struct is an expression evaluating
 ; to a constant in the range 0-4095.

STR r5,[r7],#-8 ; stores a word from r5 to the address
 ; in r7, and then decrements r7 by 8.

LDR r0,localdata ; loads a word located at label localdata
```

### 4.2.3 LDR and STR (post-indexed immediate offset)

Load and Store register. Byte and halfword loads are zero-extended or sign-extended to 32 bits.

#### Syntax

*op*{*type*}{*T*}{*cond*} *Rd*, {*Rd2*}, [*Rn*], #*offset*

where:

*op* can be any one of:

|     |                 |
|-----|-----------------|
| LDR | Load Register   |
| STR | Store Register. |

*type* can be any one of:

|    |                            |
|----|----------------------------|
| B  | unsigned Byte              |
| SB | Signed Byte (LDR only)     |
| H  | unsigned Halfword          |
| SH | Signed Halfword (LDR only) |
| -  | omitted, for Word          |
| D  | Doubleword.                |

*T* is an optional suffix. If *T* is present, the memory system treats the access as though the processor was in User mode, even if it is in a privileged mode (see *Processor mode* on page 2-4). *T* has no effect in User mode.

*cond* is an optional condition code (see *Conditional execution* on page 2-17).

*Rd* is the ARM register to load or save.

*Rd2* is the second ARM register to load or save (*type* == D only).

*Rn* is the register on which the memory address is based. *Rn* must not be the same register as *Rd* or *Rd2*.

*offset* is an immediate offset. If offset is omitted, the instruction is a zero offset instruction.

## Operation and restrictions

The value in  $Rn$  is used as the memory address for the transfer. The offset is applied to the value in  $Rn$  after the data transfer takes place. The result is written back into  $Rn$ .

The range of offsets allowed is:

- $-4095$  to  $+4095$  for ARM Word or Byte instructions.
- $-255$  to  $+255$  for ARM Signed Byte, Halfword, Signed Halfword, and Doubleword instructions.
- $-255$  to  $+255$  for all Thumb-2 instructions except Doubleword instructions.
- $-1020$  to  $+1020$  for Thumb-2 Doubleword instructions. Must be a multiple of 4.

The T suffix is not available for Thumb-2 instructions, or for ARM doubleword instructions. It is available for all other ARM instructions.

## Doubleword register restrictions

For Thumb-2 instructions, you must not specify r15 for either  $Rd$  or  $Rd2$ .

For ARM instructions:

- $Rd$  must be an even-numbered register
- $Rd$  must not be r14
- $Rd2$  must be  $R(d + 1)$ .

## Loading to r15

$Rd$  can be the PC, in either ARM or Thumb-2 code. In this case, *type* must be omitted.

A load to r15 (pc) causes a branch to the instruction at the address loaded.

In ARMv4, bits[1:0] of the value loaded must be zero.

In ARMv5 and above:

- bits[1:0] of a value loaded to r15 must not have the value 0b10
- if bit[0] of a value loaded to r15 is set, the processor changes to Thumb state.

You cannot use the T suffix when loading to r15.

## Saving from r15

In Thumb code, you cannot save from r15.

In ARM code, avoid saving from r15 if possible.

If you do save from r15, the value saved is the address of the current instruction, plus an implementation-defined constant. The constant is always the same for a particular processor.

If your assembled code might be used on different processors, you can find out what the constant is at runtime using code like the following:

```
SUB R1, PC, #4 ; R1 = address of following STR instruction
STR PC, [R0] ; Store address of STR instruction + offset,
LDR R0, [R0] ; then reload it
SUB R0, R0, R1 ; Calculate the offset as the difference
```

If your code is to be assembled for a particular processor, the value of the constant is available in `armasm` as `{PCSTOREOFFSET}`.

## Architectures

These ARM instructions are available in all versions of the ARM architecture.

These 32-bit Thumb-2 instructions are available in all T2 variants of the ARM architecture.

There are no 16-bit versions of these instructions.

The `LDRSHT`, `LDRHT`, `STRHT`, and `LDRSBT` instructions are only available in T2 variants of ARMv6.

In T and T2 variants of ARMv5 and above:

- in ARM state, a load to r15 causes a change to Thumb state if bit[0] of the value loaded is 1
- in Thumb state, a load to r15 causes a change to ARM state if bit[0] of the value loaded is 0.

#### 4.2.4 LDR and STR (register or pre-indexed register offset)

Load and Store register. Byte and halfword loads are zero-extended or sign-extended to 32 bits.

##### Syntax

`op{type}{cond} {Rd, {Rd2,}} [Rn, +/-Rm {, shift}]{!}`

where:

*op* can be either:

|     |                 |
|-----|-----------------|
| LDR | Load Register   |
| STR | Store Register. |

*type* can be any one of:

|    |                            |
|----|----------------------------|
| B  | unsigned Byte              |
| SB | Signed Byte (LDR only)     |
| H  | unsigned Halfword          |
| SH | Signed Halfword (LDR only) |
| -  | omitted, for Word          |
| D  | Doubleword.                |

*cond* is an optional condition code (see *Conditional execution* on page 2-17).

*Rd* is the ARM register to load or save.

*Rd2* is the second ARM register to load or save (*type* == D only).

*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* must not be r15.

*shift* is an optional shift. See *Operation and restrictions* for details.

! is an optional suffix. If ! is present, the instruction is a pre-indexed instruction, and *Rn* must not be the same register as *Rd* or *Rd2*.

##### Operation and restrictions

In ARM, the value in *Rm* is added to or subtracted from the value in *Rn*. In Thumb and Thumb-2, subtraction is not allowed. The result is used as the memory address for the transfer. If the instruction is a pre-indexed instruction, the result is written back into *Rn*.

The range of shifts allowed is:

- LSL 0 to 3 for all Thumb-2 instructions except Doubleword
- Any one of the following for ARM Word and unsigned Byte instructions:
  - LSL 0 to 31
  - LSR 1 to 32
  - ASR 1 to 32
  - ROR 1 to 31
  - RRX
- No shift is allowed for Thumb-2 Doubleword instructions, or for ARM Signed Byte, unsigned Halfword, Signed halfword, or Doubleword instructions.

### Doubleword register restrictions

For Thumb-2 instructions, you must not specify r15 for either *Rd* or *Rd2*.

For ARM instructions:

- *Rd* must be an even-numbered register
- *Rd* must not be r14
- *Rd2* must be  $R(d + 1)$ .

### 16-bit instructions

16-bit versions of a subset of these instructions are available in Thumb-2 code, and in Thumb code on other Thumb-capable processors.

The following restrictions apply to 16-bit instructions:

- Only register offset instructions are available. Pre-indexed register offset instructions are not available.
- *Rd*, *Rn*, and *Rm* must all be Lo registers.
- Word, unsigned Halfword, Signed Halfword, unsigned Byte and Signed Byte instructions are available. Doubleword instructions are not available.

### Loading to r15

*Rd* can be the PC, in either ARM or Thumb-2 code. In this case, *type* must be omitted.

A load to r15 (pc) causes a branch to the instruction at the address loaded.

In ARMv4, bits[1:0] of the value loaded must be zero.

In ARMv5 and above:

- bits[1:0] of a value loaded to r15 must not have the value 0b10
- if bit[0] of a value loaded to r15 is set, the processor changes to Thumb state.

You cannot use the T suffix when loading to r15.

### **Saving from r15**

In Thumb code, you cannot save from r15.

In ARM code, avoid saving from r15 if possible.

If you do save from r15, the value saved is the address of the current instruction, plus an implementation-defined constant. The constant is always the same for a particular processor.

If your assembled code might be used on different processors, you can find out what the constant is at runtime using code like the following:

```
SUB R1, PC, #4 ; R1 = address of following STR instruction
STR PC, [R0] ; Store address of STR instruction + offset,
LDR R0, [R0] ; then reload it
SUB R0, R0, R1 ; Calculate the offset as the difference
```

If your code is to be assembled for a particular processor, the value of the constant is available in `armasm` as `{PCSTOREOFFSET}`.

### **Architectures**

The ARM instructions are available in all versions of the ARM architecture.

The 32-bit Thumb-2 instructions are available in all T2 variants of the ARM architecture.

The 16-bit Thumb instructions are available in all T variants of the ARM architecture.

In T and T2 variants of ARMv5 and above:

- in ARM state, a load to r15 causes a change to Thumb state if bit[0] of the value loaded is 1
- in Thumb state, a load to r15 causes a change to ARM state if bit[0] of the value loaded is 0.



#### 4.2.5 LDR and STR (post-indexed register offset)

Load and Store. Byte and halfword loads are zero-extended or sign-extended to 32 bits.

##### Syntax

*op*{*type*}{*T*}{*cond*} *Rd*, {*Rd2*,} [*Rn*], +/-*Rm* {, *shift*}

where:

*op* can be any one of:

|     |                 |
|-----|-----------------|
| LDR | Load Register   |
| STR | Store Register. |

*type* can be any one of:

|    |                            |
|----|----------------------------|
| B  | unsigned Byte              |
| SB | Signed Byte (LDR only)     |
| H  | unsigned Halfword          |
| SH | Signed Halfword (LDR only) |
| -  | omitted, for Word          |
| D  | Doubleword.                |

*cond* is an optional condition code (see *Conditional execution* on page 2-17).

*T* is an optional suffix. If *T* is present, the memory system treats the access as though the processor was in User mode, even if it is in a privileged mode (see *Processor mode* on page 2-4). *T* has no effect in User mode.

*Rd* is the ARM register to load or save.

*Rd2* is the second ARM register to load or save (*type* == D only).

*Rn* is the register on which the memory address is based. *Rn* must not be the same register as *Rd* or *Rd2*.

*Rm* is a register containing a value to be used as the offset. *Rm* must not be r15.

*shift* is an optional shift. See *Operation and restrictions* for details.

##### Operation and restrictions

The value in *Rn* is used as the memory address for the transfer. The offset is applied to the value in *Rn* after the data transfer takes place. In ARM, the offset can be added to or subtracted from *Rn*. In Thumb-2, the offset can only be added to *Rn*. The result is written back into *Rn*. *Rn* must not be r15.

The range of shifts allowed is:

- Any one of the following for ARM Word and Unsigned Byte instructions:
  - LSL 0 to 31
  - LSR 1 to 32
  - ASR 1 to 32
  - ROR 1 to 31
  - RRX
- No shift is allowed for Thumb-2 instructions, or for ARM Signed Byte, unsigned Halfword, Signed halfword, or Doubleword instructions.

The T suffix is not available for Thumb-2 instructions, or for ARM doubleword instructions. It is available for all other ARM instructions.

### Doubleword register restrictions

For Thumb-2 instructions, you must not specify r15 for either *Rd* or *Rd2*.

For ARM instructions:

- *Rd* must be an even-numbered register
- *Rd* must not be r14
- *Rd2* must be  $R(d + 1)$ .

### Loading to r15

*Rd* can be the PC, in either ARM or Thumb-2 code. In this case, *type* must be omitted.

A load to r15 (pc) causes a branch to the instruction at the address loaded.

In ARMv4, bits[1:0] of the value loaded must be zero.

In ARMv5 and above:

- bits[1:0] of a value loaded to r15 must not have the value 0b10
- if bit[0] of a value loaded to r15 is set, the processor changes to Thumb state.

You cannot use the T suffix when loading to r15.

### Saving from r15

In Thumb code, you cannot save from r15.

In ARM code, avoid saving from r15 if possible.

If you do save from r15, the value saved is the address of the current instruction, plus an implementation-defined constant. The constant is always the same for a particular processor.

If your assembled code might be used on different processors, you can find out what the constant is at runtime using code like the following:

```
SUB R1, PC, #4 ; R1 = address of following STR instruction
STR PC, [R0] ; Store address of STR instruction + offset,
LDR R0, [R0] ; then reload it
SUB R0, R0, R1 ; Calculate the offset as the difference
```

If your code is to be assembled for a particular processor, the value of the constant is available in `armasm` as `{PCSTOREOFFSET}`.

## Architectures

These ARM instructions are available in all versions of the ARM architecture.

These 32-bit Thumb-2 instructions are available in all T2 variants of the ARM architecture.

There are no 16-bit versions of these instructions.

In T and T2 variants of ARMv5 and above:

- in ARM state, a load to r15 causes a change to Thumb state if bit[0] of the value loaded is 1
- in Thumb state, a load to r15 causes a change to ARM state if bit[0] of the value loaded is 0.

### 4.2.6 LDR (PC-relative)

Load register. The address is an offset from the PC. Byte and halfword loads are zero-extended or sign-extended to 32 bits.

#### Syntax

`LDR{type}{cond}{.w} Rd, {Rd2}, label`

where:

|              |                                                                                                                                                                                                                |
|--------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>type</i>  | can be any one of:                                                                                                                                                                                             |
| B            | unsigned Byte                                                                                                                                                                                                  |
| SB           | Signed Byte (LDR only)                                                                                                                                                                                         |
| H            | unsigned Halfword                                                                                                                                                                                              |
| SH           | Signed Halfword (LDR only)                                                                                                                                                                                     |
| -            | omitted, for Word                                                                                                                                                                                              |
| D            | Doubleword.                                                                                                                                                                                                    |
| <i>cond</i>  | is an optional condition code (see <i>Conditional execution</i> on page 2-17).                                                                                                                                 |
| <i>.w</i>    | is an optional instruction width specifier. See <i>LDR (PC-relative)</i> in <i>Thumb-2</i> for details.                                                                                                        |
| <i>Rd</i>    | is the ARM register to load or save.                                                                                                                                                                           |
| <i>Rd2</i>   | is the second ARM register to load or save ( <i>type</i> == D only).                                                                                                                                           |
| <i>label</i> | is a program-relative expression. See <i>Register-relative and program-relative expressions</i> on page 3-28 for more information.<br><i>label</i> must be within $\pm 4\text{KB}$ of the current instruction. |

#### Usage

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

#### LDR (PC-relative) in Thumb-2

You can use the *.w* width specifier to force LDR to generate a 32-bit instruction in Thumb-2 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-2 LDR instruction.

### Doubleword register restrictions

For Thumb-2 instructions, you must not specify r15 for either *Rd* or *Rd2*.

For ARM instructions:

- *Rd* must be an even-numbered register
- *Rd* must not be r14
- *Rd2* must be  $R(d + 1)$ .

### 16-bit instruction

A 16-bit version of this instruction is available in Thumb-2 code, and in Thumb code on Thumb-capable processors conforming to ARMv6 and earlier.

The following restrictions apply to the 16-bit instruction:

- *Rd* must be a Lo register
- *type* must be omitted, that is, only Load Word is available
- offset must be in the range 0 to +1020, and must be divisible by 4.

### Loading to r15

*Rd* can be the PC, in either ARM or Thumb-2 code. In this case, *type* must be omitted.

A load to r15 (pc) causes a branch to the instruction at the address loaded.

In ARMv4, bits[1:0] of the value loaded must be zero.

In ARMv5 and above:

- bits[1:0] of a value loaded to r15 must not have the value 0b10
- if bit[0] of a value loaded to r15 is set, the processor changes to Thumb state.

You cannot use the T suffix when loading to r15.

### Architectures

This ARM instruction is available in all versions of the ARM architecture.

This 32-bit Thumb-2 instruction is available in all T2 variants of the ARM architecture.

This 16-bit Thumb instruction is available in all T variants of the ARM architecture.

In T and T2 variants of ARMv5 and above:

- in ARM state, a load to r15 causes a change to Thumb state if bit[0] of the value loaded is 1
- in Thumb state, a load to r15 causes a change to ARM state if bit[0] of the value loaded is 0.

## 4.2.7 PLD

Preload. The processor can signal the memory system that a load from an address is likely in the near future.

The address can be an immediate offset from the PC, or an immediate offset, register offset, or shifted register offset, from any register.

Implementation of PLD is optional. If it is not implemented, it executes as a NOP.

### Syntax

There are three forms of the PLD instruction:

- immediate or zero offset
- register offset or shifted register offset
- PC-relative.

The syntax of the three forms is as follows, in the same order:

PLD{*cond*} [*Rn* {, #*offset*}]

PLD{*cond*} [*Rn*, +/-*Rm* {, *shift*}]

PLD{*cond*} *label*

where:

*cond* is an optional condition code (see *Conditional execution* on page 2-17).

#### ———— **Note** ————

This is an unconditional instruction in ARM. *cond* is only allowed in Thumb-2 code, using a preceding IT instruction.

*Rn* is the register on which the memory address is based.

*offset* is an immediate offset. If offset is omitted, the instruction is a zero offset instruction.

*Rm* is a register containing a value to be used as the offset. *Rm* must not be r15.

*shift* is an optional shift.

*label* is a program-relative expression. See *Register-relative and program-relative expressions* on page 3-28 for more information.

**Immediate offset**

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 allowed is:

- –4095 to +4095 for ARM instructions
- –255 to +4095 for Thumb-2 instructions.

**Zero offset**

The value in *Rn* is used as the address for the preload.

**Register or shifted register offset**

In ARM, the value in *Rm* is added to or subtracted from the value in *Rn*. In Thumb-2, the value in *Rm* can only be added to the value in *Rn*. The result used as the memory address for the preload.

The range of shifts allowed is:

- LSL 0 to 3 for Thumb-2 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

**PC-relative**

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

**Address alignment for preloads**

No alignment checking is performed for preload instructions.

**Architectures**

This ARM instruction is available in ARMv5TE and above.

This 32-bit Thumb-2 instruction is available in all T2 variants of the ARM architecture.

There is no 16-bit Thumb PLD instruction.



## 4.2.8 LDM and STM

Load and Store Multiple registers. Any combination of registers r0 to r15 can be transferred in ARM state, but there are some restrictions in Thumb state.

See also *PUSH and POP* on page 4-30.

### Syntax

*op*{*addr\_mode*}{*cond*} *Rn*{!}, *reglist*{^}

where:

|                  |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
|------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>op</i>        | can be either:                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| LDM              | Load Multiple registers                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
| STM              | Store Multiple registers.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| <i>addr_mode</i> | 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.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
| <i>cond</i>      | is an optional condition code (see <i>Conditional execution</i> on page 2-17).                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| <i>Rn</i>        | is the <i>base register</i> , the ARM register holding the initial address for the transfer. <i>Rn</i> must not be r15.                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
| !                | is an optional suffix. If ! is present, the final address is written back into <i>Rn</i> .                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| <i>reglist</i>   | is a list of one or more registers to be loaded or stored, enclosed in braces. It can contain register ranges. It must be comma separated if it contains more than one register or register range (see <i>Examples</i> on page 4-29).<br>See <i>Restrictions on reglist in 32-bit Thumb-2 instructions</i> on page 4-28.                                                                                                                                                                                                                                                  |
| ^                | 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: <ul style="list-style-type: none"> <li>• If the instruction is LDM (or LDMIA) and <i>reglist</i> 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.</li> <li>• Otherwise, data is transferred into or out of the User mode registers instead of the current mode registers.</li> </ul> |

## Restrictions on reglist in 32-bit Thumb-2 instructions

In 32-bit Thumb-2 instructions:

- the SP cannot be in the list
- the PC cannot be in the list in an STM instruction
- the PC and LR cannot both be in the list in an LDM instruction.

## 16-bit instructions

16-bit versions of a subset of these instructions are available in Thumb-2 code, and in Thumb code on other Thumb-capable processors.

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.

In addition, the PUSH and POP instructions can be expressed in this form. Some forms of PUSH and POP are also 16-bit instructions. See *PUSH and POP* on page 4-30 for details.

## Loading to r15

A load to r15 (pc) causes a branch to the instruction at the address loaded.

## Loading or storing the base register, with writeback

If *Rn* is in *reglist*, and writeback is specified with the ! suffix:

- if the instruction is STM or STMIA and *Rn* is the lowest-numbered register in *reglist*, the initial value of *Rn* is stored
- otherwise, the loaded or stored value of *Rn* is unpredictable, that is, it cannot be relied upon.

## Architectures

These ARM instructions are available in all versions of the ARM architecture.

These 32-bit Thumb-2 instructions are available in T2 variants of ARMv6 and above.

These 16-bit Thumb instructions are available in all T variants of the ARM architecture.

In T variants of ARMv5 and above, a load to r15 also causes:

- in ARM state, a change to Thumb state, if bit[0] of the value loaded is 1
- in Thumb state, a change to ARM state, if bit[0] of the value loaded is 0.

The state change behavior can be disabled by setting the L4 bit (bit[15]) in cp15. See *ARM Architecture Reference Manual* for more details.

## Examples

```
LDM r8,{r0,r2,r9} ; LDMIA is a synonym for LDM
STMDB r1!,{r3-r6,r11,r12}
```

## Incorrect examples

```
STM r5!,{r5,r4,r9} ; value stored for r5 unpredictable
LDMDA r2, {} ; must be at least one register in list
```

### 4.2.9 PUSH and POP

Push registers onto the stack.

Pop registers off the stack.

#### Syntax

`PUSH{cond} reglist`

`POP{cond} reglist`

where:

*cond* is an optional condition code (see *Conditional execution* on page 2-17).

*reglist* is a list of registers or register ranges, enclosed in braces. It can contain register ranges. It must be comma separated if it contains more than one register or register range.

#### Usage

PUSH and POP are synonyms for STMDB and LDM (or LDMIA), with the base register r13 (sp), and the adjusted address written back to the base register. PUSH and POP are the preferred mnemonic in these cases.

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 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:

- if bits[1:0] of the value loaded to pc are b00, the processor changes to ARM state
- bits[1:0] must not have the value b10.

In ARMv4T and earlier, bits[1:0] of the value loaded to pc are ignored, so POP cannot be used to change state.

## 16-bit instructions

16-bit versions of a subset of these instructions are available in Thumb-2 code, and in Thumb code on other Thumb-capable processors.

The following restriction applies to the 16-bit instructions:

- all registers in *reglist* must be Lo registers, except that PUSH can include the LR, and POP can include the PC.

## Architectures

These ARM instructions are available in all versions of the ARM architecture.

These 32-bit Thumb-2 instructions are available in T2 variants of ARMv6 and above.

These 16-bit Thumb instructions are available in all T variants of the ARM architecture.

In T variants of ARMv5 and above, a load to r15 also causes:

- in ARM state, a change to Thumb state, if bit[0] of the value loaded is 1
- in Thumb state, a change to ARM state, if bit[0] of the value loaded is 0.

The state change behavior can be disabled by setting the L4 bit (bit[15]) in cp15. See *ARM Architecture Reference Manual* for more details.

## 16-bit examples

```
PUSH {r0,r3,r5}
PUSH {r1,r4-r7} ; pushes r1, r4, r5, r6, and r7
PUSH {r0,LR}
POP {r2,r5}
POP {r0-r7,pc} ; pop and return from subroutine
```

## ARM and 32-bit Thumb-2 examples

```
PUSH {r0,r3,r5}
PUSH {r1,r4-r11}
PUSH {r0,LR}
POP {r8,r12}
POP {r0-r10,pc}
```

## Incorrect example

```
PUSH {} ; must be at least one register in list
```

#### 4.2.10 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.                       |

*cond* is an optional condition code (see *Conditional execution* on page 2-17).

##### ———— Note ————

This is an unconditional instruction in ARM. *cond* is only allowed in Thumb-2 code, using a preceding IT instruction.

*Rn* specifies the base register. Do not use r15 for *Rn*.

! 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 (see *SRS* on page 4-34).

##### Operation

Loads the PC and the CPSR from the address contained in *Rn*, and the following address. Optionally updates *Rn*.

##### Exceptions

Data Abort.

## 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-2, 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.

The results of breaking these rules are unpredictable, that is, the results cannot be relied upon. However, no special precautions are required in software, if the instructions are used 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.

If User mode is specified by *mode*, the usual rules apply about writing to CPSR (see *ARM Architecture Reference Manual* for details).

If Monitor mode is specified by *mode*, the result is unpredictable, that is, it cannot be relied upon (see *SMI* on page 4-139).

## Architectures

This ARM instruction is available in ARMv6 and above.

This 32-bit Thumb-2 instruction is available in T2 variants of ARMv6 and above.

There is no 16-bit version of this instruction.

## Example

```
RFE r13!
```

## Incorrect example

```
RFEFD r15 ; do not use r15
```

## 4.2.11 SRS

Store Return State.

### Syntax

`SRS{addr_mode}{cond} r13{!}, #modenum`

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.           |

*cond* is an optional condition code (see *Conditional execution* on page 2-17).

#### ————— **Note** —————

This is an unconditional instruction in ARM. *cond* is only allowed in Thumb-2 code, using a preceding IT instruction.

! is an optional suffix. If ! is present, the final address is written back into the r13 of the mode specified by *modenum*.

*modenum* specifies the number of the mode whose banked r13 is used as the base register, see *Processor mode* on page 2-4.

### Operation

SRS stores the r14 and the SPSR of the current mode, at the address contained in r13 of the mode specified by *modenum*, and the following word respectively. Optionally updates r13 of the mode specified by *modenum*. This is compatible with the normal use of the STM instruction for stack accesses, see *LDM and STM* on page 4-27.

### Usage

You can use SRS to store return state for an exception handler on a different stack from the one automatically selected.

### Exceptions

Data Abort.



## 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.

SRS is unpredictable in User and System modes (that is, it cannot be relied upon), because they do not have SPSRs.

## Architectures

This ARM instruction is available in ARMv6 and above.

This 32-bit Thumb-2 instruction is available in T2 variants of ARMv6 and above.

There is no 16-bit version of this instruction.

## Example

```
R13_usr EQU 16
SRSFD #R13_usr
```

## Incorrect examples

```
SRSFD #32! ; there is no mode 32
SRSEQFD #R13_usr ; SRS is always unconditional
```

## 4.2.12 LDREX and STREX

Load and Store Register EXclusive.

### Syntax

LDREX{*size*}{*cond*} *Rd*, {*Rd2*,} [*Rn* {, #*offset*}]

STREX{*size*}{*cond*} *Rd*, *Rm*, {*Rm2*,} [*Rn* {, #*offset*}]

where:

|               |                                                                                                                                                                                                                                                                                                                                                     |   |                             |   |                                   |
|---------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---|-----------------------------|---|-----------------------------------|
| <i>size</i>   | is the size of the data to be loaded or stored. <i>size</i> must be one of:                                                                                                                                                                                                                                                                         |   |                             |   |                                   |
| B             | unsigned Byte                                                                                                                                                                                                                                                                                                                                       |   |                             |   |                                   |
| H             | unsigned Halfword                                                                                                                                                                                                                                                                                                                                   |   |                             |   |                                   |
| -             | omitted for Word                                                                                                                                                                                                                                                                                                                                    |   |                             |   |                                   |
| D             | Doubleword.                                                                                                                                                                                                                                                                                                                                         |   |                             |   |                                   |
| <i>cond</i>   | is an optional condition code (see <i>Conditional execution</i> on page 2-17).                                                                                                                                                                                                                                                                      |   |                             |   |                                   |
| <i>Rd</i>     | is the destination register. After the instruction, this contains: <ul style="list-style-type: none"> <li>• for LDREX, the data loaded from memory</li> <li>• for STREX, either:               <table> <tr> <td>0</td><td>if the instruction succeeds</td></tr> <tr> <td>1</td><td>if the instruction is locked out.</td></tr> </table> </li> </ul> | 0 | if the instruction succeeds | 1 | if the instruction is locked out. |
| 0             | if the instruction succeeds                                                                                                                                                                                                                                                                                                                         |   |                             |   |                                   |
| 1             | if the instruction is locked out.                                                                                                                                                                                                                                                                                                                   |   |                             |   |                                   |
| <i>Rd2</i>    | is the second destination register for doubleword loads.                                                                                                                                                                                                                                                                                            |   |                             |   |                                   |
| <i>Rn</i>     | is the register holding the memory address.                                                                                                                                                                                                                                                                                                         |   |                             |   |                                   |
| <i>Rm</i>     | is the source register holding the data to store to memory.                                                                                                                                                                                                                                                                                         |   |                             |   |                                   |
| <i>Rm2</i>    | is the second source register for doubleword stores.                                                                                                                                                                                                                                                                                                |   |                             |   |                                   |
| <i>offset</i> | is an offset applied to the value in <i>Rn</i> . <i>offset</i> is only allowed in Thumb-2 instructions, and only if <i>size</i> is omitted. The value of <i>offset</i> can be any multiple of four value in the range 0 to 1020. If <i>offset</i> is omitted, an offset of 0 is assumed.                                                            |   |                             |   |                                   |

### LDREX

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.

## STREX

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 and the tag is cleared.
- 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.
- 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 and the tag is cleared.
- 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.

## Restrictions

*r15* must not be used for any of *Rd*, *Rn*, *Rm*, *Rd2*, or *Rm2*.

*Rd* must not be the same register as any of *Rm*, or *Rm2*. For STREX, it must not be the same register as *Rn*.

*Rd* and *Rd2* must not be the same register.

In the ARM LDREXD instruction, *Rd* must be an even numbered register, and not *r14*.

In the ARM STREXD instruction, *Rm* must be an even numbered register, and not *r14*.

## 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 instruction 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. The result of executing a STREX instruction to a different address is unpredictable, that is, it cannot be relied upon.

---

**Differences between the ARM and Thumb-2 instructions**

In Thumb-2, the word LDREX and STREX instructions have an optional offset from the base register. None of the ARM load and store exclusive instructions can have offsets.

In Thumb-2, the doubleword LDREXD and STREXD instructions can use any two general-purpose registers for the data. The ARM instructions are restricted to using consecutive registers.

**Exceptions**

Data Abort.

**Architectures**

These ARM instructions, without *size*, are available in ARMv6 and above.

These ARM instructions, with *size*, are available in ARMv6T2 and above.

These 32-bit Thumb-2 instructions are available in T2 variants of ARMv6 and above.

**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

```

### 4.2.13 CLREX

CLear EXclusive. Clears the local record of the executing processor that an address has had a request for an exclusive access.

#### Syntax

CLREX{*cond*}

where:

*cond* is an optional condition code (see *Conditional execution* on page 2-17).

#### ———— Note ————

This is an unconditional instruction in ARM. *cond* is only allowed in Thumb-2 code, using a preceding IT instruction.

#### Usage

Use the CLREX instruction to return a closely-coupled exclusive access monitor to its open-access state. This removes the requirement for a dummy store to memory. See *ARM Architecture reference Manual* for more information on synchronization primitive support.

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 ARMv6T2 and above.

This 32-bit Thumb-2 instruction is available in T2 variants of ARMv6 and above.

There is no 16-bit Thumb CLREX instruction.

#### 4.2.14 SWP and SWPB

Swap data between registers and memory.

You can use SWP and SWPB to implement semaphores. The use of SWP and SWPB is deprecated in ARMv6 and above. See *LDREX and STREX* on page 4-36 for instructions to implement more sophisticated semaphores in ARMv6 and above.

##### Syntax

SWP{B}{*cond*} *Rd*, *Rm*, [*Rn*]

where:

|             |                                                                                                                                                                                                                        |
|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>cond</i> | is an optional condition code (see <i>Conditional execution</i> on page 2-17).                                                                                                                                         |
| B           | is an optional suffix. If B is present, a byte is swapped. Otherwise, a 32-bit word is swapped.                                                                                                                        |
| <i>Rd</i>   | is an ARM register. Data from memory is loaded into <i>Rd</i> .                                                                                                                                                        |
| <i>Rm</i>   | is an ARM register. The contents of <i>Rm</i> is saved to memory.<br><i>Rm</i> can be the same register as <i>Rd</i> . In this case, the contents of the register is swapped with the contents of the memory location. |
| <i>Rn</i>   | is an ARM register. The contents of <i>Rn</i> specify the address in memory with which data is to be swapped. <i>Rn</i> must be a different register from both <i>Rd</i> and <i>Rm</i> .                               |

##### Architectures

These ARM instructions are available in all versions of the ARM architecture.

There are no Thumb or Thumb-2 SWP or SWPB instructions.

## 4.3 General data processing instructions

This section contains the following subsections:

- *Flexible second operand* on page 4-42
- *ADD, SUB, RSB, ADC, SBC, and RSC* on page 4-45  
Add, Subtract, and Reverse Subtract, each with or without Carry.
- *SUBS PC, LR (Thumb-2 only)* on page 4-49  
Return from exception without popping the stack.
- *AND, ORR, EOR, BIC, and ORN* on page 4-50  
Logical AND, OR, Exclusive OR, OR NOT, and Bit Clear.
- *BFC and BFI* on page 4-53  
Bit Field Clear and Bit Field Insert.
- *CLZ* on page 4-54  
Count Leading Zeros.
- *CMP and CMN* on page 4-55  
Compare and Compare Negative.
- *MOV and MVN* on page 4-57  
Move, Copy, and Move Not.
- *MOVT* on page 4-60  
Move Top, Wide.
- *TST and TEQ* on page 4-61  
Test and Test Equivalence.
- *SEL* on page 4-63  
Select bytes from each operand according to the state of the CPSR GE flags.
- *REV, REV16, REVSH, and RBIT* on page 4-65  
Reverse bytes or Bits.
- *ASR, LSL, LSR, ROR, and RRX* on page 4-67  
Arithmetic Shift Right.
- *IT* on page 4-70  
If-Then.

### 4.3.1 Flexible second operand

Many ARM and Thumb-2 general data processing instructions have a flexible second operand. This is shown as *Operand2* in the descriptions of the syntax of each instruction. There are some differences in the options permitted for *Operand2* in ARM and Thumb-2 instructions.

#### Syntax

*Operand2* has two possible forms:

*#constant*

*Rm*{, *shift*}

where:

*constant* is an expression evaluating to a numeric constant. The range of constants available in ARM and Thumb-2 is not exactly the same. See *Constants in Operand2* on page 4-43 for details.

*Rm* is the ARM register holding the data for the second operand. The bit pattern in the register can be shifted or rotated in various ways.

*shift* is an optional shift to be applied to *Rm*. It can be any one of:

ASR *#n* arithmetic shift right *n* bits.  $1 \leq n \leq 32$ .

LSL *#n* logical shift left *n* bits.  $0 \leq n \leq 31$ .

LSR *#n* logical shift right *n* bits.  $1 \leq n \leq 32$ .

ROR *#n* rotate right *n* bits.  $1 \leq n \leq 31$ .

RRX rotate right one bit, with extend.

*type Rs* available in ARM only, where:

*type* is one of ASR, LSL, LSR, ROR.

*Rs* is an ARM register supplying the shift amount.  
Only the least significant byte is used.

---

#### Note

---

The result of the shift operation is used as *Operand2* in the instruction, but *Rm* itself is not altered.

---



## Constants in Operand2

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 32-bit Thumb-2 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`.

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

Constants produced by rotating an 8-bit value right by 2, 4, or 6 bits are available in ARM data processing instructions, but not in Thumb-2. All other ARM constants are also available in Thumb-2.

## ASR

Arithmetic shift right by  $n$  bits divides the value contained in  $Rm$  by  $2^n$ , if the contents are regarded as a two's complement signed integer. The original bit[31] is copied into the left-hand  $n$  bits of the register.

## LSR and LSL

Logical shift right by  $n$  bits divides the value contained in  $Rm$  by  $2^n$ , if the contents are regarded as an unsigned integer. The left-hand  $n$  bits of the register are set to 0.

Logical shift left by  $n$  bits multiplies the value contained in  $Rm$  by  $2^n$ , if the contents are regarded as an unsigned integer. Overflow can occur without warning. The right-hand  $n$  bits of the register are set to 0.

## ROR

Rotate right by  $n$  bits moves the right-hand  $n$  bits of the register into the left-hand  $n$  bits of the result. At the same time, all other bits are moved right by  $n$  bits (see Figure 4-1 on page 4-44).

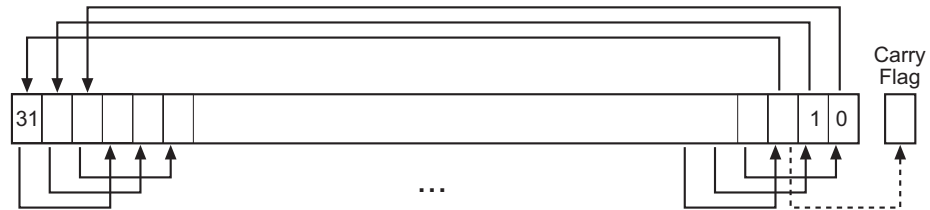


Figure 4-1 ROR

## RRX

Rotate right with extend shifts the contents of *Rm* right by one bit. The carry flag is copied into bit[31] of *Rm* (see Figure 4-2).

The old value of bit[0] of *Rm* is shifted out to the carry flag if the S suffix is specified (see *The carry flag*).

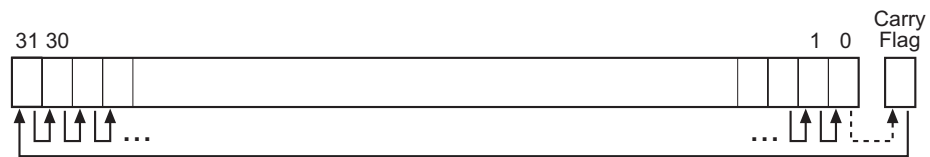


Figure 4-2 RRX

## The carry flag

The carry flag is updated to the last bit shifted out of *Rm*, if the instruction is any one of the following:

- MOV, MVN, AND, ORR, ORN, EOR or BIC, if you use the S suffix
- TEQ or TST, for which no S suffix is required.

## Instruction substitution

Certain pairs of instructions (ADD and SUB, ADC and SBC, AND and BIC, MOV and MVN, CMP and CMN) are equivalent except for the negation or logical inversion of *constant*.

If a value of *constant* is not available, but its logical inverse or negation is, the assembler substitutes the other instruction of the pair and inverts or negates *constant*.

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

### 4.3.2 ADD, SUB, RSB, ADC, SBC, and RSC

Add, Subtract, and Reverse Subtract, each with or without Carry.

See also *Parallel add and subtract* on page 4-101.

#### Syntax

*op*{*S*}{*cond*} *Rd*, *Rn*, *Operand2*

where:

|                 |                                                                                                                                                                       |
|-----------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>op</i>       | is one of:                                                                                                                                                            |
| ADD             | Add.                                                                                                                                                                  |
| ADC             | Add with Carry.                                                                                                                                                       |
| SUB             | Subtract.                                                                                                                                                             |
| RSB             | Reverse Subtract.                                                                                                                                                     |
| SBC             | Subtract with Carry.                                                                                                                                                  |
| RSC             | Reverse Subtract with Carry (ARM only).                                                                                                                               |
| <i>S</i>        | is an optional suffix. If <i>S</i> is specified, the condition code flags are updated on the result of the operation (see <i>Conditional execution</i> on page 2-17). |
| <i>cond</i>     | is an optional condition code (see <i>Conditional execution</i> on page 2-17).                                                                                        |
| <i>Rd</i>       | is the destination register.                                                                                                                                          |
| <i>Rn</i>       | is the register holding the first operand.                                                                                                                            |
| <i>Operand2</i> | is a flexible second operand. See <i>Flexible second operand</i> on page 4-42 for details of the option. See also <i>Wide constants</i> on page 4-46.                 |

#### Usage

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

The SUB instruction subtracts the value of *Operand2* from the value in *Rn*.

The RSB (Reverse Subtract) instruction subtracts the value in *Rn* from the value of *Operand2*. This is useful because of the wide range of options for *Operand2*.

You can use ADC, SBC, and RSC to synthesize multiword arithmetic (see *Multiword arithmetic examples* on page 4-48).

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

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.

The RSC (Reverse Subtract with Carry) instruction subtracts the value in *Rn* from the value of *Operand2*. If the carry flag is clear, the result is reduced by one.

In certain circumstances, the assembler can substitute one instruction for another. Be aware of this when reading disassembly listings. See *Instruction substitution* on page 4-44 for details.

### Wide constants

In Thumb-2 ADD and SUB instructions, *Operand2* can take any value in the range 0-4095, in addition to the normal range of *Operand2* values. These wide constants cannot be used with RSB, ADC, SBC or RSC instructions.

You cannot use the S suffix with wide constants.

### Use of r15 in Thumb-2 instructions

In most of these instructions, you cannot use r15 for *Rd*, or any operand.

The exception is that you can use r15 for *Rn* in ADD and 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.

See also *SUBS PC, LR (Thumb-2 only)* on page 4-49.

### Use of r15 in ARM instructions

If you use r15 as *Rn*, the value used is the address of the instruction plus 8.

If you use r15 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 (see the *Handling Processor Exceptions* chapter in *RealView Compilation Tools v2.2 Developer Guide*).

#### Caution

Do not use the S suffix when using r15 as *Rd* in User mode or System mode. The effect of such an instruction is unpredictable (that is, it cannot be relied upon), but the assembler cannot warn you at assembly time.

You cannot use r15 for *Rd* or any operand in any data processing instruction that has a register-controlled shift (see *Flexible second operand* on page 4-42).

### 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 when used in Thumb-2 code:

|                                           |                                                                                                                                           |
|-------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------|
| ADDS <i>Rd</i> , <i>Rn</i> , # <i>imm</i> | <i>imm</i> range 0-7. <i>Rd</i> and <i>Rn</i> must both be Lo registers.                                                                  |
| ADDS <i>Rd</i> , <i>Rn</i> , <i>Rm</i>    | <i>Rd</i> , <i>Rn</i> and <i>Rm</i> must all be Lo registers.                                                                             |
| ADD <i>Rd</i> , <i>Rd</i> , <i>Rm</i>     | ARMv6 and earlier: either <i>Rd</i> or <i>Rm</i> , or both, must be a Hi register.<br>ARMv6T2 and above: this restriction does not apply. |
| ADDS <i>Rd</i> , <i>Rd</i> , # <i>imm</i> | <i>imm</i> range 0-255. <i>Rd</i> must be a Lo register.                                                                                  |
| ADCS <i>Rd</i> , <i>Rd</i> , <i>Rm</i>    | <i>Rd</i> , <i>Rn</i> and <i>Rm</i> must all be Lo registers.                                                                             |
| ADD SP, SP, # <i>imm</i>                  | <i>imm</i> range 0-508, word aligned.                                                                                                     |
| ADD <i>Rd</i> , SP, # <i>imm</i>          | <i>imm</i> range 0-1020, word aligned. <i>Rd</i> must be a Lo register.                                                                   |
| ADD <i>Rd</i> , PC, # <i>imm</i>          | <i>imm</i> range 0-1020, word aligned. <i>Rd</i> must be a Lo register. Bits[1:0] of the PC are read as 0 in this instruction.            |
| SUBS <i>Rd</i> , <i>Rn</i> , <i>Rm</i>    | <i>Rd</i> , <i>Rn</i> and <i>Rm</i> must all be Lo registers.                                                                             |
| SUBS <i>Rd</i> , <i>Rn</i> , # <i>imm</i> | <i>imm</i> range 0-7. <i>Rd</i> and <i>Rn</i> both Lo registers.                                                                          |
| SUBS <i>Rd</i> , <i>Rd</i> , # <i>imm</i> | <i>imm</i> range 0-255. <i>Rd</i> must be a Lo register.                                                                                  |
| SBCS <i>Rd</i> , <i>Rd</i> , <i>Rm</i>    | <i>Rd</i> , <i>Rn</i> and <i>Rm</i> must all be Lo registers.                                                                             |
| SUB SP, SP, # <i>imm</i>                  | <i>imm</i> range 0-508, word aligned.                                                                                                     |
| RSBS <i>Rd</i> , <i>Rn</i> , #0           | <i>Rd</i> and <i>Rn</i> both Lo registers.                                                                                                |

### Architectures

These ARM instructions are available in all versions of the ARM architecture.

These 32-bit Thumb-2 instructions are available in T2 variants of ARMv6 and above.

These 16-bit Thumb instructions are available in all T variants of the ARM architecture.

**Examples**

```

ADD r2, r1, r3
SUBS r8, r6, #240 ; sets the flags on the result
RSB r4, r4, #1280 ; subtracts contents of r4 from 1280
ADCHI r11, r0, r3 ; only executed if C flag set and Z
 ; flag clear
RSCLES r0,r5,r0,LSL r4 ; conditional, flags set

```

**Incorrect example**

```

RSCLES r0,r15,r0,LSL r4 ; r15 not permitted with register
 ; controlled shift

```

**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

```

These instructions subtract one 96-bit integer from another:

```

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

```

### 4.3.3 SUBS PC, LR (Thumb-2 only)

Exception return, without stack popping.

#### Syntax

SUBS{*cond*} PC, LR, #*immed\_8*

where:

*immed\_8* is an 8-bit immediate constant.

*cond* is an optional condition code (see *Conditional execution* on page 2-17).

#### Usage

You can use SUBS PC, LR to return from an exception if there is no return state on the stack.

SUBS PC, LR subtracts a value from the link register and loads the PC with the result, then copies the SPSR to the CPSR.

#### Notes

SUBS PC, LR 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-2, 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.

The results of breaking these rules are unpredictable, that is, they cannot be relied upon. However, no special precautions are required in software, if the instructions are used to return after a valid exception entry mechanism.

MOVS pc, lr is a synonym of SUBS pc, lr, #0 in Thumb-2.

#### Architectures

This 32-bit Thumb-2 instruction is available in T2 variants of ARMv6 and above.

#### 4.3.4 AND, ORR, EOR, BIC, and ORN

Logical AND, OR, Exclusive OR, Bit Clear, and OR NOT.

##### Syntax

*op*{S}{*cond*} *Rd*, *Rn*, *Operand2*

where:

|                 |                                                                                                                                                                       |
|-----------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>op</i>       | is one of:                                                                                                                                                            |
| AND             | logical AND.                                                                                                                                                          |
| ORR             | logical OR.                                                                                                                                                           |
| EOR             | logical Exclusive OR.                                                                                                                                                 |
| BIC             | logical AND NOT.                                                                                                                                                      |
| ORN             | logical OR NOT (Thumb-2 only).                                                                                                                                        |
| <i>S</i>        | is an optional suffix. If <i>S</i> is specified, the condition code flags are updated on the result of the operation (see <i>Conditional execution</i> on page 2-17). |
| <i>cond</i>     | is an optional condition code (see <i>Conditional execution</i> on page 2-17).                                                                                        |
| <i>Rd</i>       | is the destination register.                                                                                                                                          |
| <i>Rn</i>       | is the register holding the first operand.                                                                                                                            |
| <i>Operand2</i> | is a flexible second operand. See <i>Flexible second operand</i> on page 4-42 for details of the options.                                                             |

##### Usage

The AND, EOR, and ORR instructions perform bitwise AND, Exclusive OR, and OR operations on the values in *Rn* and *Operand2*.

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*.

The ORN Thumb-2 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 BIC for AND, AND for BIC, ORN for ORR, or ORR for ORN. Be aware of this when reading disassembly listings. See *Instruction substitution* on page 4-44 for details.



## Use of r15 in Thumb-2 instructions

You cannot use r15 for *Rd* or any operand in any of these instructions.

## Use of r15 in ARM instructions

If you use r15 as *Rn*, the value used is the address of the instruction plus 8.

If you use r15 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 (see the *Handling Processor Exceptions* chapter in *RealView Compilation Tools v2.2 Developer Guide*).

### Caution

Do not use the S suffix when using r15 as *Rd* in User mode or System mode. The effect of such an instruction is unpredictable, that is, it cannot be relied upon, but the assembler cannot warn you at assembly time.

You cannot use r15 for *Rd* or any operand in any data processing instruction that has a register-controlled shift (see *Flexible second operand* on page 4-42).

## Condition flags

If S is specified, these instructions:

- update the N and Z flags according to the result
- can update the C flag during the calculation of *Operand2* (see *Flexible second operand* on page 4-42)
- do not affect the V flag.

## 16-bit instructions

The following forms of these instructions are available in Thumb code, and are 16-bit instructions when used in Thumb-2 code:

ANDS *Rd, Rd, Rm*      *Rd* and *Rm* must both be Lo registers.

EORS *Rd, Rd, Rm*      *Rd* and *Rm* must both be Lo registers.

ORRS *Rd, Rd, Rm*      *Rd* and *Rm* must both be Lo registers.

BICS *Rd*, *Rd*, *Rm*     *Rd* and *Rm* must both be Lo registers.

In the first three cases, it does not matter if you specify *OPS Rd, Rm, Rd*. The instruction is the same.

## Architectures

These ARM instructions are available in all versions of the ARM architecture.

These 32-bit Thumb-2 instructions are available in T2 variants of ARMv6 and above.

These 16-bit Thumb instructions are available in all T variants of the ARM architecture.

## ARM/Thumb-2 examples

```
AND r9, r2, #0xFF00
ORREQ r2, r0, r5
EORS r0, r0, r3, ROR r6
ANDS r9, r8, #0x19
EORS r7, r11, #0x18181818
BIC r0, r1, #0xab
ORN r7, r11, r14, ROR #4
ORNS r7, r11, r14, ASR #32
```

## Incorrect example

```
EORS r0, r15, r3, ROR r6 ; r15 not permitted with register
 ; controlled shift
```

### 4.3.5 BFC and BFI

Bit Field Clear and Bit Field Insert. Clear adjacent bits in a register, or Insert adjacent bits from one register into another.

#### Syntax

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

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

where:

|              |                                                                                                                                    |
|--------------|------------------------------------------------------------------------------------------------------------------------------------|
| <i>cond</i>  | is an optional condition code (see <i>Conditional execution</i> on page 2-17).                                                     |
| <i>Rd</i>    | is the destination register. <i>Rd</i> must not be r15.                                                                            |
| <i>Rn</i>    | is the source register. <i>Rn</i> must not be r15.                                                                                 |
| <i>lsb</i>   | is the least significant bit that is to be cleared or copied.                                                                      |
| <i>width</i> | is the number of bits to be cleared or copied. <i>width</i> must not be 0, and ( <i>width</i> + <i>lsb</i> ) must be less than 32. |

#### BFC

*width* bits in *Rd* are cleared, starting at *lsb*. Other bits in *Rd* are unchanged.

#### BFI

*width* bits in *Rd*, starting at *lsb*, are replaced by *width* bits from *Rn*, starting at bit[0]. Other bits in *Rd* are unchanged.

#### Condition flags

These instructions do not change the flags.

#### Architectures

These ARM instructions are available in ARMv6T2 and above.

These 32-bit Thumb-2 instructions are available in T2 variants of ARMv6 and above.

There are no 16-bit Thumb versions of these instructions.

### 4.3.6 CLZ

Count Leading Zeros.

#### Syntax

CLZ{*cond*} *Rd*, *Rm*

where:

*cond* is an optional condition code (see *Conditional execution* on page 2-17).

*Rd* is the destination register. *Rd* must not be r15.

*Rm* is the operand register. *Rm* must not be r15.

#### Usage

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.

#### Condition flags

This instruction does not change the flags.

#### Architectures

This ARM instruction is available in ARMv5 and above.

This 32-bit Thumb-2 instruction is available in T2 variants of ARMv6 and above.

There is no 16-bit Thumb version of this instruction.

#### Examples

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

Use the CLZ Thumb-2 instruction followed by a left shift of *Rm* by the resulting *Rd* value to normalize the value of register *Rm*. Use MOV<sub>S</sub>, rather than MOV, to flag the case where *Rm* is zero:

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

### 4.3.7 CMP and CMN

Compare and Compare Negative.

#### Syntax

`CMP{cond} Rn, Operand2`

`CMN{cond} Rn, Operand2`

where:

*cond* is an optional condition code (see *Conditional execution* on page 2-17).

*Rn* is the ARM register holding the first operand.

*Operand2* is a flexible second operand. See *Flexible second operand* on page 4-42 for details of the options.

#### Usage

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. See *Instruction substitution* on page 4-44 for details.

#### Use of r15 in ARM instructions

If you use r15 as *Rn*, the value used is the address of the instruction plus 8.

You cannot use r15 for any operand in any data processing instruction that has a register-controlled shift (see *Flexible second operand* on page 4-42).

#### Use of r15 in Thumb-2 instructions

You cannot use r15 for any operand in these instructions.

## 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 when used in Thumb-2 code:

`CMP Rn, Rm`            *Rn* and *Rm* can both be Lo or Hi registers.

`CMN Rn, Rm`            *Rn* and *Rm* must both be Lo registers.

`CMP Rn, #imm`           *Rn* must be a Lo register. *imm* range 0-255.

## Architectures

These ARM instructions are available in all versions of the ARM architecture.

These 32-bit Thumb-2 instructions are available in T2 variants of ARMv6 and above.

These 16-bit Thumb instructions are available in all T variants of the ARM architecture.

## Examples

```
CMP r2, r9
CMN r0, #6400
CMPGT r13, r7, LSL #2
```

## Incorrect example

```
CMP r2, r15, ASR r0 ; r15 not permitted with register controlled shift
```

### 4.3.8 MOV and MVN

Move and Move Not.

#### Syntax

MOV{S}{*cond*} *Rd*, *Operand2*

MVN{S}{*cond*} *Rd*, *Operand2*

where:

- S* is an optional suffix. If *S* is specified, the condition code flags are updated on the result of the operation (see *Conditional execution* on page 2-17).
- cond* is an optional condition code (see *Conditional execution* on page 2-17).
- Rd* is the destination register.
- Operand2* is a flexible second operand. See *Flexible second operand* on page 4-42 for details of the options. See also *Wide constants*.

#### Usage

The MOV instruction copies the value of *Operand2* into *Rd*.

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. See *Instruction substitution* on page 4-44 for details.

#### Wide constants

In MOV instructions, *Operand2* can take any value in the range 0-65535, in addition to the normal range of *Operand2* values. These wide constants cannot be used with MVN instructions.

You cannot use the *S* suffix with a wide constant.

You cannot use r15 as *Rd* with a wide constant.

## Use of r15

You cannot use r15 for *Rd*, or in *Operand2*, in the Thumb-2 MOV or MVN instructions. The remainder of this section applies to the ARM instructions.

If you use r15 as *Rd*, the value used is the address of the instruction plus 8.

If you use r15 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 (see the *Handling Processor Exceptions* chapter in *RealView Compilation Tools v2.2 Developer Guide*).

---

### Caution

---

Do not use the S suffix when using r15 as *Rd* in User mode or System mode. The effect of such an instruction is unpredictable, that is, it cannot be relied upon, but the assembler cannot warn you at assembly time.

---

You cannot use r15 for *Rd* or any operand in any data processing instruction that has a register-controlled shift (see *Flexible second operand* on page 4-42).

## Condition flags

If S is specified, these instructions:

- update the N and Z flags according to the result
- can update the C flag during the calculation of *Operand2* (see *Flexible second operand* on page 4-42)
- do not affect the V flag.

## 16-bit instructions

The following forms of these instructions are available in Thumb code, and are 16-bit instructions when used in Thumb-2 code:

|                             |                                                                                                                                            |
|-----------------------------|--------------------------------------------------------------------------------------------------------------------------------------------|
| MOVS <i>Rd</i> , <i>imm</i> | <i>Rd</i> must be a Lo register. <i>imm</i> range 0-255.                                                                                   |
| MOVS <i>Rd</i> , <i>Rm</i>  | <i>Rd</i> and <i>Rm</i> must both be Lo registers.                                                                                         |
| MOV <i>Rd</i> , <i>Rm</i>   | In ARMv5 and earlier, either <i>Rd</i> or <i>Rm</i> , or both, must be a Hi register. In ARMv6 and above, this restriction does not apply. |



## Architectures

These ARM instructions are available in all versions of the ARM architecture.

These 32-bit Thumb-2 instructions are available in T2 variants of ARMv6 and above.

These 16-bit Thumb instructions are available in all T variants of the ARM architecture.

## Example

```
MVNNE r11, #0xF000000B ; ARM only. This constant is not available in T2.
```

## Incorrect example

```
MVN r15,r3,ASR r0 ; r15 not permitted with register controlled shift
```

### 4.3.9 MOV<sub>T</sub>

Move Top. Writes a 16-bit immediate value to the top halfword of a register, without affecting the bottom halfword.

#### Syntax

`MOVT{cond} Rd, #immed_16`

where:

*cond* is an optional condition code (see *Conditional execution* on page 2-17).

*Rd* is the destination register. *Rd* cannot be the PC.

*immed\_16* is a 16-bit immediate constant.

#### Usage

MOV<sub>T</sub> writes *immed\_16* to *Rd*[31:16]. The write does not affect *Rd*[15:0].

You can generate any 32-bit constant with a MOV, MOV<sub>T</sub> instruction pair.

See also *MOV32 pseudo-instruction* on page 4-149.

#### Condition flags

This instruction does not change the flags.

#### Architectures

This ARM instruction is available in ARMv6T2 and above.

This 32-bit Thumb-2 instruction is available in T2 variants of ARMv6 and above.

There is no 16-bit Thumb version of this instruction.

### 4.3.10 TST and TEQ

Test bits and Test Equivalence.

#### Syntax

TST{*cond*} *Rn*, *Operand2*

TEQ{*cond*} *Rn*, *Operand2*

where:

*cond* is an optional condition code (see *Conditional execution* on page 2-17).

*Rn* is the ARM register holding the first operand.

*Operand2* is a flexible second operand. See *Flexible second operand* on page 4-42 for details of the options.

#### Usage

These instructions test the value in a register against *Operand2*. They update the condition flags on the result, but do 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.

The TEQ instruction performs a bitwise Exclusive OR operation on the value in *Rn* and the value of *Operand2*. This is the same as a 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.

#### Use of r15

You cannot use r15 for *Rn*, or in *Operand2*, in the Thumb-2 TST or TEQ instructions. The remainder of this section applies to the ARM instructions.

If you use r15 as *Rn*, the value used is the address of the instruction plus 8.

You cannot use r15 for any operand in any data processing instruction that has a register-controlled shift (see *Flexible second operand* on page 4-42).

## Condition flags

These instructions:

- update the N and Z flags according to the result
- can update the C flag during the calculation of *Operand2* (see *Flexible second operand* on page 4-42)
- do 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 when used in Thumb-2 code:

TST *Rn*, *Rm*                      *Rn* and *Rm* must both be Lo registers.

## Architectures

These ARM instructions are available in all versions of the ARM architecture.

These 32-bit Thumb-2 instructions are available in T2 variants of ARMv6 and above.

The 16-bit Thumb TST instruction is available in all T variants of the ARM architecture.

## Examples

```
TST r0, #0x3F8
TEQEQ r10, r9
TSTNE r1, r5, ASR r1
```

## Incorrect example

```
TEQ r15, r1, ROR r0 ; r15 not permitted with register
 ; controlled shift
```

### 4.3.11 SEL

Select bytes from each operand according to the state of the CPSR GE flags.

#### Syntax

SEL{*cond*} *Rd*, *Rn*, *Rm*

where:

- cond* is an optional condition code (see *Conditional execution* on page 2-17).
- 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 CPSR 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

Do not use r15 for *Rd*, *Rn*, or *Rm*.

Use the SEL instruction after one of the signed parallel instructions, see *Parallel add and subtract* on page 4-101. You can use this to select maximum or minimum values in multiple byte or halfword data.

#### Condition flags

This instruction does not change the flags.

#### Architectures

This ARM instruction is available in ARMv6 and above.

This 32-bit Thumb-2 instruction is available in T2 variants of ARMv6 and above.

There is no 16-bit Thumb version of this instruction.

## 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
```

### 4.3.12 REV, REV16, REVSH, and RBIT

Reverse bytes or bits within words or halfwords.

#### Syntax

*op{cond} Rd, Rn*

where:

|             |                                                                                |
|-------------|--------------------------------------------------------------------------------|
| <i>op</i>   | is any one of the following:                                                   |
| REV         | Reverse byte order in a word.                                                  |
| REV16       | Reverse byte order in each halfword independently.                             |
| REVSH       | Reverse byte order in the bottom halfword, and sign extend to 32 bits.         |
| RBIT        | Reverse the bit order in a 32-bit word.                                        |
| <i>cond</i> | is an optional condition code (see <i>Conditional execution</i> on page 2-17). |
| <i>Rd</i>   | is the destination register. <i>Rd</i> must not be r15.                        |
| <i>Rn</i>   | is the register holding the operand. <i>Rn</i> must not be r15.                |

#### Usage

You can use these instructions to change endianness:

|       |                                                                                                                                                                                                                   |
|-------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| REV   | converts 32-bit big-endian data into little-endian data or 32-bit little-endian data into big-endian data.                                                                                                        |
| REV16 | converts 16-bit big-endian data into little-endian data or 16-bit little-endian data into big-endian data.                                                                                                        |
| REVSH | converts either: <ul style="list-style-type: none"> <li>16-bit signed big-endian data into 32-bit signed little-endian data</li> <li>16-bit signed big-endian data into 32-bit signed big-endian data.</li> </ul> |

#### Condition flags

These instructions do not change the flags.

## 16-bit instructions

The following forms of these instructions are available in Thumb code, and are 16-bit instructions when used in Thumb-2 code:

REV *Rd*, *Rm*                      *Rd* and *Rm* must both be Lo registers.

REV16 *Rd*, *Rm*                      *Rd* and *Rm* must both be Lo registers.

REVSH *Rd*, *Rm*                      *Rd* and *Rm* must both be Lo registers.

## Architectures

Other than RBIT, these ARM instructions are available in ARMv6 and above.

The ARM RBIT instruction is available in ARMv6T2 and above.

These 32-bit Thumb-2 instructions are available in T2 variants of ARMv6 and above.

These 16-bit Thumb instructions are available in all T variants of ARMv6 and above.

## Examples

|       |        |                                         |
|-------|--------|-----------------------------------------|
| REV   | r3, r7 |                                         |
| REV16 | r0, r0 |                                         |
| REVSH | r0, r5 | ; Reverse Signed Halfword               |
| REVHS | r3, r7 | ; Reverse with Higher or Same condition |
| RBIT  | r7, r8 |                                         |



### 4.3.13 ASR, LSL, LSR, ROR, and RRX

Arithmetic Shift Right, Logical Shift Left, Logical Shift Right, Rotate Right, and Rotate Right with Extend.

These instructions are synonyms for MOV instructions with shifted register second operands.

#### Syntax

*op*{*S*}{*cond*} *Rd*, *Rm*, *Rs*

*op*{*S*}{*cond*} *Rd*, *Rm*, #*sh*

RRX{*S*}{*cond*} *Rd*, *Rm*

where:

*op* is one of ASR, LSL, LSR, or ROR.

*S* is an optional suffix. If *S* is specified, the condition code flags are updated on the result of the operation (see *Conditional execution* on page 2-17).

*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 depends on the instruction:

|     |                      |
|-----|----------------------|
| ASR | allowed shifts 1-32  |
| LSL | allowed shifts 0-31  |
| LSR | allowed shifts 1-32  |
| ROR | allowed shifts 1-31. |

#### Usage

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.

LSL provides the value of a register multiplied by a power of two. LSR provides the unsigned value of a register divided by a variable power of two. Both instructions insert zeros into the vacated bit positions.

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.

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.

### Condition flags

If S is specified, these instructions update 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 when used in Thumb-2 code:

ASRS *Rd, Rm, #sh*     *Rd* and *Rm* must both be Lo registers. Allowed shifts 1-32.

ASRS *Rd, Rm, Rs*     *Rd, Rm*, and *Rs* must all be Lo registers.

LSLS *Rd, Rm, #sh*     *Rd* and *Rm* must both be Lo registers. Allowed shifts 0-31.

LSLS *Rd, Rm, Rs*     *Rd, Rm*, and *Rs* must all be Lo registers.

LSRS *Rd, Rm, #sh*     *Rd* and *Rm* must both be Lo registers. Allowed shifts 1-32.

LSRS *Rd, Rm, Rs*     *Rd, Rm*, and *Rs* must all be Lo registers.

RORS *Rd, Rm, Rs*     *Rd, Rm*, and *Rs* must all be Lo registers.

### Architectures

These ARM instructions are available in all versions of the ARM architecture.

These 32-bit Thumb-2 instructions are available in T2 variants of ARMv6 and above.

These 16-bit Thumb instructions are available in all T variants of the ARM architecture.

**Examples**

|      |            |
|------|------------|
| ASR  | r7, r8, r9 |
| LSL  | r1, r2, r3 |
| LSLS | r1, r2, r3 |
| LSR  | r4, r5, r6 |
| LSRS | r4, r5, r6 |
| ROR  | r4, r5, r6 |
| RORS | r4, r5, r6 |

#### 4.3.14 IT

The IT (If-Then) 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.

---

##### Note

---

This is a 16-bit Thumb-2 instruction.

---

#### Syntax

`IT{x{y{z}}}{cond}`

where:

|             |                                                              |
|-------------|--------------------------------------------------------------|
| <i>x</i>    | is the condition for the second instruction in the IT block. |
| <i>y</i>    | is the condition for the third instruction in the IT block.  |
| <i>z</i>    | is the condition for the fourth instruction in the IT block. |
| <i>cond</i> | is the condition for the first instruction in the IT block.  |

Conditions for the second, third and fourth instruction in the IT block can be either:

|   |                                                                                |
|---|--------------------------------------------------------------------------------|
| T | Then. The condition applied to the instruction is <i>cond</i> .                |
| E | Else. The condition applied to the instruction is the inverse of <i>cond</i> . |

#### Usage

16-bit instructions in the IT block, other than CMP, CMN and TST, do not affect the condition code flags. You can use IT with the AL condition if you want this effect, without the instructions in the IT block being conditional.

#### Restrictions

The following instructions are not permitted in an IT block:

- IT
- Conditional branches
- CBZ and CBNZ
- TBB and TBH
- CPS, CPSID and CPSIE
- SETEND.

In addition, an unconditional branch is only allowed in an IT block if it is the last instruction in the 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 r14 and SPSR.

Instructions designed for use for 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-2 instruction is available in T2 variants of ARMv6 and above.

## Example

Branches in IT blocks must be encoded as unconditional branches (the IT condition makes them conditional). Therefore, it is sometimes beneficial to place a conditional branch into an IT block as this increases its effective range:

```

ITTE NE ; assemblers can allow IT to be omitted
ANDNE r0,r0,r1 ; 16-bit AND, not ANDS
ADDSE r2,r2,#1 ; 32-bit ADDS (16-bit ADDS does not set flags in IT block)
MOVEQ r2,r3 ; 16-bit MOV(CPY)
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
IT NE
ADD r0,r0,r1 ; syntax error: no condition code used in IT block

```

## Notes

### Branches into an IT block

Except as noted under *Exceptions*, no instruction in an IT block can be the target of any branch, either from a branch instruction or from any other instruction that changes the PC. The result of such a branch is unpredictable, that is, it cannot be relied upon.

**Branches out of an IT block**

Instructions that modify the PC must not be used in an IT block, except as the last instruction in the block. The result of any such instruction is unpredictable, that is, it cannot be relied upon.

**Unpredictable instructions in an IT block**

The assembler warns about unpredictable instructions in an IT block, for example, B, BL, and CPS. This means that the results cannot be relied upon. It also warns about instructions that change the PC, for example, BX, CZB, and RFE.

The assembler does not take account of directives inside IT blocks. Therefore, the following example results in an incorrect warning:

```
ITT EQ
DCI 0
B dloop
```

BKPT     A BKPT instruction in an IT block is always executed, even if its condition fails.

## 4.4 Multiply instructions

This section contains the following subsections:

- *MUL, MLA, and MLS* on page 4-74  
Multiply, Multiply Accumulate, and Multiply Subtract (32-bit by 32-bit, bottom 32-bit result).
- *UMULL, UMLAL, SMULL, and SMLAL* on page 4-76  
Unsigned and signed Long Multiply and Multiply Accumulate (32-bit by 32-bit, 64-bit result or 64-bit accumulator).
- *SMULxy and SMLAxy* on page 4-78  
Signed Multiply and Signed Multiply Accumulate (16-bit by 16-bit, 32-bit result).
- *SMULWy and SMLAWy* on page 4-80  
Signed Multiply and Signed Multiply Accumulate (32-bit by 16-bit, top 32-bit result).
- *SMLALxy* on page 4-82  
Signed Multiply Accumulate (16-bit by 16-bit, 64-bit accumulate).
- *SMUAD{X} and SMUSD{X}* on page 4-84  
Dual 16-bit Signed Multiply with Addition or Subtraction of products.
- *SMMUL, SMMLA, and SMMLS* on page 4-86  
Multiply, Multiply Accumulate, and Multiply Subtract (32-bit by 32-bit, top 32-bit result).
- *SMLAD and SMLSD* on page 4-88  
Dual 16-bit Signed Multiply, 32-bit Accumulation of Sum or Difference of 32-bit products.
- *SMLALD and SMLSLD* on page 4-90  
Dual 16-bit Signed Multiply, 64-bit Accumulation of Sum or Difference of 32-bit products.
- *UMAAL* on page 4-92  
Unsigned Multiply Accumulate Accumulate Long.
- *MIA, MIAPH, and MIAxy* on page 4-93  
XScale coprocessor 0 instructions (Multiplies with Internal Accumulate).

#### 4.4.1 MUL, MLA, and MLS

Multiply, Multiply-Accumulate, and Multiply-Subtract, with signed or unsigned 32-bit operands, giving the least significant 32 bits of the result.

##### Syntax

`MUL{S}{cond} Rd, Rm, Rs`

`MLA{S}{cond} Rd, Rm, Rs, Rn`

`MLS{cond} Rd, Rm, Rs, Rn`

where:

*cond* is an optional condition code (see *Conditional execution* on page 2-17).

*S* is an optional suffix. If *S* is specified, the condition code flags are updated on the result of the operation (see *Conditional execution* on page 2-17).

You cannot use *S* in a Thumb-2 MLA instruction.

You can only use *S* in a Thumb-2 MUL instruction if *Rd* and *Rm* are the same register, and all operands are low registers. This is a 16-bit instruction.

*Rd* is the destination register.

*Rm*, *Rs* are registers holding the values to be multiplied.

*Rn* is a register holding the value to be added or subtracted from.

##### Usage

The MUL instruction multiplies the values from *Rm* and *Rs*, and places the least significant 32 bits of the result in *Rd*.

The MLA instruction multiplies the values from *Rm* and *Rs*, adds the value from *Rn*, and places the least significant 32 bits of the result in *Rd*.

The MLS instruction multiplies the values from *Rm* and *Rs*, subtracts the result from the value from *Rn*, and places the least significant 32 bits of the final result in *Rd*.

Do not use r15 for *Rd*, *Rm*, *Rs*, or *Rn*.



## Condition flags

If S is specified, the MUL and MLA instructions:

- update the N and Z flags according to the result
- corrupt the C and V flag in ARMv4 and earlier
- do not affect the C or V flag in ARMv5 and above.

## 16-bit instructions

The following form of the MUL instruction is available in Thumb code, and is a 16-bit instruction when used in Thumb-2 code:

`MULS Rd, Rd, Rs`      *Rd* and *Rs* must both be Lo registers.

## Architectures

The MUL and MLA ARM instructions are available in all versions of the ARM architecture.

The MLS ARM instruction is available in T2 variants of ARMv6 and above.

These Thumb-2 instructions are available in T2 variants of ARMv6 and above.

The MULS Thumb instruction is available in all T variants of the ARM architecture.

## Examples

```
MUL r10, r2, r5
MLA r10, r2, r1, r5
MULS r0, r2, r2
MULLT r2, r3, r2
MLS r4, r5, r6, r7
```

## Incorrect example

```
MUL r15, r0, r3 ; use of r15 not permitted
```

#### 4.4.2 UMULL, UMLAL, SMULL, and SMLAL

Signed and Unsigned Long Multiply, with optional Accumulate, with 32-bit operands, and 64-bit result and accumulator.

##### Syntax

*Op*{*S*}{*cond*} *RdLo*, *RdHi*, *Rm*, *Rs*

where:

*Op* is one of UMULL, UMLAL, SMULL, or SMLAL.

*S* is an optional suffix available in ARM state only. If *S* is specified, the condition code flags are updated on the result of the operation (see *Conditional execution* on page 2-17).

*cond* is an optional condition code (see *Conditional execution* on page 2-17).

*RdLo*, *RdHi* are the destination registers. For UMLAL and SMLAL they also hold the accumulating value.

*Rm*, *Rs* are ARM registers holding the operands.

##### Usage

Do not use r15 for *RdHi*, *RdLo*, *Rm*, or *Rs*.

*RdLo* and *RdHi* must be different registers.

The UMULL instruction interprets the values from *Rm* and *Rs* 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*.

The UMLAL instruction interprets the values from *Rm* and *Rs* as unsigned integers. It multiplies these integers, and adds the 64-bit result to the 64-bit unsigned integer contained in *RdHi* and *RdLo*.

The SMULL instruction interprets the values from *Rm* and *Rs* 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*.

The SMLAL instruction interprets the values from *Rm* and *Rs* 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*.

## Condition flags

If S is specified, these instructions:

- update the N and Z flags according to the result
- do not affect the C or V flags.

## Architectures

These ARM instructions are available in all supported versions of the ARM architecture.

These 32-bit Thumb-2 instructions are available in T2 variants of ARMv6 and above.

There are no 16-bit Thumb versions of these instructions.

## Examples

```
UMULL r0, r4, r5, r6
UMLALS r4, r5, r3, r8
```

## Incorrect example

```
UMULL r1, r15, r10, r2 ; use of r15 not permitted
```

### 4.4.3 SMULxy and SMLAxy

Signed Multiply and Multiply Accumulate, with 16-bit operands and a 32-bit result and accumulator.

#### Syntax

SMUL<x><y>{cond} Rd, Rm, Rs

SMLA<x><y>{cond} Rd, Rm, Rs, Rn

where:

<x> 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*.

<y> is either B or T. B means use the bottom half (bits [15:0]) of *Rs*, T means use the top half (bits [31:16]) of *Rs*.

cond is an optional condition code (see *Conditional execution* on page 2-17).

Rd is the destination register.

Rm, Rs are the registers holding the values to be multiplied.

Rn is the register holding the value to be added.

#### Usage

Do not use r15 for *Rd*, *Rm*, *Rs*, or *Rn*.

SMULxy multiplies the 16-bit signed integers from the selected halves of *Rm* and *Rs*, and places the 32-bit result in *Rd*.

SMLAxy multiplies the 16-bit signed integers from the selected halves of *Rm* and *Rs*, adds the 32-bit result to the 32-bit value in *Rn*, and places the result in *Rd*.

#### Condition flags

These instructions do 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 (see *MRS* on page 4-134).

#### ———— Note ————

SMLAxy never clears the Q flag. To clear the Q flag, use an MSR instruction (see *MSR* on page 4-135).

## Architectures

These ARM instructions are available in ARMv6 and above, and E variants of ARMv5.

These 32-bit Thumb-2 instructions are available in T2 variants of ARMv6 and above.

There are no 16-bit Thumb versions of these instructions.

## Examples

```
SMULTBEQ r8, r7, r9
SMLABBNE r0, r2, r1, r10
SMLABT r0, r0, r3, r5
```

## Incorrect examples

```
SMLATB r0,r7,r8,r15 ; use of r15 not permitted
SMLATTSS r0,r6,r2 ; use of S suffix not permitted
```

#### 4.4.4 SMULWy and SMLAWy

Signed Multiply Wide and Signed Multiply-Accumulate Wide, with one 32-bit and one 16-bit operand, providing the top 32-bits of the result.

##### Syntax

SMULW<y>{cond} Rd, Rm, Rs

SMLAW<y>{cond} Rd, Rm, Rs, Rn

where:

<y> is either B or T. B means use the bottom half (bits [15:0]) of Rs, T means use the top half (bits [31:16]) of Rs.

cond is an optional condition code (see *Conditional execution* on page 2-17).

Rd is the destination register.

Rm, Rs are the registers holding the values to be multiplied.

Rn is the register holding the value to be added.

##### Usage

Do not use r15 for Rd, Rm, Rs, or Rn.

SMULWy multiplies the signed integer from the selected half of Rs by the signed integer from Rm, and places the upper 32-bits of the 48-bit result in Rd.

SMLAWy multiplies the signed integer from the selected half of Rs by the signed integer from Rm, adds the 32-bit result to the 32-bit value in Rn, and places the result in Rd.

##### Condition flags

These instructions do not change the flags.

##### Architectures

These ARM instructions are available in ARMv6 and above, and E variants of ARMv5.

These 32-bit Thumb-2 instructions are available in T2 variants of ARMv6 and above.

There are no 16-bit Thumb versions of these instructions.

**Examples**

|          |            |
|----------|------------|
| SMULWB   | r2, r4, r7 |
| SMULWTVS | r0, r0, r9 |

#### 4.4.5 SMLALxy

Signed Multiply-Accumulate with 16-bit operands and a 64-bit accumulator.

##### Syntax

SMLAL<x><y>{cond} RdLo, RdHi, Rm, Rs

where:

<x> 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*.

<y> is either B or T. B means use the bottom half (bits [15:0]) of *Rs*, T means use the top half (bits [31:16]) of *Rs*.

cond is an optional condition code (see *Conditional execution* on page 2-17).

RdHi, RdLo are the destination registers. They also hold the accumulate value.

Rm, Rs are the registers holding the values to be multiplied.

##### Usage

Do not use r15 for *RdHi*, *RdLo*, *Rm*, or *Rs*.

*RdHi* and *RdLo* must be different registers.

SMLALxy multiplies the signed integer from the selected half of *Rs* by the signed integer from the selected half of *Rm*, and adds the 32-bit result to the 64-bit value in *RdHi* and *RdLo*.

##### 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 ARMv5.

This 32-bit Thumb-2 instruction is available in T2 variants of ARMv6 and above.

There is no 16-bit Thumb version of this instruction.

## Examples

```
SMLALTB r2, r3, r7, r1
SMLALBTVS r0, r1, r9, r2
```

## Incorrect example

```
SMLALTT r8, r9, r3, r15 ; use of r15 not permitted
```

#### 4.4.6 SMUAD{X} and SMUSD{X}

Dual 16-bit Signed Multiply with Addition or Subtraction of products, and optional exchange of operand halves.

##### Syntax

*op*{*X*}{*cond*} *Rd*, *Rm*, *Rs*

where:

*op* is one of:

SMUAD Dual multiply, add products.

SMUSD Dual multiply, subtract products.

*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 (see *Conditional execution* on page 2-17).

*Rd* is the destination register.

*Rm*, *Rs* are the registers holding the operands.

##### Operation

Do not use r15 for any of *Rd*, *Rm*, or *Rs*.

SMUAD multiplies the bottom halfword of *Rm* with the bottom halfword of *Rs*, and the top halfword of *Rm* with the top halfword of *Rs*. It then adds the products and stores the sum to *Rd*.

SMUSD multiplies the bottom halfword of *Rm* with the bottom halfword of *Rs*, and the top halfword of *Rm* with the top halfword of *Rs*. It then subtracts the second product from the first, and stores the difference to *Rd*.

##### Condition flags

These instructions do not change the flags.

The SMUAD instruction sets the Q flag if the addition overflows.

## Architectures

These ARM instructions are available in ARMv6 and above, and E variants of ARMv5.

These 32-bit Thumb-2 instructions are available in T2 variants of ARMv6 and above.

There are no 16-bit Thumb versions of these instructions.

## Examples

|          |            |
|----------|------------|
| SMUAD    | r2, r3, r2 |
| SMUSDXNE | r0, r1, r2 |

#### 4.4.7 SMMUL, SMMLA, and SMMLS

Signed Most significant word Multiply, with optional Accumulation or Subtraction, with 32-bit operands and producing only the most significant 32-bits of the result.

##### Syntax

*op*{*R*}{*cond*} *Rd*, *Rm*, *Rs*, *Rn*

where:

*op* is one of:

SMMUL Multiply, and truncate or round.

SMMLA Multiply, accumulate, and truncate or round.

SMMLS Multiply, subtract from *Rn*, and truncate or round.

*R* is an optional parameter. If *R* is present, the result is rounded, otherwise it is truncated.

*cond* is an optional condition code (see *Conditional execution* on page 2-17).

*Rd* is the destination register.

*Rm*, *Rs* are the registers holding the operands.

*Rn* is a register holding the value to be added or subtracted from.

##### Operation

SMMUL multiplies the values from *Rm* and *Rs*, and stores the most significant 32 bits of the 64-bit result to *Rd*.

SMMLA multiplies the values from *Rm* and *Rs*, adds the value in *Rn* to the most significant 32 bits of the product, and stores the result in *Rd*.

SMMLS multiplies the values from *Rm* and *Rs*, subtracts the product from the value in *Rn* 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.

##### Usage

Do not use r15 for any of *Rd*, *Rm*, *Rs*, or *Rn*.

### Condition flags

These instructions do not change the flags.

### Architectures

These ARM instructions are available in ARMv6 and above, and E variants of ARMv5.

These 32-bit Thumb-2 instructions are available in T2 variants of ARMv6 and above.

There are no 16-bit Thumb versions of these instructions.

### Examples

|         |            |
|---------|------------|
| SMMULGE | r6, r4, r3 |
| SMMULR  | r2, r2, r2 |

#### 4.4.8 SMLAD and SMLSD

Dual 16-bit Signed Multiply with Addition or Subtraction of products and 32-bit accumulation.

##### Syntax

*op*{*X*}{*cond*} *Rd*, *Rm*, *Rs*, *Rn*

where:

|                       |                                                                                                                                                                  |
|-----------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>op</i>             | is one of:                                                                                                                                                       |
| SMLAD                 | Dual multiply, accumulate sum of products.                                                                                                                       |
| SMLSD                 | Dual multiply, accumulate difference of products.                                                                                                                |
| <i>cond</i>           | is an optional condition code (see <i>Conditional execution</i> on page 2-17).                                                                                   |
| <i>X</i>              | is an optional parameter. If <i>X</i> is present, the most and least significant halfwords of the second operand are exchanged before the multiplications occur. |
| <i>Rd</i>             | is the destination register.                                                                                                                                     |
| <i>Rm</i> , <i>Rs</i> | are the registers holding the operands.                                                                                                                          |
| <i>Rn</i>             | is the register holding the accumulate operand.                                                                                                                  |

##### Operation

SMLAD multiplies the bottom halfword of *Rm* with the bottom halfword of *Rs*, and the top halfword of *Rm* with the top halfword of *Rs*. It then adds both products to the value in *Rn* and stores the sum to *Rd*.

SMLSD multiplies the bottom halfword of *Rm* with the bottom halfword of *Rs*, and the top halfword of *Rm* with the top halfword of *Rs*. It then subtracts the second product from the first, adds the difference to the value in *Rn*, and stores the result to *Rd*.

##### Usage

Do not use r15 for any of *Rd*, *Rm*, *Rs*, or *Rn*.

##### Condition flags

These instructions do not change the flags.

## Architectures

These ARM instructions are available in ARMv6 and above, and E variants of ARMv5.

These 32-bit Thumb-2 instructions are available in T2 variants of ARMv6 and above.

There are no 16-bit Thumb versions of these instructions.

## Examples

|         |                  |
|---------|------------------|
| SMLSD   | r1, r2, r0, r7   |
| SMLSDX  | r11, r10, r2, r3 |
| SMLADLT | r1, r2, r4, r1   |

#### 4.4.9 SMLALD and SMLSLD

Dual 16-bit Signed Multiply with Addition or Subtraction of products and 64-bit Accumulation.

##### Syntax

*op*{*X*}{*cond*} *RdLo*, *RdHi*, *Rm*, *Rs*

where:

*op* is one of:

SMLALD Dual multiply, accumulate sum of products.

SMLSLD Dual multiply, accumulate difference of products.

*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 (see *Conditional execution* on page 2-17).

*RdLo*, *RdHi* are the destination registers for the 64-bit result. They also hold the 64-bit accumulate operand.

*Rm*, *Rs* are the registers holding the operands.

##### Usage

Do not use r15 for any of *RdLo*, *RdHi*, *Rm*, or *Rs*.

##### Operation

SMLALD multiplies the bottom halfword of *Rm* with the bottom halfword of *Rs*, and the top halfword of *Rm* with the top halfword of *Rs*. It then adds both products to the value in *RdLo*, *RdHi* and stores the sum to *RdLo*, *RdHi*.

SMLSLD multiplies the bottom halfword of *Rm* with the bottom halfword of *Rs*, and the top halfword of *Rm* with the top halfword of *Rs*. 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*.

##### Condition flags

These instructions do not change the flags.



## Architectures

These ARM instructions are available in ARMv6 and above, and E variants of ARMv5.

These 32-bit Thumb-2 instructions are available in T2 variants of ARMv6 and above.

There are no 16-bit Thumb versions of these instructions.

## Examples

|        |                  |
|--------|------------------|
| SMLALD | r10, r11, r5, r1 |
| SMLSLD | r3, r0, r5, r1   |

#### 4.4.10 UMAAL

Unsigned Multiply Accumulate Accumulate Long.

##### Syntax

UMAAL{*cond*} *RdLo*, *RdHi*, *Rm*, *Rs*

where:

*cond* is an optional condition code (see *Conditional execution* on page 2-17).

*RdLo*, *RdHi* are the destination registers for the 64-bit result. They also hold the two 32-bit accumulate operands.

*Rm*, *Rs* are the registers holding the multiply operands.

##### Operation

The UMAAL instruction multiplies the 32-bit values in *Rm* and *Rs*, adds the two 32-bit values in *RdHi* and *RdLo*, and stores the 64-bit result to *RdLo*, *RdHi*.

##### Usage

Do not use r15 for any of *RdLo*, *RdHi*, *Rm*, or *Rs*.

*RdLo* and *RdHi* must be distinct registers, otherwise the results are unpredictable, that is, they cannot be relied upon.

##### Condition flags

This instruction does not change the flags.

##### Architectures

This ARM instruction is available in ARMv6 and above, and E variants of ARMv5.

This 32-bit Thumb-2 instruction is available in T2 variants of ARMv6 and above.

There is no 16-bit Thumb version of this instruction.

##### Examples

|         |                |
|---------|----------------|
| UMAAL   | r8, r9, r2, r3 |
| UMAALGE | r2, r0, r5, r3 |

#### 4.4.11 MIA, MIAPH, and MIAxy

XScale coprocessor 0 instructions.

Multiply with internal accumulate (32-bit by 32-bit, 40-bit accumulate).

Multiply with internal accumulate, packed halfwords (16-bit by 16-bit twice, 40-bit accumulate).

Multiply with internal accumulate (16-bit by 16-bit, 40-bit accumulate).

##### Syntax

`MIA{cond} Acc, Rm, Rs`

`MIAPH{cond} Acc, Rm, Rs`

`MIA<x><y>{cond} Acc, Rm, Rs`

where:

|               |                                                                                                                                                                                                          |
|---------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>cond</i>   | is an optional condition code (see <i>Conditional execution</i> on page 2-17).                                                                                                                           |
| <i>Acc</i>    | is the internal accumulator. The standard name is <i>accx</i> , where <i>x</i> is an integer in the range 0 to <i>n</i> . The value of <i>n</i> depends on the processor. It is 0 in current processors. |
| <i>Rm, Rs</i> | are the ARM registers holding the values to be multiplied.<br>Do not use r15 for either <i>Rm</i> or <i>Rs</i> .                                                                                         |
| < <i>x</i> >  | is either B or T. B means use the bottom half (bits [15:0]) of <i>Rm</i> , T means use the top half (bits [31:16]) of <i>Rm</i> .                                                                        |
| < <i>y</i> >  | is either B or T. B means use the bottom half (bits [15:0]) of <i>Rs</i> , T means use the top half (bits [31:16]) of <i>Rs</i> .                                                                        |

##### Usage

The MIA instruction multiplies the signed integers from *Rs* and *Rm*, and adds the result to the 40-bit value in *Acc*.

The MIAPH instruction multiplies the signed integers from the lower halves of *Rs* and *Rm*, multiplies the signed integers from the upper halves of *Rs* and *Rm*, and adds the two 32-bit results to the 40-bit value in *Acc*.

The MIAxy instruction multiplies the signed integer from the selected half of *Rs* by the signed integer from the selected half of *Rm*, and adds the 32-bit result to the 40-bit value in *Acc*.

## Condition flags

These instructions do not change the flags.

---

### Note

---

These instructions cannot raise an exception. If overflow occurs on these instructions, the result wraps round without any warning.

---

## Architectures

These ARM instructions are only available in XScale processors.

There are no Thumb or Thumb-2 versions of these instructions.

## Examples

|         |                |
|---------|----------------|
| MIA     | acc0, r5, r0   |
| MIALE   | acc0, r1, r9   |
| MIAPH   | acc0, r0, r7   |
| MIAPHNE | acc0, r11, r10 |
| MIABB   | acc0, r8, r9   |
| MIABT   | acc0, r8, r8   |
| MIATB   | acc0, r5, r3   |
| MIATT   | acc0, r0, r6   |
| MIABTGT | acc0, r2, r5   |

## 4.5 Saturating instructions

This section contains the following subsections:

- *What are saturating instructions?*
- *QADD, QSUB, QDADD, and QDSUB* on page 4-96
- *SSAT and USAT* on page 4-98.

Some of the parallel instructions are also saturating, see *Parallel instructions* on page 4-100.

### 4.5.1 What are saturating instructions?

These operations are *saturating* (SAT). This 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 things 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 (see *MSR* on page 4-135).

---

The Q flag can also be set by two other instructions (see *SMULxy and SMLAx*y on page 4-78 and *SMULWy and SMLAWy* on page 4-80), but these instructions do not saturate.

## 4.5.2 QADD, QSUB, QDADD, and QDSUB

Signed Add, Subtract, Double and Add, Double and Subtract, saturating the result to the signed range  $-2^{31} \leq x \leq 2^{31}-1$ .

See also *Parallel add and subtract* on page 4-101.

### Syntax

*op{cond}* *Rd*, *Rm*, *Rn*

where:

*op* is one of QADD, QSUB, QDADD, or QDSUB.

*cond* is an optional condition code (see *Conditional execution* on page 2-17).

*Rd* is the destination register.

*Rm*, *Rn* are the registers holding the operands.

Do not use r15 for *Rd*, *Rm*, or *Rn*.

### Usage

The QADD instruction adds the values in *Rm* and *Rn*.

The QSUB instruction subtracts the value in *Rn* from the value in *Rm*.

The QDADD instruction calculates  $\text{SAT}(Rm + \text{SAT}(Rn * 2))$ . 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.

The QDSUB instruction calculates  $\text{SAT}(Rm - \text{SAT}(Rn * 2))$ . 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 these instructions.

See also *Parallel add and subtract* on page 4-101 for similar parallel instructions, available in ARMv6 and above only.

## Condition flags

If saturation occurs, these instructions set the Q flag. To read the state of the Q flag, use an MRS instruction (see *MRS* on page 4-134).

No other flags are affected.

## Architectures

These ARM instructions are available in ARMv6 and above, and E variants of ARMv5.

These 32-bit Thumb-2 instructions are available in T2 variants of ARMv6 and above.

There are no 16-bit Thumb versions of these instructions.

## Examples

```
QADD r0, r1, r9
QDSUBLT r9, r0, r1
```

## Incorrect examples

```
QSUBS r3, r4, r2 ; use of S suffix not permitted
QDADD r11, r15, r0 ; use of r15 not permitted
```

### 4.5.3 SSAT and USAT

Signed Saturate and Unsigned Saturate to any bit position, with optional shift before saturating.

See also *SSAT16 and USAT16* on page 4-106.

#### Syntax

*op*{*cond*} *Rd*, #*sat*, *Rm*{, *shift*}

where:

*op* is either SSAT or USAT.

*cond* is an optional condition code (see *Conditional execution* on page 2-17).

*Rd* is the destination register. *Rd* must not be r15.

*sat* specifies the bit position to saturate to, in the range 0 to 31.

*Rm* is the register containing the operand. *Rm* must not be r15.

*shift* is an optional shift. It must be one of the following:

ASR #*n* where *n* is in the range 1-32

LSL #*n* where *n* is in the range 0-31.

#### Operation

The SSAT instruction applies the specified shift, then saturates to the signed range  $-2^{\text{sat}-1} \leq x \leq 2^{\text{sat}-1} - 1$ .

The USAT instruction applies the specified shift, then saturates to the unsigned range  $0 \leq x \leq 2^{\text{sat}} - 1$ .

#### Condition flags

If saturation occurs, these instructions set the Q flag. To read the state of the Q flag, use an MRS instruction (see *MRS* on page 4-134).

#### Architectures

These ARM instructions are available in ARMv6 and above, and E variants of ARMv5.

These 32-bit Thumb-2 instructions are available in T2 variants of ARMv6 and above.

There are no 16-bit Thumb versions of these instructions.



**Examples**

```
SSAT r7, #16, r7, LSL #4
USATNE r0, #7, r5
```

**Incorrect example**

```
USATGT r0, #7, r15, ASR #16 ; use of r15 not permitted
```

## 4.6 Parallel instructions

This section contains the following subsections:

- *Parallel add and subtract* on page 4-101  
Various byte-wise and halfword-wise additions and subtractions.
- *USAD8 and USADA8* on page 4-104  
Unsigned sum of absolute differences, and accumulate unsigned sum of absolute differences.
- *SSAT16 and USAT16* on page 4-106  
Parallel halfword saturating instructions.

There are also some parallel unpacking instructions, see *SXT*, *SXTA*, *UXT*, and *UXTA* on page 4-110.

## 4.6.1 Parallel add and subtract

Various byte-wise and halfword-wise additions and subtractions.

### Syntax

`<prefix>op{cond} Rd, Rn, Rm`

where:

|                             |                                                                                         |
|-----------------------------|-----------------------------------------------------------------------------------------|
| <code>&lt;prefix&gt;</code> | is one of:                                                                              |
| S                           | Signed arithmetic modulo $2^8$ or $2^{16}$ . Sets CPSR GE flags.                        |
| Q                           | Signed saturating arithmetic.                                                           |
| SH                          | Signed arithmetic, halving the results.                                                 |
| U                           | Unsigned arithmetic modulo $2^8$ or $2^{16}$ . Sets CPSR GE flags.                      |
| UQ                          | Unsigned saturating arithmetic.                                                         |
| UH                          | Unsigned arithmetic, halving the results.                                               |
| <code>op</code>             | is one of:                                                                              |
| ADD8                        | Byte-wise Addition                                                                      |
| ADD16                       | Halfword-wise Addition.                                                                 |
| SUB8                        | Byte-wise Subtraction.                                                                  |
| SUB16                       | Halfword-wise Subtraction.                                                              |
| ASX                         | Exchange halfwords of <i>Rm</i> , then Add top halfwords and Subtract bottom halfwords. |
| SAX                         | Exchange halfwords of <i>Rm</i> , then Subtract top halfwords and Add bottom halfwords. |
| <code>cond</code>           | is an optional condition code (see <i>Conditional execution</i> on page 2-17).          |
| <i>Rd</i>                   | is the destination register. Do not use r15 for <i>Rd</i> .                             |
| <i>Rm</i> , <i>Rn</i>       | are the ARM registers holding the operands. Do not use r15 for <i>Rm</i> or <i>Rn</i> . |

### Operation

These instructions perform arithmetic operations separately on the bytes or halfwords of the operands. They perform two or four additions or subtractions, or one addition and one subtraction.

You can choose various kinds of arithmetic:

- Signed or unsigned arithmetic modulo  $2^8$  or  $2^{16}$ . This sets the CPSR GE flags (see *Condition flags*).
- Signed saturating arithmetic to one of the signed ranges  $-2^{15} \leq x \leq 2^{15} - 1$  or  $-2^7 \leq x \leq 2^7 - 1$ . The Q flag is not affected even if these operations saturate.
- Unsigned saturating arithmetic to one of the unsigned ranges  $0 \leq x \leq 2^{16} - 1$  or  $0 \leq x \leq 2^8 - 1$ . The Q flag is not affected even if these operations saturate.
- Signed or unsigned arithmetic, halving the results. This cannot cause overflow.

### Condition flags

These instructions do not affect the N, Z, C, V, or Q flags.

The Q, SH, UQ and UH prefix variants of these instructions do not change the flags.

The S and U prefix variants of these instructions set the GE flags in the CPSR as follows:

- For byte-wise operations, the GE flags are used in the same way as the C (Carry) flag for 32-bit SUB and ADD instructions:
  - 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.
- For halfword-wise operations, the GE flags are used in the same way as the C (Carry) flag for normal word-wise SUB and ADD instructions:
  - GE[1:0] for bits[15:0] of the result
  - GE[3:2] for bits[31:16] of the result.

You can use these flags to control a following SEL instruction, see *SEL* on page 4-63.

---

#### Note

For halfword-wise operations, GE[1:0] are set or cleared together, and GE[3:2] are set or cleared together.

---

## Architectures

These ARM instructions are available in ARMv6 and above.

These 32-bit Thumb-2 instructions are available in T2 variants of ARMv6 and above.

There are no 16-bit Thumb versions of these instructions.

## Examples

```
SHADD8 r4, r3, r9
USAXNE r0, r0, r2
```

## Incorrect examples

```
QHADD r2, r9, r3 ; No such instruction, should be QHADD8 or QHADD16
UQSUB16NE r1, r15, r0 ; Use of r15 not permitted
SAX r10, r8, r5 ; Must have a prefix.
```

## 4.6.2 USAD8 and USADA8

Unsigned Sum of Absolute Differences, and Accumulate unsigned sum of absolute differences.

### Syntax

USAD8{*cond*} *Rd*, *Rm*, *Rs*

USADA8{*cond*} *Rd*, *Rm*, *Rs*, *Rn*

where:

*cond* is an optional condition code (see *Conditional execution* on page 2-17).

*Rd* is the destination register.

*Rm* is the register holding the first operand.

*Rs* is the register holding the second operand.

*Rn* is the register holding the accumulate operand.

Do not use r15 for *Rd*, *Rm*, *Rs*, or *Rn*.

### Operation

The USAD8 instruction finds the four differences between the unsigned values in corresponding bytes of *Rm* and *Operand2*. It adds the absolute values of the four differences, and saves the result to *Rd*.

The USADA8 instruction adds the absolute values of the four differences to the value in *Rn*, and saves the result to *Rd*.

### Condition flags

These instructions do not alter any flags.

### Architectures

These ARM instructions are available in ARMv6 and above.

These 32-bit Thumb-2 instructions are available in T2 variants of ARMv6 and above.

There are no 16-bit Thumb versions of these instructions.

**Examples**

```
USAD8 r2, r4, r6
USADA8 r0, r3, r5, r2
USADA8VS r0, r4, r0, r1
```

**Incorrect examples**

```
USADA8 r2, r4, r6 ; USADA8 requires four registers
USAD8CC r0, r3, r15 ; use of r15 not permitted
USADA16 r0, r4, r0, r1 ; no such instruction
```

### 4.6.3 SSAT16 and USAT16

Parallel halfword Saturating instructions.

#### Syntax

*op*{*cond*} *Rd*, #*sat*, *Rm*

where:

*op* is one of:

SSAT16 Signed saturation.

USAT16 Unsigned saturation.

*cond* is an optional condition code (see *Conditional execution* on page 2-17).

*Rd* is the destination register.

*sat* specifies the bit position to saturate to, and is in the range 1 to 16 for SSAT16, or 0 to 15 for USAT16.

*Rm* is the register holding the operand.

Do not use r15 for *Rd* or *Rm*.

#### Operation

Halfword-wise signed and unsigned saturation to any bit position.

The SSAT16 instruction saturates each halfword to the signed range  $-2^{\text{sat}} - 1 \leq x \leq 2^{\text{sat}} - 1$ .

The USAT16 instruction saturates each halfword to the unsigned range  $0 \leq x \leq 2^{\text{sat}} - 1$ .

#### Condition flags

If saturation occurs on either halfword, these instructions set the Q flag. To read the state of the Q flag, use an MRS instruction (see *MRS* on page 4-134).

#### Architectures

These ARM instructions are available in ARMv6 and above.

These 32-bit Thumb-2 instructions are available in T2 variants of ARMv6 and above.

There are no 16-bit Thumb versions of these instructions.



**Examples**

```
SSAT16 r7, #12, r7
USAT16 r0, #7, r5
```

**Incorrect examples**

```
SSAT16 r1, #16, r2, LSL #4 ; shifts not permitted with halfword saturations
USAT16 r0, #11, r15 ; use of r15 not permitted
```

## 4.7 Packing and unpacking instructions

This section contains the following subsections:

- *SBFX and UBFX* on page 4-109  
Signed or Unsigned Bit Field extract.
- *SXT, SXTA, UXT, and UXTA* on page 4-110  
Sign Xxtend or zero extend instructions, with optional Add.
- *PKHBT and PKHTB* on page 4-113  
Halfword PacKing instructions.

### 4.7.1 SBFX and UBFX

Signed and Unsigned Bit Field extract. Copies adjacent bits from one register into the least significant bits of a second register, and sign extends or zero extends to 32 bits.

#### Syntax

*op{cond} Rd, Rm, #lsb, #width*

where:

*op* is either SBFX or UBFX.

*cond* is an optional condition code (see *Conditional execution* on page 2-17).

*Rd* is the destination register.

*Rm* is the source register.

*lsb* is the bit number of least significant bit in the bitfield, in the range 0 to 31.

*width* is the width of the bitfield, in the range 1 to (1+*lsb*).

Do not use r15 for *Rd* or *Rn*

#### Condition flags

These instructions do not alter any flags.

#### Architectures

These ARM instructions are available in ARMv6T2 and above.

These 32-bit Thumb-2 instructions are available in T2 variants of ARMv6 and above.

There are no 16-bit Thumb versions of these instructions.

## 4.7.2 SXT, SXTA, UXT, and UXTA

Sign extend or zero extend, with optional Add.

These instructions do any one of the following:

- Sign or zero extend an 8-bit value to 32 bits, and optionally add a 32-bit value.
- Sign or zero extend a 16-bit value to 32 bits, and optionally add a 32-bit value.
- Sign or zero extend two 8-bit values to two 16-bit values, and optionally add two 16-bit values.

### Syntax

*op*<extend>{*cond*} *Rd*, {*Rn*,} *Rm*{, *rotation*}

where:

*op* is one of:

|      |                      |
|------|----------------------|
| SXT  | Sign extend.         |
| UXT  | Zero extend.         |
| SXTA | Sign extend and add. |
| UXTA | Zero extend and add. |

<extend> is one of:

|     |                                                |
|-----|------------------------------------------------|
| B16 | Extends two 8-bit values to two 16-bit values. |
| B   | Extends an 8-bit value to a 32-bit value.      |
| H   | Extends a 16-bit value to a 32-bit value.      |

*cond* is an optional condition code (see *Conditional execution* on page 2-17).

*Rd* is the destination register.

*Rn* is the register holding the number to add (SXTA and UXTA only).

*Rm* is the register holding the value to extend.

*rotation* is one of:

|         |                                                |
|---------|------------------------------------------------|
| ROR #8  | Value from <i>Rm</i> is rotated right 8 bits.  |
| ROR #16 | Value from <i>Rm</i> is rotated right 16 bits. |
| ROR #24 | Value from <i>Rm</i> is rotated right 24 bits. |

If *rotation* is omitted, no rotation is performed.

You must not use r15 for *Rd*, *Rn*, or *Rm*.

## Operation

These instructions do the following:

1. Rotate the value from *Rm* right by 0, 8, 16 or 24 bits (ARM and Thumb-2 only).
2. Do one of the following to the value obtained:
  - Extract bits[7:0], sign or zero extend to 32 bits. If the instruction is extend and add, add the value from *Rn*.
  - Extract bits[15:0], sign or zero extend to 32 bits. If the instruction is extend and add, add the value from *Rn*.
  - Extract bits[23:16] and bits[7:0] and sign or zero extend them to 16 bits. If the instruction is extend and add, 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.

## Condition flags

These instructions do not change the flags.

## 16-bit instructions

The following forms of these instructions are available in Thumb code, and are 16-bit instructions when used in Thumb-2 code:

|                            |                                                    |
|----------------------------|----------------------------------------------------|
| SXTB <i>Rd</i> , <i>Rm</i> | <i>Rd</i> and <i>Rm</i> must both be Lo registers. |
| SXTH <i>Rd</i> , <i>Rm</i> | <i>Rd</i> and <i>Rm</i> must both be Lo registers. |
| UXTB <i>Rd</i> , <i>Rm</i> | <i>Rd</i> and <i>Rm</i> must both be Lo registers. |
| UXTH <i>Rd</i> , <i>Rm</i> | <i>Rd</i> and <i>Rm</i> must both be Lo registers. |

## Architectures

These ARM instructions are available in ARMv6 and above.

These 32-bit Thumb-2 instructions are available in T2 variants of ARMv6 and above.

These 16-bit Thumb instructions are available in ARMv6 and above.

## Examples

```
SXTH r3, r9, r4
UXTAB16EQ r0, r0, r4, ROR #16
```

## Incorrect examples

```
UXTAB r0, r2, r15 ; use of r15 not permitted
SXTH r9, r3, r2, ROR #12 ; rotation must be by 0, 8, 16, or 24.
```

### 4.7.3 PKHBT and PKHTB

Halfword Packing instructions.

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

*op{cond} Rd, Rn, Rm{, shift}*

where:

*op* is one of:

- |       |                                                                                         |
|-------|-----------------------------------------------------------------------------------------|
| PKHBT | Combines bits[15:0] of <i>Rn</i> with bits[31:16] of the shifted value from <i>Rm</i> . |
| PKHTB | Combines bits[31:16] of <i>Rn</i> with bits[15:0] of the shifted value from <i>Rm</i> . |

*cond* is an optional condition code (see *Conditional execution* on page 2-17).

*Rd* is the destination register.

*Rn* is the register holding the first operand.

*Rm* is the register holding the first operand.

*shift* is one of:

- |               |                                                                                     |
|---------------|-------------------------------------------------------------------------------------|
| LSL <i>#n</i> | Logical Shift Left. <i>n</i> is in the range 0 to 31. Only available for PKHBT.     |
| ASR <i>#n</i> | Arithmetic Shift Right. <i>n</i> is in the range 1 to 32. Only available for PKHTB. |

Do not use r15 for *Rd*, *Rn*, or *Rm*.

#### Condition flags

These instructions do not change the flags.

#### Architectures

These ARM instructions are available in ARMv6 and above.

These 32-bit Thumb-2 instructions are available in T2 variants of ARMv6 and above.

There are no 16-bit Thumb versions of these instructions.

## 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 examples

```
PKHBT r4, r15, r1 ; use of r15 not permitted
PKHBTQ r4, r5, r1, ASR #8 ; ASR not permitted with PKHBT
```



## 4.8 Branch instructions

This section contains the following subsections:

- *B, BL, BX, BLX, and BXJ* on page 4-116  
Branch, Branch with Link, Branch and exchange instruction set, Branch with Link and exchange instruction set, Branch and change instruction set to Jazelle.
- *CBZ and CBNZ* on page 4-119  
Compare against zero and branch.
- *TBB and TBH* on page 4-120  
Table Branch Byte or Halfword.

### 4.8.1 B, BL, BX, BLX, and BXJ

Branch, Branch with Link, Branch and exchange instruction set, Branch with Link and exchange instruction set, Branch and change to Jazelle state.

#### Syntax

$op\{cond\}\{.W\} \text{ label}$

$op\{cond\} Rm$

where:

$op$  is one of:

|     |                                                 |
|-----|-------------------------------------------------|
| B   | Branch.                                         |
| BL  | Branch with link.                               |
| BX  | Branch and exchange instruction set.            |
| BLX | Branch with link, and exchange instruction set. |
| BXJ | Branch, and change to Jazelle execution.        |

$cond$  is an optional condition code (see *Conditional execution* on page 2-17).  $cond$  is not available on all forms of this instruction, see *Instruction availability and branch ranges* on page 4-117.

$.W$  is an optional instruction width specifier to force the use of a 32-bit B instruction in Thumb-2. See *B in Thumb-2* on page 4-118 for details.

$label$  is a program-relative expression. See *Register-relative and program-relative expressions* on page 3-28 for more information.

$Rm$  is a register containing an address to branch to.

#### Operation

All these instructions cause a branch to  $label$ , or to the address contained in  $Rm$ . In addition:

- The BL and BLX instructions copy the address of the next instruction into r14 (lr, the link register).
- The BX and BLX instructions can change the processor state from ARM to Thumb, or from Thumb to ARM.

BLX  $label$  always changes the state.

BX  $Rm$  and BLX  $Rm$  derive the target state 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.
- The BXJ instruction changes the processor state to Jazelle.

### Instruction availability and branch ranges

Table 4-2 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 4-2 Branch instruction availability and range**

| Instruction    | ARM              |         | 16-bit Thumb      |         | 32-bit Thumb-2     |
|----------------|------------------|---------|-------------------|---------|--------------------|
| B label        | ±32MB            | (All)   | ±2KB              | (All T) | ±16MB <sup>a</sup> |
| B{cond} label  | ±32MB            | (All)   | –252 to +258      | (All T) | ±1MB <sup>a</sup>  |
| B Rm           | Use BX Rm        |         | Use BX Rm         |         | Use 16-bit BX Rm   |
| B{cond} Rm     | Use BX{cond} Rm  |         | -                 |         | -                  |
| BL label       | ±32MB            | (All)   | ±4MB <sup>b</sup> | (All T) | ±16MB              |
| BL{cond} label | ±32MB            | (All)   | -                 |         | -                  |
| BL Rm          | Use BLX Rm       |         | Use BLX Rm        |         | Use 16-bit BLX Rm  |
| BL{cond} Rm    | Use BLX{cond} Rm |         | -                 |         | -                  |
| BX Rm          | Available        | (4T, 5) | Available         | (All T) | Use 16-bit         |
| BX{cond} Rm    | Available        | (4T, 5) | -                 |         | -                  |
| BLX label      | ±32MB            | (5)     | ±4MB <sup>c</sup> | (5T)    | ±16MB              |
| BLX Rm         | Available        | (5)     | Available         | (5T)    | Use 16-bit         |
| BLX{cond} Rm   | Available        | (5)     | -                 |         | -                  |
| BXJ Rm         | Available        | (5J, 6) | -                 |         | Available          |
| BXJ{cond} Rm   | Available        | (5J, 6) | -                 |         | -                  |

a. Use .W to instruct the assembler to use this 32-bit instruction.

b. This is an instruction pair.

c. This is an instruction pair.

## Extending branch ranges

Machine-level B and 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 (see the chapter describing basic linker functionality in *RealView Compilation Tools v2.2 Linker and Utilities Guide*). The added code is called a *veneer*.

## B in Thumb-2

You can use the .W width specifier to force B to generate a 32-bit instruction in Thumb-2 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-2 instruction.

## Condition flags

These instructions do not change the flags.

## Architectures

See *Instruction availability and branch ranges* on page 4-117 for details of availability of these instructions in each architecture.

## Examples

```

B loopA
BLE ng+8
BL subC
BLLT rtX
BEQ {pc}+4 ; #0x8004
```

## 4.8.2 CBZ and CBNZ

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

---

### Note

---

These are Thumb-2 16-bit instructions.

---

### 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 save one instruction in compare with zero and branch code sequences.

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

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

Except that it does not change the condition code 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.

These instructions must not be used inside an IT block.

### Condition flags

These instructions do not change the flags.

### Architectures

These 16-bit Thumb-2 instructions are available in T2 variants of ARMv6 and above.

### 4.8.3 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* can be r15 in this instruction. The value used is the address of the current instruction +4.

*Rm* is the index register. This contains an integer pointing to a single byte within the table. The offset within the table depends on the instruction:

TBB the value of the index

TBH twice the value of the index.

#### 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.

#### Notes

If r15 is specified for *Rn*, the value used is the address of the instruction plus 4.

#### Exceptions

Data abort.

#### Architectures

These 32-bit Thumb-2 instructions are available in T2 variants of ARMv6 and above.

There are no ARM, or 16-bit Thumb, versions of these instructions.

## 4.9 Coprocessor instructions

This section does not describe Vector Floating Point instructions (see Chapter 5 *Vector Floating Point Programming*).

It contains the following sections:

- *CDP and CDP2* on page 4-122  
Coprocessor Data oPerations
- *MCR, MCR2, MCRR, and MCRR2* on page 4-123  
Move to Coprocessor from ARM Register or Registers, possibly with coprocessor operations
- *MRC, MRC2, MRRC and MRRC2* on page 4-125  
Move to ARM Register or Registers from Coprocessor, possibly with coprocessor operations
- *LDC and STC* on page 4-127  
Transfer data between memory and Coprocessor.
- *LDC2 and STC2* on page 4-129  
Transfer data between memory and Coprocessor (alternative instructions).

### 4.9.1 CDP and CDP2

Coprocessor data operations.

#### Syntax

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

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

where:

|                                      |                                                                                                                                            |
|--------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------|
| <i>cond</i>                          | is an optional condition code (see <i>Conditional execution</i> on page 2-17).                                                             |
| <i>coproc</i>                        | is the name of the coprocessor the instruction is for. The standard name is <i>pn</i> , where <i>n</i> is an integer in the range 0 to 15. |
| <i>opcode1</i>                       | is a coprocessor-specific opcode.                                                                                                          |
| <i>CRd</i> , <i>CRn</i> , <i>CRm</i> | are coprocessor registers.                                                                                                                 |
| <i>opcode2</i>                       | is an optional coprocessor-specific opcode.                                                                                                |

#### Usage

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

#### ———— Note ————

CDP2 is always unconditional.

#### Exceptions

Undefined Instruction.

#### Architectures

The CDP ARM instruction is available in all versions of the ARM architecture.

The CDP2 ARM instruction is available in ARMv5 and above.

These 32-bit Thumb-2 instructions are available in T2 variants of ARMv6 and above.

There are no 16-bit Thumb versions of these instructions.



## 4.9.2 MCR, MCR2, MCRR, and MCRR2

Move to Coprocessor from ARM Register or Registers. Depending on the coprocessor, you might be able to specify various operations in addition.

### Syntax

`MCR{cond} coproc, opcode1, Rd, CRn, CRm{, opcode2}`

`MCR2 coproc, opcode1, Rd, CRn, CRm{, opcode2}`

`MCRR{cond} coproc, opcode1, Rd, Rn, CRm`

`MCRR2 coproc, opcode1, Rd, Rn, CRm`

where:

*cond* is an optional condition code (see *Conditional execution* on page 2-17).

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

*opcode1* is a coprocessor-specific opcode.

*Rd, Rn* are ARM source registers. Do not use r15 for *Rd* or *Rn*.

*CRn, CRm* are coprocessor registers.

*opcode2* is an optional coprocessor-specific opcode.

### Usage

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

#### Note

MCR2 and MCRR2 are always unconditional in ARM state.

### Exceptions

Undefined Instruction.

## Architectures

The MCR ARM instruction is available in all versions of the ARM architecture.

The MCR2 ARM instruction is available in ARMv5 and above.

The MCRR ARM instruction is available in ARMv6 and above, and E variants of ARMv5 excluding xP variants.

The MCRR2 ARM instruction is available in ARMv6 and above.

These 32-bit Thumb-2 instructions are available in T2 variants of ARMv6 and above.

There are no 16-bit Thumb versions of these instructions.

### 4.9.3 MRC, MRC2, MRRC and MRRC2

Move to ARM Register or Registers from Coprocessor.

Depending on the coprocessor, you might be able to specify various operations in addition.

#### Syntax

`MRC{cond} coproc, opcode1, Rd, CRn, CRm{, opcode2}`

`MRC2 coproc, opcode1, Rd, CRn, CRm{, opcode2}`

`MRRC{cond} coproc, opcode1, Rd, Rn, CRm`

`MRRC2 coproc, opcode1, Rd, Rn, CRm`

where:

*cond* is an optional condition code (see *Conditional execution* on page 2-17).

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

*opcode1* is a coprocessor-specific opcode.

*Rd, Rn* are ARM source registers. Do not use r15 for *Rd* or *Rn* in MRRC or MRRC2. In MRC and MRC2, if *Rd* is r15, only the flags field is affected.

*CRn, CRm* are coprocessor registers.

*opcode2* is an optional coprocessor-specific opcode.

#### Usage

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

#### Note

MRRC2 and MRRC2 are always unconditional.

#### Exceptions

Undefined Instruction.

## Architectures

The MRC ARM instruction is available in all versions of the ARM architecture.

The MRC2 ARM instruction is available in ARMv5 and above.

The MRRC ARM instruction is available in ARMv6 and above, and E variants of ARMv5 excluding xP variants.

The MRRC2 ARM instruction is available in ARMv6 and above.

These 32-bit Thumb-2 instructions are available in T2 variants of ARMv6 and above.

There are no 16-bit Thumb versions of these instructions.

#### 4.9.4 LDC and STC

Transfer Data between memory and Coprocessor.

##### Syntax

These instructions have three possible forms:

- zero offset
- pre-indexed offset
- post-indexed offset.

The syntax of the three forms, in the same order, are:

*op*{*L*}{*cond*} *coproc*, *CRd*, [*Rn*]

*op*{*L*}{*cond*} *coproc*, *CRd*, [*Rn*, #*{-}offset*]{!}

*op*{*L*}{*cond*} *coproc*, *CRd*, [*Rn*], #*{-}offset*

where:

*op* is either LDC or STC.

*cond* is an optional condition code (see *Conditional execution* on page 2-17).

*L* is an optional suffix specifying a long transfer.

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

*CRd* is the coprocessor register to load or save.

*Rn* is the register on which the memory address is based. If r15 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*.

##### Usage

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

## Exceptions

Undefined Instruction. Data Abort.

## Architectures

These ARM instructions are available in all versions of the ARM architecture.

These 32-bit Thumb-2 instructions are available in T2 variants of ARMv6 and above.

There are no 16-bit Thumb versions of these instructions.

## Notes

Use of PC relative addressing in the STC instruction is deprecated in ARMv6T2.

### 4.9.5 LDC2 and STC2

Transfer Data between memory and Coprocessor, alternative instructions.

#### Syntax

These instructions have three possible forms:

- zero offset
- pre-indexed offset
- post-indexed offset.

The syntax of the three forms, in the same order, are:

*op*{*L*} *coproc*, *CRd*, [*Rn*]

*op*{*L*} *coproc*, *CRd*, [*Rn*, #{-}offset]{!}

*op*{*L*} *coproc*, *CRd*, [*Rn*], #{-}offset

where:

*op* is either LDC2 or 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 *pn*, where *n* is an integer in the range 0 to 15.

*CRd* is the coprocessor register to load or save.

*Rn* is the register on which the memory address is based. If r15 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*.

## Usage

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

---

### Note

---

LDC2 and STC2 are always unconditional.

---

## Exceptions

Undefined Instruction. Data Abort.

## Architectures

These ARM instructions are available in ARMv5 and above.

These 32-bit Thumb-2 instructions are available in T2 variants of ARMv6 and above.

There are no 16-bit Thumb versions of these instructions.



## 4.10 Miscellaneous instructions

This section contains the following subsections:

- *BKPT* on page 4-132  
Breakpoint.
- *SWI* on page 4-133  
SoftWare Interrupt.
- *MRS* on page 4-134  
Move the contents of the CPSR or SPSR to a general-purpose register.
- *MSR* on page 4-135  
Load specified fields of the CPSR or SPSR with an immediate constant, or from the contents of a general-purpose register.
- *CPS* on page 4-137  
Change Processor State.
- *SMI* on page 4-139  
Secure Monitor Interrupt.
- *SETEND* on page 4-140  
Set the Endianness bit in the CPSR.
- *NOP, SEV, WFE, WFI, and YIELD* on page 4-141  
Set Event, Wait For Event, Wait for Interrupt, and Yield.
- *MAR and MRA* on page 4-143  
XScale coprocessor 0 instructions.  
Transfer between two general-purpose registers and a 40-bit internal accumulator.

### 4.10.1 BKPT

Breakpoint.

#### Syntax

BKPT *immed*

where:

- immed* is an expression evaluating to an integer in the range:
- 0-65536 (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, *immed* is ignored by the ARM hardware. However, a debugger can use it to store additional information about the breakpoint.

#### Architectures

This ARM instruction is available in ARMv5 and above.

This 16-bit Thumb instruction is available in T variants of ARMv5 and above.

There is no 32-bit Thumb-2 version of this instruction.

## 4.10.2 SWI

SoftWare Interrupt.

### Syntax

`SWI{cond} immed`

where:

*cond* is an optional condition code (see *Conditional execution* on page 2-17).

*immed* 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 16-bit Thumb instruction.

### Usage

The SWI instruction causes a SWI exception. This means that the processor mode changes to Supervisor, the CPSR is saved to the Supervisor mode SPSR, and execution branches to the SWI vector (see the *Handling Processor Exceptions* chapter in *RealView Compilation Tools v2.2 Developer Guide*).

*immed* is ignored by the processor. However, it can be retrieved by the exception handler to determine what service is being requested.

#### ————— Note —————

As part of the development of the ARM assembly language, the SWI instruction has been renamed to SVC (SuperVisor Call). In this release of RVCT, SWI instructions disassemble to SVC, with a comment to say that this was formerly SWI, for example:

```
SVC 0x42; 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 and T2 variants of ARMv6 and above.

### 4.10.3 MRS

Move the contents of the CPSR or SPSR to a general-purpose register.

#### Syntax

`MRS{cond} Rd, psr`

where:

*cond* is an optional condition code (see *Conditional execution* on page 2-17).

*Rd* is the destination register. *Rd* must not be r15.

*psr* is either CPSR or SPSR.

#### 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 programmer's 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.

#### Caution

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 does not know in what processor mode the code will be executed.

If you do this, the result is unpredictable, that is, it cannot be relied upon.

The CPSR execution state bits 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.

#### 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-2 instruction is available in T2 variants of ARMv6 and above.

There is no 16-bit Thumb version of this instruction.

## 4.10.4 MSR

Load specified fields of the CPSR or SPSR with an immediate constant, or from the contents of a general-purpose register.

### Syntax

MSR{*cond*} <*psr*>\_<*fields*>, #*constant*

MSR{*cond*} <*psr*>\_<*fields*>, *Rm*

where:

|                   |                                                                                                                                                             |
|-------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>cond</i>       | is an optional condition code (see <i>Conditional execution</i> on page 2-17).                                                                              |
| < <i>psr</i> >    | is either CPSR or SPSR.                                                                                                                                     |
| < <i>fields</i> > | specifies the <i>Program Status Register</i> (PSR) field or fields to be moved. < <i>fields</i> > can be one or more of:                                    |
| c                 | control field mask byte, PSR[7:0]                                                                                                                           |
| x                 | extension field mask byte, PSR[15:8]                                                                                                                        |
| s                 | status field mask byte, PSR[23:16]                                                                                                                          |
| f                 | flags field mask byte, PSR[31:24].                                                                                                                          |
| <i>constant</i>   | is an expression evaluating to a numeric constant. The constant must correspond to an 8-bit pattern rotated by an even number of bits within a 32-bit word. |

#### ———— **Note** ————

The #*constant* form of this instruction is not available in Thumb-2.

*Rm* is the source register.

### Usage

See *MRS* on page 4-134.

In User mode:

- Writes to unallocated, privileged or execution state bits in the CPSR are ignored. This ensures that User mode programs cannot change to a privileged mode.
- Writes to the state bits in the CPSR (defined by StateMask) are ignored in Thumb-2. Previously the behavior was unpredictable, that is, it could not be relied upon.

If you access the SPSR when in User or System mode, the result is unpredictable, that is, it cannot be relied upon.

### **Condition flags**

This instruction updates the flags explicitly if the *f* field is specified.

### **Architectures**

This ARM instruction is available in ARMv3 and above.

This 32-bit Thumb-2 instruction is available in T2 variants of ARMv6 and above.

There is no 16-bit Thumb version of this instruction.

### 4.10.5 CPS

CPS (Change Processor State) changes one or more of the mode, A, I, and F bits in the CPSR, without changing the other CPSR bits.

CPS is only allowed in privileged modes, and has no effect in User mode.

CPS cannot be conditional, and is not allowed in an IT block.

#### Syntax

*CPSeffect 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.

#### 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 when used in Thumb-2 code:

CPSIE *iflags* You cannot specify a mode change in a 16-bit Thumb instruction.

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-2 instruction is available in T2 variants of ARMv6 and above.

This 16-bit Thumb instruction is available in T variants of ARMv6 and above.

## Examples

```
CPSIE if ; enable interrupts and fast interrupts
CPSID A ; disable imprecise aborts
CPSID ai, #17 ; disable imprecise aborts and interrupts, and enter FIQ mode
CPS #16 ; enter User mode
```

## Incorrect example

```
CPSEQ #19 ; CPS cannot be conditional
```



## 4.10.6 SMI

Secure Monitor Interrupt.

For details see the *ARM Architecture Reference Manual Security Extensions supplement*.

### Syntax

`SMI{cond} #immed_16`

where:

*cond* is an optional condition code (see *Conditional execution* on page 2-17).

*immed\_16* is a 16-bit immediate value. This is ignored by the ARM processor, but can be used by the SMI exception handler to determine what service is being requested.

### Architectures

This ARM instruction is available in Z variants of ARMv6 and above.

This 32-bit Thumb-2 instruction is available in Z variants of ARMv6T2 and above.

There is no 16-bit Thumb version of this instruction.

### 4.10.7 SETEND

Set the endianness bit in the CPSR, without affecting any other bits in the CPSR.

SETEND cannot be conditional, and is not allowed in an IT block.

#### Syntax

SETEND *specifier*

where:

|                  |                |
|------------------|----------------|
| <i>specifier</i> | 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.

#### Architectures

This ARM instruction is available in ARMv6 and above.

This 16-bit Thumb instruction is available in T variants of ARMv6 and above.

There is no 32-bit Thumb-2 version of this instruction.

#### 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

```

#### 4.10.8 NOP, SEV, WFE, WFI, and YIELD

No Operation, Set Event, Wait For Event, Wait for Interrupt, and Yield.

##### Syntax

`NOP{cond}`

`SEV{cond}`

`WFE{cond}`

`WFI{cond}`

`YIELD{cond}`

where:

*cond* is an optional condition code (see *Conditional execution* on page 2-17).

##### Usage

These are hint instructions. It is optional whether they are implemented or not. If any one of them is not implemented, it behaves as a NOP.

##### **NOP**

NOP does nothing. If NOP is not implemented as a specific instruction on your target architecture, the assembler 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.

##### **SEV**

SEV causes an event to be signaled to all cores within a multiprocessor system. If SEV is implemented, WFE must also be implemented.

##### **WFE**

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.

### **WFI**

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.

### **YIELD**

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**

These ARM instructions are available in ARMv6T2 and above, and K variants of ARMv6.

These 32-bit Thumb-2 instructions are available in T2 variants of ARMv6 and above, and K variants of ARMv6.

These 16-bit Thumb instructions are available in ARMv6T2 and above, and K variants of ARMv6.

### 4.10.9 MAR and MRA

XScale coprocessor 0 instructions.

Transfer between two general-purpose registers and a 40-bit internal accumulator.

#### Syntax

`MAR{cond} Acc, RdLo, RdHi`

`MRA{cond} RdLo, RdHi, Acc`

where:

*cond* is an optional condition code (see *Conditional execution* on page 2-17).

*Acc* is the internal accumulator. The standard name is *accx*, where *x* is an integer in the range 0 to *n*. The value of *n* depends on the processor. It is 0 for current processors.

*RdLo*, *RdHi* are general-purpose registers.

#### Usage

The MAR instruction copies the contents of *RdLo* to bits[31:0] of *Acc*, and the least significant byte of *RdHi* to bits[39:32] of *Acc*.

The MRA instruction:

- copies bits[31:0] of *Acc* to *RdLo*
- copies bits[39:32] of *Acc* to *RdHi* bits[7:0]
- sign extends the value by copying bit[39] of *Acc* to bits[31:8] of *RdHi*.

#### Architectures

These ARM instructions are only available in XScale processors.

There are no Thumb or Thumb-2 versions of these instructions.

#### Examples

```
MAR acc0, r0, r1
MRA r4, r5, acc0
MARNE acc0, r9, r2
MRAGT r4, r8, acc0
```

## 4.11 Pseudo-instructions

The ARM assembler supports a number of pseudo-instructions that are translated into the appropriate combination of ARM, Thumb, or Thumb-2 instructions at assembly time.

The pseudo-instructions are described in the following sections:

- *ADR pseudo-instruction* on page 4-145  
Load a program-relative or register-relative address (short range, position independent)
- *ADRL pseudo-instruction* on page 4-147  
Load a program-relative or register-relative address into a register (medium range, position independent)
- *MOV32 pseudo-instruction* on page 4-149  
Load a register with a 32-bit constant value or an address (unlimited range, but not position independent). Available for ARMv6T2 and above only.
- *LDR pseudo-instruction* on page 4-151  
Load a register with a 32-bit constant value or an address (unlimited range, but not position independent). Available for all ARM architectures.

### 4.11.1 ADR pseudo-instruction

Load a program-relative or register-relative address into a register.

#### Syntax

`ADR{cond}{.w} register, label`

where:

|                 |                                                                                                                                                         |
|-----------------|---------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>cond</i>     | is an optional condition code (see <i>Conditional execution</i> on page 2-17).                                                                          |
| .w              | is an optional instruction width specifier. See <i>ADR in Thumb-2</i> on page 4-146 for details.                                                        |
| <i>register</i> | is the register to load.                                                                                                                                |
| <i>label</i>    | is a program-relative or register-relative expression. See <i>Register-relative and program-relative expressions</i> on page 3-28 for more information. |

#### Usage

ADR always assembles to one instruction. The assembler attempts to produce a single ADD or SUB instruction to load the address. If the address cannot be constructed in a single instruction, an error is generated and the assembly fails.

ADR produces position-independent code, because the address is program-relative or register-relative.

Use the ADRL pseudo-instruction to assemble a wider range of effective addresses.

If *label* is program-relative, it must evaluate to an address in the same assembler area as the ADR pseudo-instruction, see *AREA* on page 6-64.

#### Range

The available range depends on the instruction set in use:

|                       |                                                                                                     |
|-----------------------|-----------------------------------------------------------------------------------------------------|
| <b>ARM</b>            | ±255 bytes to a byte or halfword-aligned address.<br>±1020 bytes to a word-aligned address.         |
| <b>32-bit Thumb-2</b> | ±4095 bytes to a byte, halfword, or word-aligned address.                                           |
| <b>16-bit Thumb</b>   | 0 to 1020 bytes. <i>label</i> must be word-aligned. You can use the ALIGN directive to ensure this. |

The given range is relative to a point two words after the address of the current instruction. In ARM and Thumb-2, more distant addresses can be in range if the alignment is 16-bytes or more relative to this point.

### **ADR in Thumb-2**

You can use the `.W` width specifier to force ADR to generate a 32-bit instruction in Thumb-2 code.

ADR.W always generates a 32-bit instruction, even if the address could be generated in a 16-bit ADD or SUB.

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-2 ADD instruction.



### 4.11.2 ADRL pseudo-instruction

Load a program-relative or register-relative address into a register. It is similar to the ADR pseudo-instruction. ADRL can load a wider range of addresses than ADR because it generates two data processing instructions.

---

#### Note

---

ADRL is not available when assembling 16-bit Thumb instructions. Use it only in ARM or Thumb-2 code.

---

#### Syntax

`ADRL{cond} register, label`

where:

- cond* is an optional condition code (see *Conditional execution* on page 2-17).
- register* is the register to load.
- label* is a program-relative or register-relative expression. See *Register-relative and program-relative expressions* on page 3-28 for more information.

#### 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. See *LDR pseudo-instruction* on page 4-151 for information on loading a wider range of addresses (see also *Loading constants into registers* on page 2-23).

ADRL produces position-independent code, because the address is program-relative or register-relative.

If *label* is program-relative, it must evaluate to an address in the same assembler area as the ADRL pseudo-instruction, see *AREA* on page 6-64.

## Range

The available range depends on the instruction set in use:

|                       |                                                                      |
|-----------------------|----------------------------------------------------------------------|
| <b>ARM</b>            | $\pm 64\text{KB}$ to a byte or halfword-aligned address.             |
|                       | $\pm 256\text{KB}$ bytes to a word-aligned address.                  |
| <b>32-bit Thumb-2</b> | $\pm 1\text{MB}$ bytes to a byte, halfword, or word-aligned address. |
| <b>16-bit Thumb</b>   | ADRL is not available.                                               |

The given range is relative to a point two words after the address of the current instruction. In ARM and Thumb-2, more distant addresses can be in range if the alignment is 16-bytes or more relative to this point.

### 4.11.3 MOV32 pseudo-instruction

Load a register with either:

- a 32-bit constant value
- any address.

MOV32 always generates two 32-bit instructions.

If MOV32 is used to load an address, the resulting code is not position-independent.

#### Syntax

MOV32{*cond*} *register*, *expr*

where:

*cond* is an optional condition code (see *Conditional execution* on page 2-17).

*register* is the register to be loaded.

*expr* can be any one of the following:

*symbol* A label in this or another program area.

*constant* Any 32-bit constant.

*symbol* + *constant* A label plus a 32-bit constant.

The assembler generates a MOV, MOVT pair of instructions. This allows you to load any 32-bit constant, or to access the whole address space.

If *symbol* is external, the assembler places a linker relocation directive in the object file. The linker generates the address at link time,

#### Usage

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 program-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.

## **Architectures**

This ARM pseudo-instruction is available in all versions of the ARM architecture.

This 32-bit Thumb-2 pseudo-instruction is available in T2 variants of ARMv6 and above.

There is no 16-bit Thumb version of this pseudo-instruction.

#### 4.11.4 LDR pseudo-instruction

Load a register with either:

- a 32-bit constant value
- an address.

---

##### Note

---

This section describes the LDR *pseudo*-instruction only. See *Memory access instructions* on page 4-7 for information on the LDR *instruction*.

Also, see *Loading with LDR Rd, =const* on page 2-28, for information on loading constants with the LDR pseudo-instruction.

---

#### Syntax

LDR{*cond*}{.w} *register*,=[*expr* | *label-expr*]

where:

- |                   |                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
|-------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>cond</i>       | is an optional condition code (see <i>Conditional execution</i> on page 2-17).                                                                                                                                                                                                                                                                                                                                                                   |
| .w                | is an optional instruction width specifier. See <i>LDR in Thumb-2</i> on page 4-152 for details.                                                                                                                                                                                                                                                                                                                                                 |
| <i>register</i>   | is the register to be loaded.                                                                                                                                                                                                                                                                                                                                                                                                                    |
| <i>expr</i>       | <p>evaluates to a numeric constant:</p> <ul style="list-style-type: none"> <li>• The assembler generates a MOV or MVN instruction, if the value of <i>expr</i> is within range.</li> <li>• If the value of <i>expr</i> is <i>not</i> within range of a MOV or MVN instruction, the assembler places the constant in a literal pool and generates a program-relative LDR instruction that reads the constant from the literal pool.</li> </ul>    |
| <i>label-expr</i> | <p>is a program-relative or external expression. The assembler places the value of <i>label-expr</i> in a literal pool and generates a program-relative LDR instruction that loads the value from the literal pool.</p> <p>If <i>label-expr</i> 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.</p> |

## Usage

The main purposes of the LDR pseudo-instruction are:

- To generate literal constants when an immediate value cannot be moved into a register because it is out of range of the MOV and MVN instructions
- To load a program-relative or external address into a register. The address remains valid regardless of where the linker places the ELF section containing the LDR.

### ———— Note ————

An address loaded in this way is fixed at link time, so the code is *not* position-independent.

The offset from the PC to the value in the literal pool must be less than  $\pm 4\text{KB}$  (ARM, 32-bit Thumb-2) or in the range 0 to +1KB (Thumb, 16-bit Thumb-2). You are responsible for ensuring that there is a literal pool within range. See *LTORG* on page 6-15 for more information.

See *Loading constants into registers* on page 2-23 for a more detailed explanation of how to use LDR, and for more information on MOV and MVN.

## Architectures

This ARM pseudo-instruction is available in all versions of the ARM architecture.

For 32-bit Thumb-2, see *LDR in Thumb-2*.

This 16-bit Thumb pseudo-instruction is available in all T variants of the ARM architecture.

### LDR in Thumb-2

You can use the `.W` width specifier to force LDR to generate a 32-bit instruction in Thumb-2 code.

LDR.W always generates a 32-bit instruction, even if the constant could be loaded in a 16-bit MOV, or there is a literal pool within reach of a 16-bit pc-relative load.

LDR without `.W` always generates a 16-bit instruction in Thumb code, even if that results in a 16-bit pc-relative load for a constant that could be generated in a 32-bit MOV or MVN instruction.

You can also use MOV32 (see *MOV32 pseudo-instruction* on page 4-149). This can place any 32-bit value in a register without a load operation, using two instructions.

**Examples**

```

LDR r3,=0xff0 ; loads 0xff0 into r3
 ; => MOV 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

```





# Chapter 5

## Vector Floating Point Programming

This chapter provides reference information about programming the *Vector Floating Point* coprocessor in assembly language. It contains the following sections:

- *The Vector Floating Point coprocessor* on page 5-4
- *Floating-point registers* on page 5-5
- *Vector and scalar operations* on page 5-7
- *VFP and condition codes* on page 5-8
- *VFP system registers* on page 5-10
- *Flush-to-zero mode* on page 5-13
- *VFP instructions* on page 5-15
- *VFP pseudo-instruction* on page 5-36
- *VFP directives and vector notation* on page 5-37.

See Table 5-1 on page 5-2 to locate descriptions of individual instructions.

Table 5-1 Location of VFP instructions

| Mnemonic     | Brief description                                           | Page      | Operation | Architecture |
|--------------|-------------------------------------------------------------|-----------|-----------|--------------|
| FABS         | Absolute value                                              | page 5-16 | Vector    | All          |
| FADD         | Add                                                         | page 5-17 | Vector    | All          |
| FCMP         | Compare                                                     | page 5-18 | Scalar    | All          |
| FCPY         | Copy                                                        | page 5-16 | Vector    | All          |
| FCVTDS       | Convert single-precision to double-precision                | page 5-19 | Scalar    | All          |
| FCVTSD       | Convert double-precision to single-precision                | page 5-20 | Scalar    | All          |
| FDIV         | Divide                                                      | page 5-21 | Vector    | All          |
| FLD          | Load (see also <i>FLD pseudo-instruction</i> on page 5-36)  | page 5-22 | Scalar    | All          |
| FLDM         | Load multiple                                               | page 5-24 | -         | All          |
| FMAC         | Multiply-accumulate                                         | page 5-26 | Scalar    | All          |
| FMDHR, FMDLR | Transfer from one ARM® register to half of double-precision | page 5-28 | Scalar    | All          |
| FMDRR        | Transfer from two ARM registers to double-precision         | page 5-27 | Scalar    | VFPv2        |
| FMRDH, FMRDL | Transfer from half of double-precision to ARM register      | page 5-28 | Scalar    | All          |
| FMRRD        | Transfer from double-precision to two ARM registers         | page 5-27 | Scalar    | VFPv2        |
| FMRRS        | Transfer between two ARM registers and two single-precision | page 5-30 | Scalar    | VFPv2        |
| FMRS         | Transfer from single-precision to ARM register              | page 5-29 | Scalar    | All          |
| FMRX         | Transfer from VFP system register to ARM register           | page 5-31 | -         | All          |
| FMSC         | Multiply-subtract                                           | page 5-26 | Vector    | All          |
| FMSR         | Transfer from ARM register to single-precision              | page 5-29 | Scalar    | All          |
| FMSRR        | Transfer between two ARM registers and two single-precision | page 5-30 | Scalar    | VFPv2        |
| FMSTAT       | Transfer VFP status flags to ARM CPSR status flags          | page 5-31 | -         | All          |
| FMUL         | Multiply                                                    | page 5-32 | Vector    | All          |
| FMXR         | Transfer from ARM register to VFP system register           | page 5-31 | -         | All          |

**Table 5-1 Location of VFP instructions (continued)**

| <b>Mnemonic</b> | <b>Brief description</b>                             | <b>Page</b> | <b>Operation</b> | <b>Architecture</b> |
|-----------------|------------------------------------------------------|-------------|------------------|---------------------|
| FNEG            | Negate                                               | page 5-16   | Vector           | All                 |
| FNMAC           | Negate-multiply-accumulate                           | page 5-26   | Vector           | All                 |
| FNMSC           | Negate-multiply-subtract                             | page 5-26   | Vector           | All                 |
| FMUL            | Negate-multiply                                      | page 5-32   | Vector           | All                 |
| FSITO           | Convert signed integer to floating-point             | page 5-33   | Scalar           | All                 |
| FSQRT           | Square Root                                          | page 5-34   | Vector           | All                 |
| FST             | Store                                                | page 5-22   | Scalar           | All                 |
| FSTM            | Store multiple                                       | page 5-24   | -                | All                 |
| FSUB            | Subtract                                             | page 5-17   | Vector           | All                 |
| FTOSI, FTQSI    | Convert floating-point to signed or unsigned integer | page 5-35   | Scalar           | All                 |
| FUITO           | Convert unsigned integer to floating-point           | page 5-33   | Scalar           | All                 |

## 5.1 The Vector Floating Point coprocessor

The *Vector Floating Point* (VFP) coprocessor, 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 in this chapter. There is a summary of the standard in the floating-point chapter in *RealView Compilation Tools v2.2 Compiler and Libraries Guide*.

Short vectors of up to eight single-precision or four double-precision numbers are handled particularly efficiently. Most arithmetic instructions can be used on these vectors, enabling single-instruction, multiple-data (SIMD) parallelism. In addition, the floating-point load and store instructions have multiple register forms, enabling vectors to be transferred to and from memory efficiently.

For more details of the VFP coprocessor, see *ARM Architecture Reference Manual*.

### 5.1.1 VFP architectures

There are two versions of the VFP architecture. VFPv2 has all the instructions that VFPv1 has, and four additional instructions.

The additional instructions enable you to transfer two 32-bit words between ARM registers and VFP registers with one instruction.

———— **Note** ————

Support for VFPv1 is obsolete. The default is VFPv2.

————

## 5.2 Floating-point registers

The VFP coprocessor has 32 single-precision registers, s0 to s31. Each register can contain either a single-precision floating-point value, or a 32-bit integer.

These 32 registers are also treated as 16 double-precision registers, d0 to d15.  $d_n$  occupies the same hardware as  $s(2n)$  and  $s(2n+1)$ .

You can use:

- some registers for single-precision values at the same time as you are using others for double-precision values
- the same registers for single-precision values and double-precision values at different times.

Do not attempt to use corresponding single-precision and double-precision registers at the same time. No damage is caused but the results are meaningless.

### 5.2.1 Register banks

The VFP registers are arranged as four banks of:

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

See Figure 5-1 for further clarification.

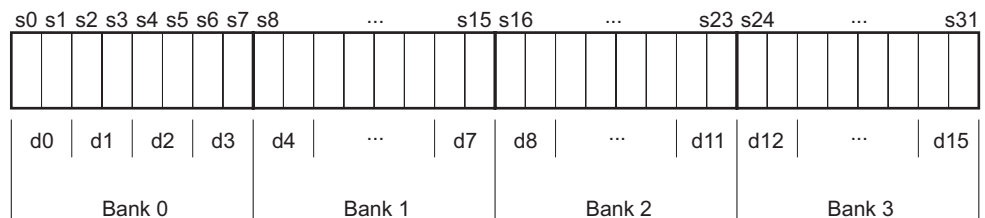


**Figure 5-1 VFP register banks**

### 5.2.2 Vectors

A vector 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 (see *FPSCR, the floating-point status and control register* on page 5-10).

A vector can start from any register. The first register used by a vector is specified in the register fields in the individual instructions.

#### Vector wrap-around

If the 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.

#### Vector stride

Vectors can occupy consecutive registers, as in the examples above, or they can occupy alternate registers. This is controlled by the STRIDE bits in the FPSCR (see *FPSCR, the floating-point status and control register* on page 5-10). 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}.

#### Restriction on vector length

A vector cannot use the same register twice. Enabling for vector wrap-around, this means that 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.

## 5.3 Vector and scalar operations

You can use VFP arithmetic instructions to operate:

- on scalars
- on vectors
- on scalars and vectors together.

Use the LEN bits in the FPSCR to control the length of vectors (see *FPSCR, the floating-point status and control register* on page 5-10).

When LEN is 1 all operations are scalar.

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

When LEN is greater than 1, the behavior of arithmetic operations depends on which register bank the destination and operand registers are in (see *Register banks* on page 5-5).

Given instructions of the following general forms:

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

the behavior is as follows:

- If *Fd* is in the first bank of registers, s0 to s7 or d0 to d3, the operation is scalar.
- If the *Fm* is in the first bank of registers, but *Fd* is not, the operation is mixed.
- If neither *Fd* nor *Fm* are in the first bank of registers, the operation is vector.

#### Scalar operations

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

#### 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*.

#### Mixed scalar and vector operations

For single-operand instructions, *Op* acts on the single value in *Fm*. LEN copies of the result are placed in the vector starting at *Fd*.

For multiple-operand instructions, *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*.

## 5.4 VFP and condition codes

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

In Thumb® state on a Thumb-2 processor, you can use an IT instruction to set condition codes on up to four following VFP instructions. See *IT* on page 4-70 for details.

The only VFP instruction that can be used to update the status flags is FCMPL. It does not update the flags in the CPSR directly, but updates a separate set of flags in the FPSCR (see *FPSCR, the floating-point status and control register* on page 5-10).

---

### Note

---

To use these flags to control conditional instructions, including conditional VFP instructions, you must first copy them into the CPSR using an FMSTAT instruction (see *FMRX, FMXR, and FMSTAT* on page 5-31).

---

Following an FCMPL instruction, the precise meanings of the flags are different from their meanings following an ARM 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 meanings of the condition code mnemonics are shown in Table 5-2 on page 5-9.



**Table 5-2 Condition codes**

| <b>Mnemonic</b> | <b>Meaning after ARM data processing instruction</b> | <b>Meaning after VFP FCMP instruction</b> |
|-----------------|------------------------------------------------------|-------------------------------------------|
| EQ              | Equal                                                | Equal                                     |
| NE              | Not equal                                            | Not equal, or unordered                   |
| CS / HS         | Carry set / Unsigned higher or same                  | Greater than or equal, or unordered       |
| CC / LO         | Carry clear / 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 flags in the CPSR determines the meaning of condition codes.

## 5.5 VFP system registers

Three VFP system registers are accessible to you in all implementations of VFP:

- *FPSCR*, the floating-point status and control register
- *FPEXC*, the floating-point exception register on page 5-12
- *FPSID*, the floating-point system ID register on page 5-12
- *Modifying individual bits of a VFP system register* on page 5-12.

A particular implementation of VFP can have additional registers (see the technical reference manual for the VFP coprocessor you are using).

### 5.5.1 FPSCR, the floating-point status and control register

The FPSCR contains all the user-level VFP status and control bits:

**bits[31:28]** Are the N, Z, C, and V flags. These are the VFP status flags. They cannot be used to control conditional execution until they have been copied into the status flags in the CPSR (see *VFP and condition codes* on page 5-8).

**bit[24]** Is the flush-to-zero mode control bit:

**0** Flush-to-zero mode is disabled.

**1** Flush-to-zero mode is enabled.

Flush-to-zero mode can provide greater performance, depending on your hardware and software, at the expense of loss of range (see *Flush-to-zero mode* on page 5-13).

———— **Note** —————

Flush-to-zero mode must not be used when IEEE 754 compatibility is a requirement.

**bits[23:22]** Control rounding mode as follows:

**0b00** *Round to Nearest (RN)* mode.

**0b01** *Round towards Plus infinity (RP)* mode.

**0b10** *Round towards Minus infinity (RM)* mode.

**0b11** *Round towards Zero (RZ)* mode.

**bits[21:20]** STRIDE is the distance between successive values in a vector (see *Vectors* on page 5-6). Stride is controlled as follows:

**0b00** stride = 1

**0b11** stride = 2.

**bits[18:16]** LEN is the number of registers used by each vector (see *Vectors* on page 5-6). It is 1 + the value of bits[18:16]:

**0b000**      LEN = 1

.

.

**0b111**      LEN = 8.

**bits[12:8]** Are the exception trap enable bits:

**IXE**      inexact exception enable

**UFE**      underflow exception enable

**OFE**      overflow exception enable

**DZE**      division by zero exception enable

**IOE**      invalid operation exception enable.

This manual does not cover the use of floating-point exception trapping. For information see the technical reference manual for the VFP coprocessor you are using.

**bits[4:0]** Are the cumulative exception bits:

**IXC**      inexact exception

**UFC**      underflow exception

**OFC**      overflow exception

**DZC**      division by zero exception

**IOC**      invalid operation exception.

Cumulative exception bits are set when the corresponding exception occurs. They remain set until you clear them by writing directly to the FPSCR.

**all other bits** Are unused in the basic VFP specification. They can be used in particular implementations (see the technical reference manual for the VFP coprocessor you are using). Do not modify these bits except in accordance with any use in a particular implementation.

To change some bits without affecting other bits, use a read-modify-write procedure (see *Modifying individual bits of a VFP system register* on page 5-12).

### 5.5.2 FPEXC, the floating-point exception register

You can only access the FPEXC in privileged modes. It contains the following bits:

- bit[31]** Is the EX bit. You can read it in all VFP implementations. In some implementations you might also be able to write to it.
- If the value is 0, the only significant state in the VFP system is the contents of the general-purpose registers plus FPSCR and FPEXC.
- If the value is 1, you require implementation-specific information to save state (see the technical reference manual for the VFP coprocessor you are using).
- bit[30]** Is the EN bit. You can read and write it in all VFP implementations.
- If the value is 1, the VFP coprocessor is enabled and operates normally.
- If the value is 0, the VFP coprocessor is disabled. When the coprocessor is disabled, you can read or write the FPSID or FPEXC registers, but other VFP instructions are treated as Undefined Instructions.
- bits[29:0]** Might be used by particular implementations of VFP. You can use all the VFP functions described in this chapter without accessing these bits.
- You must not alter these bits except in accordance with their use in a particular implementation (see the technical reference manual for the VFP coprocessor you are using).

To change some bits without affecting other bits, use a read-modify-write procedure (see *Modifying individual bits of a VFP system register*).

### 5.5.3 FPSID, the floating-point system ID register

The FPSID is a read-only register. You can read it to find out which implementation of the VFP architecture your program is running on.

### 5.5.4 Modifying individual bits of a VFP system register

To change some bits of a VFP system register without affecting other bits, use a read-modify-write procedure similar to the following example:

```

FMRX r10,FPSCR ; copy FPSCR into r10
BIC r10,r10,#0x00370000 ; clears STRIDE and LEN
ORR r10,r10,#0x00030000 ; sets STRIDE = 1, LEN = 4
FMXR FPSCR,r10 ; copy r10 back into FPSCR

```

See *FMRX*, *FMXR*, and *FMSTAT* on page 5-31.

## 5.6 Flush-to-zero mode

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.

Flush-to-zero mode replaces denormalized numbers with +0. This does not comply with IEEE 754 arithmetic, but in some circumstances can improve performance considerably.

### 5.6.1 When to use flush-to-zero mode

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

### 5.6.2 The effects of using flush-to-zero mode

With certain exceptions (see *Operations not affected by flush-to-zero mode* on page 5-14), 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.

An inexact exception occurs whenever a denormalized number is used as an operand, or a result is flushed to zero. Underflow exceptions do not occur in flush-to-zero mode.

### 5.6.3 Operations not affected by flush-to-zero mode

The following operations can be carried out on denormalized numbers even in flush-to-zero mode, without flushing the results to zero:

- Copy, absolute value, and negate (see *FABS*, *FCPY*, and *FNEG* on page 5-16).
- Load and store (see *FLD* and *FST* on page 5-22).
- Load multiple and store multiple (see *FLDM* and *FSTM* on page 5-24).
- Transfer between floating-point registers and ARM general-purpose registers (see *FMDRR* and *FMRRD* on page 5-27 and *FMRRS* and *FMSRR* on page 5-30).

## 5.7 VFP instructions

This section contains the following subsections:

- *FABS, FCPY, and FNEG* on page 5-16  
Floating-point absolute value, copy, and negate.
- *FADD and FSUB* on page 5-17  
Floating-point add and subtract.
- *FCMP* on page 5-18  
Floating-point compare.
- *FCVTDS* on page 5-19  
Convert single-precision floating-point to double-precision.
- *FCVTSD* on page 5-20  
Convert double-precision floating-point to single-precision.
- *FDIV* on page 5-21  
Floating-point divide.
- *FLD and FST* on page 5-22  
Floating-point load and store.
- *FLDM and FSTM* on page 5-24  
Floating-point load multiple and store multiple.
- *FMAC, FNMAC, FMSC, and FNMSC* on page 5-26  
Floating-point multiply accumulate instructions.
- *FMDRR and FMRRD* on page 5-27  
Transfer contents between ARM registers and a double-precision floating-point register.
- *FMRRS and FMSRR* on page 5-30  
Transfer contents between a single-precision floating-point register and an ARM register.
- *FMRX, FMXR, and FMSTAT* on page 5-31  
Transfer contents between an ARM register and a VFP system register.
- *FMUL and FNMUL* on page 5-32  
Floating-point multiply and negate-multiply.
- *FSITO and FUITO* on page 5-33  
Convert signed integer to floating-point and unsigned integer to floating-point.
- *FSQRT* on page 5-34  
Floating-point square root.
- *FTOSI and FTOUI* on page 5-35  
Convert floating-point to signed integer and floating-point to unsigned integer.

### 5.7.1 FABS, FCPY, and FNEG

Floating-point copy, absolute value, and negate.

These instructions can be scalar, vector, or mixed (see *Vector and scalar operations* on page 5-7).

#### Syntax

`<op><precision>{<cond>} Fd, Fm`

where:

`<op>` must be one of FCPY, FABS, or FNEG.

`<precision>` must be either S for single-precision, or D for double-precision.

`cond` is an optional condition code (see *VFP and condition codes* on page 5-8).

`Fd` is the VFP register for the result.

`Fm` is the VFP register holding the operand.

The precision of `Fd` and `Fm` must match the precision specified in `<precision>`.

#### Usage

The FCPY instruction copies the contents of `Fm` into `Fd`.

The FABS instruction takes the contents of `Fm`, clears the sign bit, and places the result in `Fd`. This gives the absolute value.

The FNEG instruction takes the contents of `Fm`, changes the sign bit, and places the result in `Fd`. This gives the negation of the value.

If the operand is a NaN, the sign bit is determined in each case as above, but no exception is produced.

#### Exceptions

None of these instructions can produce any exceptions.

#### Examples

```
FABSD d3, d5
FNEGSMI s15, s15
```



## 5.7.2 FADD and FSUB

Floating-point add and subtract.

FADD and FSUB can be scalar, vector, or mixed (see *Vector and scalar operations* on page 5-7).

### Syntax

FADD<precision>{cond} Fd, Fn, Fm

FSUB<precision>{cond} Fd, Fn, Fm

where:

<precision> must be either S for single-precision, or D for double-precision.

cond is an optional condition code (see *VFP and condition codes* on page 5-8).

Fd is the VFP register for the result.

Fn is the VFP register holding the first operand.

Fm is the VFP register holding the second operand.

The precision of Fd, Fn and Fm must match the precision specified in <precision>.

### Usage

The FADD instruction adds the values in Fn and Fm and places the result in Fd.

The FSUB instruction subtracts the value in Fm from the value in Fn and places the result in Fd.

### Exceptions

FADD and FSUB instructions can produce Invalid Operation, Overflow, or Inexact exceptions.

### Examples

|         |             |
|---------|-------------|
| FSUBSEQ | s2, s4, s17 |
| FADDDGT | d4, d0, d12 |
| FSUBD   | d0, d0, d12 |

### 5.7.3 FCMP

Floating-point compare.

FCMP is always scalar.

#### Syntax

FCMP{E}<precision>{cond} Fd, Fm

FCMP{E}Z<precision>{cond} Fd

where:

**E** is an optional parameter. If E is present, an exception is raised if either operand is any kind of NaN. Otherwise, an exception is raised only if either operand is a signaling NaN.

**Z** is a parameter specifying comparison with zero.

**<precision>** must be either S for single-precision, or D for double-precision.

**cond** is an optional condition code (see *VFP and condition codes* on page 5-8).

**Fd** is the VFP register holding the first operand.

**Fm** is the VFP register holding the second operand. Omit *Fm* for a compare with zero instruction.

The precision of *Fd* and *Fm* must match the precision specified in *<precision>*.

#### Usage

The FCMP instruction subtracts the value in *Fm* from the value in *Fd* and sets the VFP condition flags on the result (see *VFP and condition codes* on page 5-8).

#### Exceptions

FCMP instructions can produce Invalid Operation exceptions.

#### Examples

```
FCMPS s3, s0
FCMPEDNE d5, d13
FCMPZSEQ s2
```

### 5.7.4 FCVTDS

Convert single-precision floating-point to double-precision.

FCVTDS is always scalar.

#### Syntax

FCVTDS{*cond*} *Dd*, *Sm*

where:

*cond* is an optional condition code (see *VFP and condition codes* on page 5-8).  
*Dd* is a double-precision VFP register for the result.  
*Sm* is a single-precision VFP register holding the operand.

#### Usage

The FCVTDS instruction converts the single-precision value in *Sm* to double-precision and places the result in *Dd*.

#### Exceptions

FCVTDS instructions can produce Invalid Operation exceptions.

#### Examples

```
FCVTDS d5, s7
FCVTDSGT d0, s4
```

### 5.7.5 FCVTSD

Convert double-precision floating-point to single-precision.

FCVTSD is always scalar.

#### Syntax

FCVTSD{*cond*} *Sd*, *Dm*

where:

*cond* is an optional condition code (see *VFP and condition codes* on page 5-8).

*Sd* is a single-precision VFP register for the result.

*Dm* is a double-precision VFP register holding the operand.

#### Usage

The FCVTSD instruction converts the double-precision value in *Dm* to single-precision and places the result in *Sd*.

#### Exceptions

FCVTSD instructions can produce Invalid Operation, Overflow, Underflow, or Inexact exceptions.

#### Examples

```
FCVTSD s3, d14
FCVTSDMI s0, d1
```

## 5.7.6 FDIV

Floating-point divide.

FDIV can be scalar, vector, or mixed (see *Vector and scalar operations* on page 5-7).

### Syntax

FDIV<precision>{cond} Fd, Fn, Fm

where:

<precision> must be either S for single-precision, or D for double-precision.

cond is an optional condition code (see *VFP and condition codes* on page 5-8).

Fd is the VFP register for the result.

Fn is the VFP register holding the first operand.

Fm is the VFP register holding the second operand.

The precision of Fd, Fn and Fm must match the precision specified in <precision>.

### Usage

The FDIV instruction divides the value in Fn by the value in Fm and places the result in Fd.

### Exceptions

FDIV operations can produce Division by Zero, Invalid Operation, Overflow, Underflow, or Inexact exceptions.

### Examples

|         |              |
|---------|--------------|
| FDIVS   | s8, s0, s12  |
| FDIVSNE | s2, s27, s28 |
| FDIVD   | d10, d2, d10 |

### 5.7.7 FLD and FST

Floating-point load and store.

#### Syntax

FLD<precision>{cond} Fd, [Rn{, #offset}]

FST<precision>{cond} Fd, [Rn{, #offset}]

FLD<precision>{cond} Fd, label

FST<precision>{cond} Fd, label

where:

<precision> must be either S for single-precision, or D for double-precision.

cond is an optional condition code (see *VFP and condition codes* on page 5-8).

Fd is the VFP register to be loaded or saved. The precision of Fd must match the precision specified in <precision>.

Rn is the ARM register holding the base address for the transfer.

offset is an optional numeric expression. It must evaluate to a numeric constant 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 program-relative expression. See *Register-relative and program-relative expressions* on page 3-28 for more information.

label must be within  $\pm 1\text{KB}$  of the current instruction.

#### Usage

The FLD instruction loads a floating-point register from memory. The FST instruction saves the contents of a floating-point register to memory.

One word is transferred if <precision> is S. Two words are transferred if <precision> is D.

There is also an FLD pseudo-instruction (see *FLD pseudo-instruction* on page 5-36).

**Examples**

```
FLDD d5, [r7, #-12]
FLDSNE s3, [r2, #72+count]
FSTS s2, [r5]
FLDD d2, [r15, #addr-{PC}]
FLDS s9, fpconst
```

### 5.7.8 FLDM and FSTM

Floating-point load multiple and store multiple.

#### Syntax

FLDM<addressmode><precision>{cond} Rn,{!} VFPregisters

FSTM<addressmode><precision>{cond} Rn,{!} VFPregisters

where:

|                                                                                                                                                                |                                                                                                                                                                                      |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <addressmode>                                                                                                                                                  | must be one of:                                                                                                                                                                      |
| IA                                                                                                                                                             | meaning Increment address After each transfer.                                                                                                                                       |
| DB                                                                                                                                                             | meaning Decrement address Before each transfer.                                                                                                                                      |
| EA                                                                                                                                                             | meaning Empty Ascending stack operation. This is the same as DB for loads, and the same as IA for saves.                                                                             |
| FD                                                                                                                                                             | meaning Full Descending stack operation. This is the same as IA for loads, and the same as DB for saves.                                                                             |
| <precision>                                                                                                                                                    | must be one of:                                                                                                                                                                      |
| S                                                                                                                                                              | for single-precision.                                                                                                                                                                |
| D                                                                                                                                                              | for double-precision.                                                                                                                                                                |
| X                                                                                                                                                              | for unspecified precision.                                                                                                                                                           |
| cond                                                                                                                                                           | is an optional condition code (see <i>VFP and condition codes</i> on page 5-8).                                                                                                      |
| 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.                                                                                                   |
| <p style="text-align: center;">———— <b>Note</b> ————</p> <p>If ! is not specified, &lt;addressmode&gt; must be IA.</p> <p style="text-align: center;">————</p> |                                                                                                                                                                                      |
| VFPregisters                                                                                                                                                   | is a list of consecutive floating-point 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. |



## Usage

The FLDM instruction loads several consecutive floating-point registers from memory.

The FSTM instruction saves the contents of several consecutive floating-point registers to memory.

If *<precision>* is specified as D, *VFPregisters* must be a list of double-precision registers, and two words are transferred for each register in the list.

If *<precision>* is specified as S, *VFPregisters* must be a list of single-precision registers, and one word is transferred for each register in the list.

## Unspecified precision

If *<precision>* is specified as X, *VFPregisters* must be specified as double-precision registers. However, any or all of the specified double-precision registers can actually contain two single-precision values or integers.

The number of words transferred might be  $2n$  or  $(2n + 1)$ , where  $n$  is the number of double-precision registers in the list. This is implementation dependent. However, if writeback is specified,  $Rn$  is always adjusted by  $(2n+1)$  words.

You must only use unspecified-precision loads and saves in matched pairs, to save and restore data. The format of the saved data is implementation-dependent.

## Examples

```
FLDMIAS r2, {s1-s5}
FSTMFDD r13!, {d3-d6}
FSTMIAS r0!, {s31}
```

The following instructions are equivalent:

```
FLDMIAS r7, {s3-s7}
FLDMIAS r7, {s3,s4,s5,s6,s7}
```

The following instructions must always be used as a matching pair:

```
FSTMFDX r13!, {d0-d3}
FLDMFDX r13!, {d0-d3}
```

The following instruction is illegal, as the registers in the list are not consecutive:

```
FLDMIAD r13!, {d0,d2,d3}
```

### 5.7.9 FMAC, FNMAC, FMSC, and FNMSC

Floating-point multiply-accumulate, negate-multiply-accumulate, multiply-subtract and negate-multiply-subtract.

These instructions can be scalar, vector, or mixed (see *Vector and scalar operations* on page 5-7).

#### Syntax

`<op><precision>{cond} Fd, Fn, Fm`

where:

`<op>` must be one of FMAC, FNMAC, FMSC, or FNMSC.

`<precision>` must be either S for single-precision, or D for double-precision.

`cond` is an optional condition code (see *VFP and condition codes* on page 5-8).

`Fd` is the VFP register for the result.

`Fn` is the VFP register holding the first operand.

`Fm` is the VFP register holding the second operand.

The precision of `Fd`, `Fn` and `Fm` must match the precision specified in `<precision>`.

#### Usage

The FMAC instruction calculates  $Fd + Fn * Fm$  and places the result in `Fd`.

The FNMAC instruction calculates  $Fd - Fn * Fm$  and places the result in `Fd`.

The FMSC instruction calculates  $-Fd + Fn * Fm$  and places the result in `Fd`.

The FNMSC instruction calculates  $-Fd - Fn * Fm$  and places the result in `Fd`.

#### Exceptions

These operations can produce Invalid Operation, Overflow, Underflow, or Inexact exceptions.

#### Examples

|          |               |
|----------|---------------|
| FMACD    | d8, d0, d8    |
| FMACS    | s20, s24, s28 |
| FNMSCSLE | s6, s0, s26   |

### 5.7.10 FMDRR and FMRRD

Transfer contents between two ARM registers and a double-precision floating-point register.

#### Syntax

`FMDRR{cond} Dn, Rd, Rn`

`FMRRD{cond} Rd, Rn, Dn`

where:

*cond* is an optional condition code (see *VFP and condition codes* on page 5-8).

*Dn* is the VFP double-precision register.

*Rd, Rn* are ARM registers. Do not use r15.

#### Usage

`FMDRR Dn, Rd, Rn` transfers the contents of *Rd* into the low half of *Dn*, and the contents of *Rn* into the high half of *Dn*.

`FMRRD Rd, Rn, Dn` transfers the contents of the low half of *Dn* into *Rd*, and the contents of the high half of *Dn* into *Rn*.

#### Exceptions

These instructions do not produce any exceptions.

#### Architectures

These instructions are available in VFPv2 and above.

#### Examples

```
FMDRR d5, r3, r4
FMRRDPL r12, r2, d2
```

### 5.7.11 FMDHR, FMDLR, FMRDH, and FMRDL

Transfer contents between an ARM register and a half of a double-precision floating-point register.

#### Syntax

FMDHR{*cond*} *Dn*, *Rd*

FMDLR{*cond*} *Dn*, *Rd*

FMRDH{*cond*} *Rd*, *Dn*

FMRDL{*cond*} *Rd*, *Dn*

where:

*cond* is an optional condition code (see *VFP and condition codes* on page 5-8).

*Dn* is the VFP double-precision register.

*Rd* is the ARM register. *Rd* must not be r15.

#### Usage

These instructions are used together as matched pairs:

- Use FMDHR with FMDLR:
 

|       |                                                                 |
|-------|-----------------------------------------------------------------|
| FMDHR | copy the contents of <i>Rd</i> into the high half of <i>Dn</i>  |
| FMDLR | copy the contents of <i>Rd</i> into the low half of <i>Dn</i> . |
- Use FMRDH with FMRDL:
 

|       |                                                                 |
|-------|-----------------------------------------------------------------|
| FMRDH | copy the contents of the high half of <i>Dn</i> into <i>Rd</i>  |
| FMRDL | copy the contents of the low half of <i>Dn</i> into <i>Rd</i> . |

#### Exceptions

These instructions do not produce any exceptions.

#### Examples

```
FMDHR d5, r3
FMDLR d5, r12
FMRDH r5, d3
FMRDL r9, d3
FMDLRPL d2, r1
```

### 5.7.12 FMRS and FMSR

Transfer contents between a single-precision floating-point register and an ARM register.

#### Syntax

FMRS{*cond*} *Rd*, *Sn*

FMSR{*cond*} *Sn*, *Rd*

where:

*cond* is an optional condition code (see *VFP and condition codes* on page 5-8).

*Sn* is the VFP single-precision register.

*Rd* is the ARM register. *Rd* must not be r15.

#### Usage

The FMRS instruction transfers the contents of *Sn* into *Rd*.

The FMSR instruction transfers the contents of *Rd* into *Sn*.

#### Exceptions

These instructions do not produce any exceptions.

#### Examples

|        |         |
|--------|---------|
| FMRS   | r2, s0  |
| FMSRNE | s30, r5 |

### 5.7.13 FMRRS and FMSRR

Transfer contents between two consecutive single-precision floating-point registers and two ARM registers.

#### Syntax

FMRRS{*cond*} *Rd*, *Rn*, {*Sm*, *Sm'*}

FMSRR{*cond*} {*Sm*, *Sm'*}, *Rd*, *Rn*

where:

*cond* is an optional condition code (see *VFP and condition codes* on page 5-8).

*Sm* is a VFP single-precision register.

*Sm'* is the next VFP single-precision registers after *Sm*.

*Rd*, *Rn* are the ARM registers. Do not use r15.

#### Usage

The FMRRS instruction transfers the contents of *Sm* into *Rd*, and the contents of *Sm'* into *Rn*.

The FMSRR instruction transfers the contents of *Rd* into *Sm*, and the contents of *Rn* into *Sm'*.

#### Exceptions

These instructions do not produce any exceptions.

#### Architectures

These instructions are available in VFPv2 and above.

#### Examples

```
FMRRS r2, r3, {s0,s1}
FMSRRNE {s27,s28}, r5, r2
```

#### Incorrect examples

```
FMRRS r2, r3, {s2,s4} ; VFP registers must be consecutive
FMSRR {s5,s6}, r15, r0 ; you must not use r15
```

### 5.7.14 FMRX, FMXR, and FMSTAT

Transfer contents between an ARM register and a VFP system register.

#### Syntax

`FMRX{cond} Rd, VFPsysreg`

`FMXR{cond} VFPsysreg, Rd`

`FMSTAT{cond}`

where:

*cond* is an optional condition code (see *VFP and condition codes* on page 5-8).

*VFPsysreg* is the VFP system register, usually FPSCR, FPSID, or FPEXC (see *Floating-point registers* on page 5-5).

*Rd* is the ARM register.

#### Usage

The FMRX instruction transfers the contents of *VFPsysreg* into *Rd*.

The FMXR instruction transfers the contents of *Rd* into *VFPsysreg*.

The FMSTAT instruction is a synonym for FMRX r15, FPSCR. It transfers the floating-point condition flags to the corresponding flags in the ARM CPSR (see *VFP and condition codes* on page 5-8).

#### Note

These instructions stall the ARM until all current VFP operations complete.

#### Exceptions

These instructions do not produce any exceptions.

#### Examples

```
FMSTAT
FMSTATNE
FMXR FPSCR, r2
FMRX r3, FPSID
```

### 5.7.15 FMUL and FNMUL

Floating-point multiply and negate-multiply.

FMUL and FNMUL can be scalar, vector, or mixed (see *Vector and scalar operations* on page 5-7).

#### Syntax

FMUL<precision>{cond} Fd, Fn, Fm

FNMUL<precision>{cond} Fd, Fn, Fm

where:

<precision> must be either S for single-precision, or D for double-precision.

cond is an optional condition code (see *VFP and condition codes* on page 5-8).

Fd is the VFP register for the result.

Fn is the VFP register holding the first operand.

Fm is the VFP register holding the second operand.

The precision of Fd, Fn and Fm must match the precision specified in <precision>.

#### Usage

The FMUL instruction multiplies the values in Fn and Fm and places the result in Fd.

The FNMUL instruction multiplies the values in Fn and Fm and places the negation of the result in Fd.

#### Exceptions

FMUL and FNMUL operations can produce Invalid Operation, Overflow, Underflow, or Inexact exceptions.

#### Examples

|         |               |
|---------|---------------|
| FNMULS  | s10, s10, s14 |
| FMULDLT | d0, d7, d8    |



### 5.7.16 FSITO and FUITO

Convert signed integer to floating-point and unsigned integer to floating-point.

FSITO and FUITO are always scalar.

#### Syntax

FSITO<precision>{cond} Fd, Sm

FUITO<precision>{cond} Fd, Sm

where:

<precision> must be either S for single-precision, or D for double-precision.

cond is an optional condition code (see *VFP and condition codes* on page 5-8).

Fd is a VFP register for the result. The precision of Fd must match the precision specified in <precision>.

Sm is a single-precision VFP register holding the integer operand.

#### Usage

The FSITO instruction converts the signed integer value in Sm to floating-point and places the result in Fd.

The FUITO instruction converts the unsigned integer value in Sm to floating-point and places the result in Fd.

#### Exceptions

FSITOS and FUITOS instructions can produce Inexact exceptions.

FSITOD and FUITOD instructions do not produce any exceptions.

#### Examples

```
FUITOD d3, s31 ; unsigned integer to double-precision
FSITOD d5, s16 ; signed integer to double-precision
FSITOSNE s2, s2 ; signed integer to single-precision
```

### 5.7.17 FSQRT

Floating-point square root instruction.

This instruction can be scalar, vector, or mixed (see *Vector and scalar operations* on page 5-7).

#### Syntax

FSQRT<precision>{cond} Fd, Fm

where:

<precision> must be either S for single-precision, or D for double-precision.

cond is an optional condition code (see *VFP and condition codes* on page 5-8).

Fd is the VFP register for the result.

Fm is the VFP register holding the operand.

The precision of Fd and Fm must match the precision specified in <precision>.

#### Usage

The FSQRT instruction calculates the square root of the value of the contents of Fm and places the result in Fd.

#### Exceptions

FSQRT operations can produce Invalid Operation or Inexact exceptions.

#### Examples

|          |          |
|----------|----------|
| FSQRTS   | s4, s28  |
| FSQRTD   | d14, d6  |
| FSQRTSNE | s15, s13 |

### 5.7.18 FTOSI and FTUI

Convert floating-point to signed integer and floating-point to unsigned integer.

FTOSI and FTUI are always scalar.

#### Syntax

FTOSI{Z}<precision>{cond} Sd, Fm

FTUI{Z}<precision>{cond} Sd, Fm

where:

- Z** is an optional parameter specifying rounding towards zero. If specified, this overrides the rounding mode currently specified in the FPSCR. The FPSCR is not altered.
- <precision>** must be either S for single-precision, or D for double-precision.
- cond** is an optional condition code (see *VFP and condition codes* on page 5-8).
- Sd** is a single-precision VFP register for the integer result.
- Fm** is a VFP register holding the operand. The precision of *Fm* must match the precision specified in *<precision>*.

#### Usage

The FTOSI instruction converts the floating-point value in *Fm* to a signed integer and places the result in *Sd*.

The FTUI instruction converts the floating-point value in *Fm* to an unsigned integer and places the result in *Sd*.

#### Exceptions

FTOSI and FTUI instructions can produce Invalid Operation or Inexact exceptions.

#### Examples

|         |         |
|---------|---------|
| FTOSID  | s10, d2 |
| FTUID   | s3, d1  |
| FTOSIZS | s3, s31 |

## 5.8 VFP pseudo-instruction

There is one VFP pseudo-instruction.

### 5.8.1 FLD pseudo-instruction

The FLD pseudo-instruction loads a VFP floating-point register with a single-precision or double-precision floating-point constant.

---

#### Note

You can use FLD only if the command-line option `--fpu` is set to `vfpv2`.

This section describes the FLD *pseudo*-instruction only. See *FLD and FST* on page 5-22 for information on the FLD *instruction*.

---

#### Syntax

```
FLD<precision>{cond} fp-register,=fp-literal
```

where:

*<precision>* can be S for single-precision, or D for double-precision.

*cond* is an optional condition code.

*fp-register* is the floating-point register to be loaded.

*fp-literal* is a single-precision or double-precision floating-point literal (see *Floating-point literals* on page 3-27).

#### Usage

The assembler places the constant in a literal pool and generates a program-relative FLD instruction to read the constant from the literal pool. One word in the literal pool is used to store a single-precision constant. Two words are used to store a double-precision constant.

The offset from the PC to the constant must be less than 1KB. You are responsible for ensuring that there is a literal pool within range. See *LTORG* on page 6-15 for more information.

#### Examples

```
FLDD d1,=3.12E106 ; loads 3.12E106 into d1
FLDS s31,=3.12E-16 ; loads 3.12E-16 into s31
```

## 5.9 VFP directives and vector notation

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

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

- `VFPASSERT SCALAR` on page 5-38
- `VFPASSERT VECTOR` on page 5-39.

If you use `VFPASSERT` directives, you must specify vector details in all VFP data processing instructions. The vector notation is described in *Vector notation*. If you do not use `VFPASSERT` directives you must not use this notation.

### 5.9.1 Vector notation

In VFP data processing instructions, specify vectors of VFP registers using angle brackets:

- `sn` is a single-precision scalar register  $n$
- `sn<>` is a single-precision vector whose length and stride are given by the current vector length and stride, starting at register  $n$
- `sn<L>` is a single-precision vector of length  $L$ , stride 1, starting at register  $n$
- `sn<L:S>` is a single-precision vector of length  $L$ , stride  $S$ , starting at register  $n$
- `dn` is a double-precision scalar register  $n$
- `dn<>` is a double-precision vector whose length and stride are given by the current vector length and stride, starting at register  $n$
- `dn<L>` is a double-precision vector of length  $L$ , stride 1, starting at register  $n$
- `dn<L:S>` is a double-precision vector of length  $L$ , stride  $S$ , starting at register  $n$ .

You can use this vector notation with names defined using the `DN` and `SN` directives (see *DN and SN* on page 6-12).

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

## 5.9.2 VFPASSERT SCALAR

The VFPASSERT SCALAR directive informs the assembler that following VFP instructions are in scalar mode.

### 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 the 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 conformant. See the *Using the Procedure Call Standard* appendix in *RealView Compilation Tools v2.2 Developer Guide* for more information.

See also:

- *Vector notation* on page 5-37
- *VFPASSERT VECTOR* on page 5-39.

---

### 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
fadd d4, d4, d0 ; okay
fadds s4<3>, s0, s8<3> ; ERROR, vector in scalar mode
fabss s24<1>, s28<1> ; ERROR, vector in scalar mode
 ; (even though length==1)
```

### 5.9.3 VFPASSERT VECTOR

The VFPASSERT VECTOR directive informs the assembler that 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 the 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 conformant. See the *Using the Procedure Call Standard* appendix in *RealView Compilation Tools v2.2 Developer Guide* for more information.

See:

- *Vector notation* on page 5-37
- *VFPASSERT SCALAR* on page 5-38.

---

#### 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**

```

FMRX r10,FPSCR
BIC r10,r10,#0x00370000
ORR r10,r10,#0x00020000 ; set length = 3, stride = 1
FMXR FPSCR,r10

VFPASSERT VECTOR ; assert vector mode, unspecified length & stride
faddd d4, d4, d0 ; ERROR, scalar in vector mode
fadds s16<3>, s0, s8<3> ; okay
fabss s24<1>, s28<1> ; wrong length, but not faulted (unspecified)

FMRX r10,FPSCR
BIC r10,r10,#0x00370000
ORR r10,r10,#0x00030000 ; set length = 4, stride = 1
FMXR FPSCR,r10

VFPASSERT VECTOR<4> ; assert vector mode, length 4, stride 1
fadds s24<4>, s0, s8<4> ; okay
fabss s24<2>, s24<2> ; ERROR, wrong length

FMRX r10,FPSCR
BIC r10,r10,#0x00370000
ORR r10,r10,#0x00130000 ; set length = 4, stride = 2
FMXR FPSCR,r10

VFPASSERT VECTOR<4:2> ; assert vector mode, length 4, stride 2
fadds s8<4>, s0, s16<4> ; ERROR, wrong stride
fabss s16<4:2>, s28<4:2> ; okay
fadds s8<>, s2, s16<> ; okay (s8 and s16 both have
 ; length 4 and stride 2.
 ; s2 is scalar.)

```



# Chapter 6

## Directives Reference

This chapter describes the directives that are provided by the ARM® assembler, `armasm`. It contains the following sections:

- *Alphabetical list of directives* on page 6-2
- *Symbol definition directives* on page 6-4
- *Data definition directives* on page 6-13
- *Assembly control directives* on page 6-28
- *Frame directives* on page 6-37
- *Reporting directives* on page 6-52
- *Instruction set and syntax selection directives* on page 6-57
- *Miscellaneous directives* on page 6-61.

---

### Note

---

None of these directives is available in the inline assemblers in the ARM C and C++ compilers.

---

## 6.1 Alphabetical list of directives

Table 6-1 shows a complete list of the ARM directives. Use it to locate individual directives described in the rest of this chapter.

**Table 6-1 Location of directives**

| Directive                | Page      | Directive                       | Page      | Directive                     | Page      |
|--------------------------|-----------|---------------------------------|-----------|-------------------------------|-----------|
| ALIGN                    | page 6-62 | EQU                             | page 6-68 | KEEP                          | page 6-78 |
| ARM <i>and</i> CODE32    | page 6-58 | EXPORT <i>or</i> GLOBAL         | page 6-69 | LCLA, LCLL, <i>and</i> LCLS   | page 6-7  |
| AREA                     | page 6-64 | EXPORTAS                        | page 6-71 | LTORG                         | page 6-15 |
| ASSERT                   | page 6-52 | EXTERN                          | page 6-72 | MACRO <i>and</i> MEND         | page 6-29 |
| CN                       | page 6-10 | FIELD                           | page 6-17 | MAP                           | page 6-16 |
| CODE16                   | page 6-60 | FRAME ADDRESS                   | page 6-39 | MEND <i>see</i> MACRO         | page 6-29 |
| COMMON                   | page 6-27 | FRAME POP                       | page 6-40 | MEXIT                         | page 6-32 |
| CP                       | page 6-11 | FRAME PUSH                      | page 6-41 | NOFP                          | page 6-79 |
| DATA                     | page 6-27 | FRAME REGISTER                  | page 6-43 | OPT                           | page 6-54 |
| DCB                      | page 6-19 | FRAME RESTORE                   | page 6-44 | PRESERVE8 <i>see</i> REQUIRE8 | page 6-80 |
| DCD <i>and</i> DCDU      | page 6-20 | FRAME SAVE                      | page 6-46 | PROC <i>see</i> FUNCTION      | page 6-50 |
| DCDO                     | page 6-21 | FRAME STATE REMEMBER            | page 6-47 | REQUIRE                       | page 6-79 |
| DCFD <i>and</i> DCFDU    | page 6-22 | FRAME STATE RESTORE             | page 6-48 | REQUIRE8 <i>and</i> PRESERVE8 | page 6-80 |
| DCFS <i>and</i> DCFSU    | page 6-23 | FUNCTION <i>or</i> PROC         | page 6-50 | RLIST                         | page 6-9  |
| DCI                      | page 6-24 | GBLA, GBLL, <i>and</i> GBLS     | page 6-5  | RN                            | page 6-82 |
| DCQ <i>and</i> DCQU      | page 6-25 | GET <i>or</i> INCLUDE           | page 6-74 | ROUT                          | page 6-83 |
| DCW <i>and</i> DCWU      | page 6-26 | GLOBAL <i>see</i> EXPORT        | page 6-69 | SETA, SETL, <i>and</i> SETS   | page 6-8  |
| DN <i>and</i> SN         | page 6-12 | IF, ELSE, ENDF, <i>and</i> ELIF | page 6-33 | SN <i>see</i> DN              | page 6-12 |
| ELIF, ELSE <i>see</i> IF | page 6-33 | IMPORT                          | page 6-75 | SPACE                         | page 6-18 |
| END                      | page 6-66 | INCBIN                          | page 6-77 | THUMB                         | page 6-59 |

**Table 6-1 Location of directives (continued)**

| <b>Directive</b>       | <b>Page</b> | <b>Directive</b>       | <b>Page</b> | <b>Directive</b>      | <b>Page</b> |
|------------------------|-------------|------------------------|-------------|-----------------------|-------------|
| ENDFUNC <i>or</i> ENDP | page 6-51   | INCLUDE <i>see</i> GET | page 6-74   | TTL <i>and</i> SUBT   | page 6-56   |
| ENDIF <i>see</i> IF    | page 6-33   | INFO                   | page 6-53   | WHILE <i>and</i> WEND | page 6-36   |
| ENTRY                  | page 6-67   |                        |             |                       |             |

## 6.2 Symbol definition directives

This section describes the following directives:

- *GBLA*, *GBLL*, and *GBLS* on page 6-5  
Declare a global arithmetic, logical, or string variable.
- *LCLA*, *LCLL*, and *LCLS* on page 6-7  
Declare a local arithmetic, logical, or string variable.
- *SETA*, *SETL*, and *SETS* on page 6-8  
Set the value of an arithmetic, logical, or string variable.
- *RLIST* on page 6-9  
Define a name for a set of general-purpose registers.
- *CN* on page 6-10  
Define a coprocessor register name.
- *CP* on page 6-11  
Define a coprocessor name.
- *DN* and *SN* on page 6-12  
Define a double-precision or single-precision VFP register name.

### 6.2.1 GBLA, GBLL, and GBLS

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, "".

#### Syntax

`<gb1x> variable`

where:

`<gb1x>` 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

Using one of these directives for a variable that is already defined re-initializes the variable to the same values given above.

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 (see *SETA*, *SETL*, and *SETS* on page 6-8).

See *LCLA*, *LCLL*, and *LCLS* on page 6-7 for information on declaring local variables.

Global variables can also be set with the `-predefine` assembler command-line option. See *Command syntax* on page 3-2 for more information.

Examples

Example 6-1 declares a variable `objectsize`, sets the value of `objectsize` to `0xFF`, and then uses it later in a `SPACE` directive.

Example 6-1

---

```
objectsize GBLA objectsize ; declare the variable name
 SETA 0xFF ; set its value
 .
 . ; other code
 .
 SPACE objectsize ; quote the variable
```

---

Example 6-2 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`.

Example 6-2

---

```
armasm --pd "objectsize SETA 0xFF" --o objectfile sourcefile
```

---

## 6.2.2 LCLA, LCLL, and LCLS

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, "".

### Syntax

`<lc1x> variable`

where:

`<lc1x>` 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

Using one of these directives for a variable that is already defined re-initializes the variable to the same values given above.

The scope of the variable is limited to a particular instantiation of the macro that contains it (see *MACRO* and *MEND* on page 6-29).

Set the value of the variable with a SETA, SETL, or SETS directive (see *SETA*, *SETL*, and *SETS* on page 6-8).

See *GBLA*, *GBLL*, and *GBLS* on page 6-5 for information on declaring global variables.

### 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
```

### 6.2.3 SETA, SETL, and SETS

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.

#### Syntax

*variable* <setx> *expr*

where:

<setx> is one of SETA, SETL, or SETS.

*variable* is the name of a variable declared by a GBLA, GBLL, GBLS, LCLA, LCLL, or LCLS directive.

*expr* is an expression that is:

- numeric, for SETA (see *Numeric expressions* on page 3-25)
- logical, for SETL (see *Logical expressions* on page 3-28)
- string, for SETS (see *String expressions* on page 3-24).

#### Usage

You must declare *variable* using a global or local declaration directive before using one of these directives. See *GBLA*, *GBLL*, and *GBLS* on page 6-5 and *LCLA*, *LCLL*, and *LCLS* on page 6-7 for more information.

You can also predefine variable names on the command line. See *Command syntax* on page 3-2 for more information.

#### Examples

|               |      |               |
|---------------|------|---------------|
|               | GBLA | VersionNumber |
| VersionNumber | SETA | 21            |
|               | GBLL | Debug         |
| Debug         | SETL | {TRUE}        |
|               | GBLS | VersionString |
| VersionString | SETS | "Version 1.0" |



## 6.2.4 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 listed in *Predefined register and coprocessor names* on page 3-15.

*list-of-registers*

is a comma-delimited list of register names and/or 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 `--checkreglist` 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,r15}
```

### 6.2.5 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 listed in *Predefined register and coprocessor names* on page 3-15.

*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
```

## 6.2.6 CP

The CP directive defines a name for a specified coprocessor. The coprocessor number must be within the range 0 to 15.

### 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 listed in *Predefined register and coprocessor names* on page 3-15.

*expr* evaluates to a coprocessor number from 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
```

## 6.2.7 DN and SN

The DN directive defines a name for a specified double-precision VFP register. The names d0-d15 and D0-D15 are predefined.

The SN directive defines a name for a specified single-precision VFP register. The names s0-s31 and S0-S31 are predefined.

### Syntax

*name* DN *expr*

*name* SN *expr*

where:

*name* is the name to be assigned to the VFP register. *name* cannot be the same as any of the predefined names listed in *Predefined register and coprocessor names* on page 3-15.

*expr* evaluates to a double-precision VFP register number from 0 to 15, or a single-precision VFP register number from 0 to 31 as appropriate.

### Usage

Use DN or SN to allocate convenient names to VFP registers, to help you to remember what you use each one for.

#### ———— Note ————

Avoid conflicting uses of the same register under different names.

You cannot specify a vector length in a DN or SN directive (see *VFP directives and vector notation* on page 5-37).

### 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
```

## 6.3 Data definition directives

This section describes the following directives to allocate memory, define data structures, set initial contents of memory:

- *LTORG* on page 6-15  
Set an origin for a literal pool.
- *MAP* on page 6-16  
Set the origin of a storage map.
- *FIELD* on page 6-17  
Define a field within a storage map.
- *SPACE* on page 6-18  
Allocate a zeroed block of memory.
- *DCB* on page 6-19  
Allocate bytes of memory, and specify the initial contents.
- *DCD* and *DCDU* on page 6-20  
Allocate words of memory, and specify the initial contents.
- *DCDO* on page 6-21  
Allocate words of memory, and specify the initial contents as offsets from the static base register.
- *DCFD* and *DCFDU* on page 6-22  
Allocate doublewords of memory, and specify the initial contents as double-precision floating-point numbers.
- *DCFS* and *DCFSU* on page 6-23  
Allocate words of memory, and specify the initial contents as single-precision floating-point numbers.
- *DCI* on page 6-24  
Allocate words of memory, and specify the initial contents. Mark the location as code not data.
- *DCQ* and *DCQU* on page 6-25  
Allocate doublewords of memory, and specify the initial contents as 64-bit integers.

- *DCW and DCWU* on page 6-26  
Allocate halfwords of memory, and specify the initial contents.
- *COMMON* on page 6-27  
Allocate a block of memory at a symbol, and specify the alignment.
- *DATA* on page 6-27  
Mark data within a code section. Obsolete, for backwards compatibility only.

### 6.3.1 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, LDFD, and LDFS pseudo-instructions. See *LDR pseudo-instruction* on page 4-151. 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.
```

### 6.3.2 MAP

The MAP directive sets the origin of a storage map to a specified address. The storage-map location counter, {VAR}, is set to the same address. ^ is a synonym for MAP.

#### Syntax

MAP *expr*{, *base-register*}

where:

*expr* is a numeric or program-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 program-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. See *FIELD* on page 6-17 for an example.

The MAP directive can be used any number of times to define multiple storage maps.

The {VAR} counter is set to zero before the first MAP directive is used.

#### Examples

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



### 6.3.3 FIELD

The FIELD directive describes space within a storage map that has been defined using the MAP directive. # is a synonym for FIELD.

#### 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 (see *MAP* on page 6-16).

#### Example

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]
```

### 6.3.4 SPACE

The SPACE directive reserves a zeroed block of memory. % is a synonym for SPACE.

#### Syntax

```
{label} SPACE expr
```

where:

*expr* evaluates to the number of zeroed bytes to reserve (see *Numeric expressions* on page 3-25).

#### Usage

Use the ALIGN directive to align any code following a SPACE directive. See *ALIGN* on page 6-62 for more information.

See also:

- *DCB* on page 6-19
- *DCD and DCDU* on page 6-20
- *DCDO* on page 6-21
- *DCW and DCWU* on page 6-26.

#### Example

```
 AREA MyData, DATA, READWRITE
data1 SPACE 255 ; defines 255 bytes of zeroed store
```

### 6.3.5 DCB

The DCB directive allocates one or more bytes of memory, and defines the initial runtime contents of the memory. = is a synonym for DCB.

#### Syntax

```
{label} DCB expr{,expr}...
```

where:

*expr* is either:

- a numeric expression that evaluates to an integer in the range –128 to 255 (see *Numeric expressions* on page 3-25).
- 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. See *ALIGN* on page 6-62 for more information.

See also:

- *DCD and DCDU* on page 6-20
- *DCQ and DCQU* on page 6-25
- *DCW and DCWU* on page 6-26
- *SPACE* on page 6-18.

#### Example

Unlike C strings, ARM assembler strings are not null-terminated. You can construct a null-terminated C string using DCB as follows:

```
C_string DCB "C_string",0
```

### 6.3.6 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.

& is a synonym for DCD.

DCDU is the same, except that the memory alignment is arbitrary.

#### Syntax

```
{label} DCD{U} expr{,expr}
```

where:

*expr* is either:

- a numeric expression (see *Numeric expressions* on page 3-25).
- a program-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.

See also:

- *DCB* on page 6-19
- *DCW and DCWU* on page 6-26
- *DCQ and DCQU* on page 6-25
- *SPACE* on page 6-18.

#### 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
```

### 6.3.7 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.
```

### 6.3.8 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. Double-precision numbers occupy two words and must be word aligned to be used in arithmetic operations.

DCFUD 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 (see *Floating-point literals* on page 3-27).

#### Usage

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.

See also *DCFS* and *DCFSU* on page 6-23.

#### Examples

```
DCFD 1E308,-4E-100
DCFUD 10000,-.1,3.1E26
```

### 6.3.9 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. Single-precision numbers occupy one word and must be word aligned to be used in arithmetic operations.

DCDSU 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 (see *Floating-point literals* on page 3-27).

#### Usage

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.

See also *DCFD* and *DCFDU* on page 6-22.

#### Example

```
DCFS 1E3,-4E-9
DCFSU 1.0,-.1,3.1E6
```

### 6.3.10 DCI

In ARM code, the DCI directive allocates one or more words of memory, aligned on four-byte boundaries, and defines the initial runtime contents of the memory.

In Thumb® code, the DCI directive allocates one or more halfwords of memory, aligned on two-byte boundaries, and defines the initial runtime contents of the memory.

#### Syntax

```
{label} DCI expr{,expr}
```

where:

*expr* is a numeric expression (see *Numeric expressions* on page 3-25).

#### 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 inserts up to three bytes of padding before the first defined word, if necessary, to achieve four-byte alignment. In Thumb code, DCI inserts an initial byte of padding, if necessary, to achieve two-byte alignment.

You can use DCI to insert a bit pattern into the instruction stream. For example, use:

```
DCI 0x46c0
```

to insert the Thumb operation MOV r8,r8.

See also *DCD and DCDU* on page 6-20 and *DCW and DCWU* on page 6-26.

#### Example

```
MACRO ; this macro translates newinstr Rd,Rm
 ; to the appropriate machine code
newinst $Rd,$Rm
DCI 0xe16f0f10 :OR: ($Rd:SHL:12) :OR: $Rm
MEND
```



### 6.3.11 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 (see *Numeric literals* on page 3-26).

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.

See also:

- *DCB* on page 6-19
- *DCD* and *DCDU* on page 6-20
- *DCW* and *DCWU* on page 6-26
- *SPACE* on page 6-18.

#### Example

```

data AREA MiscData, DATA, READWRITE
 DCQ -225,2_101 ; 2_101 means binary 101.
 DCQU number+4 ; number must already be defined.
```

### 6.3.12 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 expr{,expr}...
```

where:

*expr* is a numeric expression that evaluates to an integer in the range –32768 to 65535 (see *Numeric expressions* on page 3-25).

#### 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.

See also:

- *DCB* on page 6-19
- *DCD and DCDU* on page 6-20
- *DCQ and DCQU* on page 6-25
- *SPACE* on page 6-18.

#### Example

```
data DCW -225,2*number ; number must already be defined
 DCWU number+4
```

### 6.3.13 COMMON

The `COMMON` directive allocates a block of memory, of the defined size, at the specified symbol. You specify how the memory is aligned. If alignment is omitted, the default alignment is 4. If size is omitted, the default size is 0.

You can access this memory as you would any other memory, but no space is allocated in object files.

#### Syntax

```
COMMON symbol{, size{, alignment}}
```

where:

*symbol* is the symbol name. The symbol name is case-sensitive.

*size* is the number of bytes to reserve.

*alignment* is the alignment.

#### Usage

The linker allocates the required space as zero initialized memory during the link stage.

#### Example

```
COMMON xyz,255,4 ; defines 255 bytes of ZI store, word-aligned
```

### 6.3.14 DATA

The `DATA` directive is no longer required. It is ignored by the assembler.

## 6.4 Assembly control directives

This section describes the following directives to control conditional assembly, looping, inclusions, and macros:

- *MACRO* and *MEND* on page 6-29
- *MEXIT* on page 6-32
- *IF*, *ELSE*, *ENDIF*, and *ELIF* on page 6-33
- *WHILE* and *WEND* on page 6-36.

### 6.4.1 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.

## 6.4.2 MACRO and MEND

The **MACRO** directive marks the start of the definition of a macro. Macro expansion terminates at the **MEND** directive. See *Using macros* on page 2-43 for more information.

### Syntax

Two directives are used to define a macro. The syntax is:

```

MACRO
{$label} macroname {$parameter{,$parameter}...}
; code
MEND

```

where:

|                    |                                                                                                                                                                                                                                                                          |
|--------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>\$label</i>     | is a parameter that is substituted with a symbol given when the macro is invoked. The symbol is usually a label.                                                                                                                                                         |
| <i>macroname</i>   | is the name of the macro. It must not begin with an instruction or directive name.                                                                                                                                                                                       |
| <i>\$parameter</i> | is a parameter that is substituted when the macro is invoked. A default value for a parameter can be set using this format:<br><i>\$parameter</i> ="default value"<br>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. See *MEXIT* on page 6-32 if you want to enable an early exit from a macro, for example, from within a loop.

Within the macro body, parameters such as *\$label*, *\$parameter* can be used in the same way as other variables (see *Assembly time substitution of variables* on page 3-19). 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.

Macros define the scope of local variables (see *LCLA*, *LCLL*, and *LCLS* on page 6-7).

Macros can be nested (see *Nesting directives* on page 6-28).

## Examples

```

; macro definition

$label MACRO ; start macro definition
 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

```

Using a macro to produce 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
```

### 6.4.3 MEXIT

The MEXIT directive is used to exit 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.

See also *MACRO* and *MEND* on page 6-29.

#### Example

```
$abc MACRO
 macro abc $param1,$param2
 ; code
 WHILE condition1
 ; code
 IF condition2
 ; code
 MEXIT
 ELSE
 ; code
 ENDIF
 WEND
 ; code
 MEND
```



#### 6.4.4 IF, ELSE, ENDIF, and ELIF

The IF directive introduces a condition that is used to decide whether to assemble a sequence of instructions and/or directives. `[` is a synonym for IF.

The ELSE directive marks the beginning of a sequence of instructions and/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 and/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. See *Using ELIF* on page 6-34 for details.

##### Syntax

```
IF logical-expression ... {ELSE ...} ENDIF
```

where:

*logical-expression*

is an expression that evaluates to either {TRUE} or {FALSE}.

See *Relational operators* on page 3-34.

##### Usage

Use IF with ENDIF, and optionally with ELSE, for sequences of instructions and/or directives that are only to be assembled or acted on under a specified condition.

IF...ENDIF conditions can be nested (see *Nesting directives* on page 6-28).

## 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

Example 6-3 assembles the first set of instructions if NEWVERSION is defined, or the alternative set otherwise.

### Example 6-3 Assembly conditional on a variable being defined

---

```

IF :DEF:NEWVERSION
 ; first set of instructions/directives
ELSE
 ; alternative set of instructions/directives
ENDIF

```

---

Invoking `armasm` as follows defines `NEWVERSION`, so the first set of instructions and directives are assembled:

```
armasm --PD "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
```

Example 6-4 assembles the first set of instructions if `NEWVERSION` has the value `{TRUE}`, or the alternative set otherwise.

### Example 6-4 Assembly conditional on a variable value

---

```

IF NEWVERSION = {TRUE}
 ; first set of instructions/directives
ELSE
 ; alternative set of instructions/directives
ENDIF

```

---

Invoking `armasm` as follows causes the first set of instructions and directives to be assembled:

```
armasm --PD "NEWVERSION SETL {TRUE}" test.s
```

Invoking `armasm` as follows causes the second set of instructions and directives to be assembled:

```
armasm --PD "NEWVERSION SETL {FALSE}" test.s
```

### 6.4.5 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} (see *Logical expressions* on page 3-28).

#### 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 (see *Nesting directives* on page 6-28).

#### Example

```
count SETA 1 ; you are not restricted to
 WHILE count <= 4 ; such simple conditions
count SETA count+1 ; In this case,
 ; code ; this code will be
 ; code ; repeated four times
 WEND
```

## 6.5 Frame directives

This section describes the following directives:

- *FRAME ADDRESS* on page 6-39
- *FRAME POP* on page 6-40
- *FRAME PUSH* on page 6-41
- *FRAME REGISTER* on page 6-43
- *FRAME RESTORE* on page 6-44
- *FRAME RETURN ADDRESS* on page 6-45
- *FRAME SAVE* on page 6-46
- *FRAME STATE REMEMBER* on page 6-47
- *FRAME STATE RESTORE* on page 6-48
- *FRAME UNWIND ON* on page 6-49
- *FRAME UNWIND OFF* on page 6-49
- *FUNCTION* or *PROC* on page 6-50
- *ENDFUNC* or *ENDP* on page 6-51.

Correct use of these directives:

- enables the `armlink --callgraph` option to calculate stack usage of assembler functions.

The following rules are used to 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.

## 6.5.1 FRAME ADDRESS

The `FRAME ADDRESS` directive describes how to calculate the canonical frame address for following instructions. You can only use it in functions with `FUNCTION` and `ENDFUNC` or `PROC` and `ENDP` directives.

### Syntax

```
FRAME ADDRESS reg[,offset]
```

where:

|               |                                                                                                                                                 |
|---------------|-------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>reg</i>    | is the register on which the canonical frame address is to be based. This is <code>sp</code> unless the function uses a separate frame pointer. |
| <i>offset</i> | is the offset of the canonical frame address from <i>reg</i> . If <i>offset</i> 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.

#### 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` (see *FRAME PUSH* on page 6-41).

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` (see *FRAME POP* on page 6-40).

### 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
```

## 6.5.2 FRAME POP

Use the `FRAME POP` directive to inform the assembler when the callee reloads registers. 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.

### Syntax

There are two alternative syntaxes for `FRAME POP`:

```
FRAME POP {reglist}
```

```
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.

The assembler calculates the new offset for the canonical frame address. It assumes that:

- each ARM register popped occupied four bytes on the stack
- each VFP single-precision register popped occupied four bytes on the stack, plus an extra four-byte word for each list.

See *FRAME ADDRESS* on page 6-39 and *FRAME RESTORE* on page 6-44.



### 6.5.3 FRAME PUSH

Use the `FRAME PUSH` directive to inform the assembler when the callee saves registers, normally at function entry. You can only use it within functions with `FUNCTION` and `ENDFUNC` or `PROC` and `ENDP` directives.

#### Syntax

There are two alternative syntaxes for `FRAME PUSH`:

```
FRAME PUSH {reglist}
```

```
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.

The assembler calculates the new offset for the canonical frame address. 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.

See *FRAME ADDRESS* on page 6-39 and *FRAME SAVE* on page 6-46.

**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
```

## 6.5.4 FRAME REGISTER

Use the `FRAME REGISTER` directive to maintain a record of the locations of function arguments held in registers. You can only use it within functions with `FUNCTION` and `ENDFUNC` or `PROC` and `ENDP` directives.

### 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.

### 6.5.5 FRAME RESTORE

Use the `FRAME RESTORE` directive to inform the assembler that the contents of specified registers have been restored to the values they had on entry to the function. You can only use it within functions with `FUNCTION` and `ENDFUNC` or `PROC` and `ENDP` directives.

#### 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

Use `FRAME RESTORE` 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` (see *FRAME POP* on page 6-40).

---

## 6.5.6 FRAME RETURN ADDRESS

The FRAME RETURN ADDRESS directive provides for functions that use a register other than r14 for their return address. You can only use it within functions with FUNCTION and ENDFUNC or PROC and ENDP directives.

---

### Note

---

Any function that uses a register other than r14 for its return address is not AAPCS compliant. Such a function must not be exported.

---

### 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 r14 for its return address. Otherwise, a debugger cannot backtrace through the function.

Use FRAME RETURN ADDRESS immediately after the FUNCTION or PROC directive that introduces the function.

## 6.5.7 FRAME SAVE

The `FRAME SAVE` directive describes the location of saved register contents relative to the canonical frame address. You can only use it within functions with `FUNCTION` and `ENDFUNC` or `PROC` and `ENDP` directives.

### 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

Use `FRAME SAVE` 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` (see *FRAME PUSH* on page 6-41).

---

## 6.5.8 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. You can only use it within functions with `FUNCTION` and `ENDFUNC` or `PROC` and `ENDP` directives.

### 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. See:

- *FRAME STATE RESTORE* on page 6-48
- *FUNCTION or PROC* on page 6-50.

### 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

```

## 6.5.9 FRAME STATE RESTORE

The `FRAME STATE RESTORE` directive restores information about how to calculate the canonical frame address and locations of saved register values. You can only use it within functions with `FUNCTION` and `ENDFUNC` or `PROC` and `ENDP` directives.

### Syntax

`FRAME STATE RESTORE`

### Usage

See:

- *FRAME STATE REMEMBER* on page 6-47
- *FUNCTION* or *PROC* on page 6-50.



### 6.5.10 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.

See also *Controlling exception table generation* on page 3-13.

### 6.5.11 FRAME UNWIND OFF

The `FRAME UNWIND OFF` directive instructs the assembler to produce *nounwind* tables for this and subsequent functions.

#### Syntax

`FRAME UNWIND OFF`

#### Usage

You can use this directive outside functions. In this case, the assembler produces *nounwind* tables for all following functions until it reaches a `FRAME UNWIND ON` directive.

See also *Controlling exception table generation* on page 3-13.

## 6.5.12 FUNCTION or PROC

The FUNCTION directive marks the start of an AAPCS-conforming 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 will assume that the AAPCS is in use.

*reglist2* is an optional list of callee saved VFP registers.

### 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 *sp*, and the frame state stack to be empty.

Each FUNCTION directive must have a matching ENDFUNC directive. You must not nest FUNCTION/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.

See also *FRAME ADDRESS* on page 6-39 to *FRAME STATE RESTORE* on page 6-48.

### ———— 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. See *ALIGN* on page 6-62 for further information.

## 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

```

### 6.5.13 ENDFUNC or ENDP

The ENDFUNC directive marks the end of an AAPCS-conforming function (see *FUNCTION or PROC* on page 6-50). ENDP is a synonym for ENDFUNC.

## 6.6 Reporting directives

This section describes the following directives:

- *ASSERT*  
generates an error message if an assertion is false during assembly.
- *INFO* on page 6-53  
generates diagnostic information during assembly.
- *OPT* on page 6-54  
sets listing options.
- *TTL* and *SUBT* on page 6-56  
insert titles and subtitles in listings.

### 6.6.1 ASSERT

The *ASSERT* directive generates an error message during the second pass of the 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.

See also *INFO* on page 6-53.

#### Example

```

 ASSERT label1 <= label2 ; Tests if the address
 ; represented by label1
 ; is <= the address
 ; represented by label2.
```

## 6.6.2 INFO

The INFO directive supports diagnostic generation on either pass of the assembly.

! is very similar to INFO, but has less detailed reporting.

### Syntax

INFO *numeric-expression*, *string-expression*

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 during pass two.

If the expression does not evaluate to zero, *string-expression* is printed as an error message and the assembly fails.

*string-expression*

is an expression that evaluates to a string.

### Usage

INFO provides a flexible means of creating custom error messages. See *Numeric expressions* on page 3-25 and *String expressions* on page 3-24 for additional information on numeric and string expressions.

See also *ASSERT* on page 6-52.

### Examples

```
INFO 0, "Version 1.0"

IF endofdata <= label1
 INFO 4, "Data overrun at label1"
ENDIF
```

6.6.3 OPT

The OPT directive sets listing options from within the source code.

Syntax

OPT *n*

where:

*n* is the OPT directive setting. Table 6-2 lists valid settings.

Table 6-2 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. See *Command syntax* on page 3-2 for information on the `--list` option.

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

```

### 6.6.4 TTL and SUBT

The TTL directive inserts a title at the start of each page of a listing file. The title is printed on each page until a new TTL directive is issued.

The SUBT directive places a subtitle on the pages of a listing file. The subtitle is printed on each page until a new SUBT directive is issued.

#### 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 the pages 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 the pages 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.

#### Example

```
TTL First Title ; places a title on the first
 ; and subsequent pages of a
 ; listing file.
SUBT First Subtitle ; places a subtitle on the
 ; second and subsequent pages
 ; of a listing file.
```



## 6.7 Instruction set and syntax selection directives

This section describes the following directives:

- *ARM and CODE32* on page 6-58
- *THUMB* on page 6-59
- *CODE16* on page 6-60.

### 6.7.1 ARM and CODE32

The ARM directive and the CODE32 directive are synonyms. They instruct the assembler to interpret subsequent instructions as ARM instructions. If necessary, they also insert up to three bytes of padding to align to the next word boundary.

In this mode, the assembler accepts both the latest assembly language and the earlier version.

#### Syntax

ARM

CODE32

#### Usage

In files that contain a mixture of ARM and Thumb code:

- Use ARM when changing from Thumb state to ARM state. ARM (or CODE32) must precede any ARM code.
- Use THUMB when changing from ARM state to Thumb state, if the following code uses ARM syntax (see *THUMB* on page 6-59).
- Use CODE16 when changing from ARM state to Thumb state, if the following code uses old Thumb syntax (see *CODE16* on page 6-60).

ARM and CODE32 do not assemble to instructions that change the state. They only instruct the assembler to assemble ARM instructions as appropriate, and insert padding if necessary.

#### Example

This example shows how ARM and CODE16 can be used to branch from ARM to 16-bit Thumb instructions.

```

 AREA ChangeState, CODE, READONLY
 ARM
 ; This section starts in ARM state
 LDR r0,=start+1 ; Load the address and set the
 ; least significant bit
 BX r0 ; Branch and exchange instruction sets
 ; Not necessarily in same section
 ; Following instructions are old Thumb
 CODE16
start MOV r1,#10 ; old Thumb instructions

```

## 6.7.2 THUMB

The THUMB directive instructs the assembler to interpret subsequent instructions as 32-bit Thumb-2 instructions or 16-bit Thumb instructions using new syntax. If necessary, it also inserts padding to align to the next halfword boundary.

### Syntax

THUMB

### Usage

In files that contain a mixture of ARM and Thumb code:

- Use THUMB when changing from ARM state to Thumb state, if the following code uses ARM syntax.
- Use CODE16 when changing from ARM state to Thumb state, if the following code uses old Thumb syntax (see *CODE16* on page 6-60).
- Use ARM when changing from Thumb state to ARM state. ARM (or CODE32) must precede any ARM code (see *ARM and CODE32* on page 6-58).

THUMB does not assemble to an instruction that changes the state. It only instructs the assembler to assemble Thumb-2 or Thumb instructions as appropriate, and insert padding if necessary.

### Example

This example shows how ARM and THUMB can be used to branch from ARM to Thumb instructions.

```

AREA ChangeState, CODE, READONLY
ARM
 ; This section starts in ARM state
LDR r0,=start+1 ; Load the address and set the
 ; least significant bit
BX r0 ; Branch and exchange instruction sets

 ; Not necessarily in same section
THUMB ; Following instructions are Thumb-2

GSthing PROC
B {pc}+2 ; #0x8002
B {pc}+4 ; #0x8004

```

### 6.7.3 CODE16

The CODE16 directive instructs the assembler to interpret subsequent instructions as 16-bit Thumb instructions using old Thumb syntax. If necessary, it also inserts a byte of padding to align to the next halfword boundary.

#### Syntax

CODE16

#### Usage

In files that contain a mixture of ARM and Thumb code:

- Use CODE16 when changing from ARM state to Thumb state, if the following code uses old Thumb syntax.
- Use THUMB when changing from ARM state to Thumb state, if the following code uses ARM syntax (see *THUMB* on page 6-59).
- Use ARM when changing from Thumb state to ARM state. ARM (or CODE32) must precede any ARM code (see *ARM and CODE32* on page 6-58).

CODE16 does not assemble to instructions that change the state. It only instructs the assembler to assemble Thumb instructions as appropriate, and insert padding if necessary.

#### Example

This example shows how CODE16 can be used to branch from ARM to Thumb instructions.

```

 AREA ChangeState, CODE, READONLY
 ARM

 LDR r0,=start+1 ; This section starts in ARM state
 ; Load the address and set the
 ; least significant bit
 BX r0 ; Branch and exchange instruction sets
 ; Not necessarily in same section

 CODE16
start MOV r1,#10 ; Following instructions are old Thumb
 ; old Thumb instructions

```

## 6.8 Miscellaneous directives

This section describes the following directives:

- *ALIGN* on page 6-62
- *AREA* on page 6-64
- *END* on page 6-66
- *ENTRY* on page 6-67
- *EQU* on page 6-68
- *EXPORT* or *GLOBAL* on page 6-69
- *EXPORTAS* on page 6-71
- *EXTERN* on page 6-72
- *GET* or *INCLUDE* on page 6-74
- *IMPORT* on page 6-75
- *INCBIN* on page 6-77
- *KEEP* on page 6-78
- *NOFP* on page 6-79
- *REQUIRE* on page 6-79
- *REQUIRE8* and *PRESERVE8* on page 6-80
- *RN* on page 6-82
- *ROUT* on page 6-83.

### 6.8.1 ALIGN

The ALIGN directive aligns the current location to a specified boundary by padding with zeros.

#### Syntax

```
ALIGN {expr{,offset{,pad }}}
```

where:

|               |                                                                               |
|---------------|-------------------------------------------------------------------------------|
| <i>expr</i>   | is a numeric expression evaluating to any power of 2 from $2^0$ to $2^{31}$ . |
| <i>offset</i> | can be any numeric expression.                                                |
| <i>pad</i>    | can be any numeric expression.                                                |

#### Operation

The current location is aligned to the next address of the form:

$$offset + n * expr$$

If *expr* is not specified, ALIGN sets the current location to the next word (four byte) boundary.

The unused bytes between the previous and the new current location are filled with copies of the least significant byte of *pad*, or zeros if *pad* is not specified.

#### 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.
- LDRD and STRD doubleword data transfers must be eight-byte aligned. Use ALIGN 8 before memory allocation directives such as DCQ (see *Data definition directives* on page 6-13) if the data is to be accessed using LDRD or STRD.

- 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 (see *AREA* on page 6-64 and *Examples*).

## 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

```

```

 AREA OffsetExample, CODE
 DCB 1 ; This example places the two
 ALIGN 4,3 ; bytes in the first and fourth
 DCB 1 ; bytes of the same word.

```

```

 AREA Example, CODE, READONLY
start LDR r6,=label1
 ; code
 MOV pc,lr
label1 DCB 1 ; pc now misaligned
 ALIGN 4 ; ensures that subroutine1 addresses
subroutine1 ; the following instruction.
 MOV r5,#0x5

```

## 6.8.2 AREA

The AREA directive instructs the assembler to assemble a new code or data section. Sections are independent, named, indivisible chunks of code or data that are manipulated by the linker. See *ELF sections and the AREA directive* on page 2-14 for more information.

### Syntax

```
AREA sectionname{,attr}{,attr}...
```

where:

|                    |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |      |                                                         |        |                                                                                                                                                              |
|--------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------|---------------------------------------------------------|--------|--------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>sectionname</i> | <p>is the name that the section is to be given.</p> <p>You can choose any name for your sections. However, names starting with a digit must be enclosed in bars or a missing section name error is generated. For example, <code> 1_DataArea </code>.</p> <p>Certain names are conventional. For example, <code> .text </code> is used for code sections produced by the C compiler, or for code sections otherwise associated with the C library.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |      |                                                         |        |                                                                                                                                                              |
| <i>attr</i>        | <p>are one or more comma-delimited section attributes. Valid attributes are:</p> <p><i>ALIGN=expression</i></p> <p>By default, ELF sections are aligned on a four-byte boundary. <i>expression</i> can have any integer value from 0 to 31. The section is aligned on a <math>2^{\text{expression}}</math>-byte boundary. For example, if <i>expression</i> is 10, the section is aligned on a 1KB boundary.</p> <p><i>This is not the same as the way that the ALIGN directive is specified. See ALIGN on page 6-62.</i></p> <p style="text-align: center;"><b>Note</b></p> <p>Do not use <code>ALIGN=0</code> or <code>ALIGN=1</code> for code sections.</p> <p><i>ASSOC=section</i></p> <p><i>section</i> specifies an associated ELF section. <i>sectionname</i> must be included in any link that includes <i>section</i></p> <tr> <td>CODE</td> <td>Contains machine instructions. READONLY is the default.</td> </tr> <tr> <td>COMDEF</td> <td>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.</td> </tr> | CODE | Contains machine instructions. READONLY is the default. | 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. |
| CODE               | Contains machine instructions. READONLY is the default.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |      |                                                         |        |                                                                                                                                                              |
| 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. See the chapter describing basic linker functionality in *RealView Compilation Tools v2.2 Linker and Utilities Guide*.

|           |                                                                                                                                                                                                                                                                                                                                                                                                                           |
|-----------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 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.                                                                                                                                                                                                                                                                                                                                                                |
| 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 (see the chapter describing basic linker functionality in <i>RealView Compilation Tools v2.2 Linker and Utilities Guide</i> ). |
| READONLY  | Indicates that this section should 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.                                                                                                                                                                                                                                                                                                                          |

## 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.

You should normally use separate ELF sections for code and data. 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 local labels is defined by AREA directives, optionally subdivided by ROUT directives (see *Local labels* on page 3-22 and *ROUT* on page 6-83).

There must be at least one AREA directive for an assembly.

### Example

The following example defines a read-only code section named Example.

```
AREA Example, CODE, READONLY ; An example code section.
; code
```

## 6.8.3 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. See *GET or INCLUDE* on page 6-74 for more information.

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.

## 6.8.4 ENTRY

The ENTRY directive declares an entry point to a program.

### Syntax

ENTRY

### Usage

You must specify at least one ENTRY point for a program. If no ENTRY exists, a warning is generated at link time.

You must not use more than one ENTRY directive in a single source file. Not every source file has to have an ENTRY directive. If more than one ENTRY exists in a single source file, an error message is generated at assembly time.

### Example

```
AREA ARMex, CODE, READONLY
ENTRY ; Entry point for the application
```

## 6.8.5 EQU

The EQU directive gives a symbolic name to a numeric constant, a register-relative value or a program-relative value. \* is a synonym for EQU.

### Syntax

*name* EQU *expr*{, *type*}

where:

*name* is the symbolic name to assign to the value.

*expr* is a register-relative address, a program-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 will be 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.

See *KEEP* on page 6-78 and *EXPORT* or *GLOBAL* on page 6-69 for information on exporting symbols.

### 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
```

## 6.8.6 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 {symbol}{[WEAK,attr]}
```

where:

*symbol* is the symbol name to export. The symbol name is case-sensitive. If *symbol* is omitted, all symbols are exported.

[WEAK] means that this instance of *symbol* should only be imported into other sources if no other source exports an alternative instance. If [WEAK] is used without *symbol*, all exported symbols are weak.

[*attr*] is symbol visibility when linked into a dynamic component. By default, symbol binding defines visibility, that is, global and weak symbols are visible to other components and local symbols are not hidden.

Valid attributes are:

DYNAMIC *symbol* is visible to other components, and can be redefined by other components.

HIDDEN *symbol* cannot be referenced outside the component where it is defined, either directly or indirectly.

The linker also accepts INTERNAL and currently treats it as HIDDEN. If you specify both, for example:

```
EXPORT SymA[WEAK,INTERNAL,HIDDEN]
```

the assembler chooses the most restrictive (INTERNAL).

PROTECTED *symbol* is visible to other components, and cannot be redefined by other components.

## 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.

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.
```

See also *IMPORT* on page 6-75.

## Example

```
AREA Example, CODE, READONLY
EXPORT DoAdd ; Export the function name
 ; to be used by external
 ; modules.
DoAdd ADD r0, r0, r1
```

## 6.8.7 EXPORTAS

The EXPORTAS directive enables you to export a symbol to 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.

See also *EXPORT* or *GLOBAL* on page 6-69.

### 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 will
 ;; appear in the object file string table as data1.

one EQU 2
EXPORTAS one, two
EXPORT one ;; the symbol 'two' will appear in the object
 ;; file's symbol table with the value 2.
```

### 6.8.8 EXTERN

The EXTERN directive provides the assembler with a name that is not defined in the current assembly.

EXTERN is very similar to IMPORT, except that the name is not imported if no reference to it is found in the current assembly (see *IMPORT* on page 6-75, and *EXPORT* or *GLOBAL* on page 6-69).

#### Syntax

```
EXTERN symbol {[WEAK,attr]}
```

where:

*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*] is symbol visibility when linked into a dynamic component. By default, the symbol binding defines visibility, that is, global and weak symbols are visible to other external objects and local symbols are not hidden.

Valid attributes are:

DYNAMIC *symbol* is visible to other components, and can be redefined by other components.

HIDDEN *symbol* cannot be referenced outside the component where it is defined, either directly or indirectly.

The linker also accepts INTERNAL and currently treats it as HIDDEN. If you specify both, for example:

```
EXTERN SymA[WEAK,INTERNAL,HIDDEN]
```

the assembler chooses the most restrictive (INTERNAL).

PROTECTED *symbol* is visible to other components, and cannot be redefined by other components.



## 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

This 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.
```

### 6.8.9 GET or INCLUDE

The GET directive includes a file within the file being assembled. The included file is assembled at the location of the GET directive. INCLUDE is a synonym for GET.

#### Syntax

GET *filename*

where:

*filename* is the name of the file to be included in the assembly. The assembler accepts pathnames in either UNIX or MS-DOS format.

#### Usage

GET is useful for including macro definitions, EQUs, and storage maps in an assembly. When assembly of the included file is complete, assembly continues at the line following the GET directive.

By default the assembler searches the current place for included files. The current place is the directory where the calling file is located. Use the -i assembler command-line option to add directories to the search path. File names and directory names containing spaces must not be enclosed in double quotes ( " " ).

The included file can contain additional GET directives to include other files (see *Nesting directives* on page 6-28).

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.

GET cannot be used to include object files (see *INCBIN* on page 6-77).

#### Example

```
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
```

## 6.8.10 IMPORT

The `IMPORT` directive provides the assembler with a name that is not defined in the current assembly.

`IMPORT` is very similar to `EXTERN`, except that the name is imported whether or not it is referred to in the current assembly (see *EXTERN* on page 6-72, and *EXPORT* or *GLOBAL* on page 6-69).

### Syntax

```
IMPORT symbol {[WEAK,attr]}
```

where:

|                 |                                                                                                                                                                                                              |
|-----------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>symbol</i>   | 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.                                       |
| [ <i>attr</i> ] | is symbol visibility when linked into a dynamic component. By default, symbol binding defines visibility, that is, global and weak symbols are visible to other components and local symbols are not hidden. |

Valid attributes are:

|           |                                                                                                                                                                                                                                                                                                                                                                             |
|-----------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| DYNAMIC   | <i>symbol</i> is visible to other components, and can be redefined by other components.                                                                                                                                                                                                                                                                                     |
| HIDDEN    | <i>symbol</i> cannot be referenced outside the component where it is defined, either directly or indirectly.<br>The linker also accepts <code>INTERNAL</code> and currently treats it as <code>HIDDEN</code> . If you specify both, for example:<br><code>IMPORT SymA[WEAK,INTERNAL,HIDDEN]</code><br>the assembler chooses the most restrictive ( <code>INTERNAL</code> ). |
| PROTECTED | <i>symbol</i> is visible to other components, and cannot be redefined by other components.                                                                                                                                                                                                                                                                                  |

## 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.

To avoid trying to access symbols that are not found at link time, use code like the example in *EXTERN* on page 6-72.

## 6.8.11 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
```

### 6.8.12 KEEP

The KEEP directive instructs the assembler to retain local symbols in the symbol table in the object file.

#### Syntax

```
KEEP {symbol}
```

where:

*symbol* is the name of the local symbol to keep. If *symbol* is not specified, all local symbols are kept except register-relative symbols.

#### Usage

By default, the only symbols that the assembler describes in its output object file are:

- exported symbols
- symbols that are relocated against.

Use KEEP to preserve local symbols that can be used to help debugging. Kept symbols appear in the ARM debuggers and in linker map files.

KEEP cannot preserve register-relative symbols (see *MAP* on page 6-16).

#### Example

```
label ADC r2,r3,r4
 KEEP label ; makes label available to debuggers
 ADD r2,r2,r5
```

### 6.8.13 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.

### 6.8.14 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.

### 6.8.15 REQUIRE8 and PRESERVE8

The REQUIRE8 directive specifies that the current file requires eight-byte alignment of the stack. It sets the REQ8 build attribute to inform the linker.

The PRESERVE8 directive specifies that the current file preserves eight-byte alignment of the stack. It sets the PRES8 build attribute to inform the linker.

The linker ensures 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.

#### Syntax

```
REQUIRE8 {bool}
```

```
PRESERVE8 {bool}
```

where:

*bool* is an optional Boolean constant, either {TRUE} or {FALSE}.

#### Usage

If you are using unaligned data accesses in ARMv6 systems, LDRD and STRD instructions (doubleword transfers) do not require eight-byte alignment. Instead, they require four-byte alignment.

However, on other systems, LDRD and STRD instructions only work correctly if the address they access is eight-byte aligned.

Where required, if your code includes LDRD or STRD transfers to or from the stack, use REQUIRE8 to set the REQ8 build attribute on your file. The assembler gives a warning if you use LDRD or STRD addressed sp relative without specifying REQUIRE8.

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.

#### ————— 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 with:

```
armasm --diag_warning 1546
```



See *Command syntax* on page 3-2 for details.

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

```

## 6.8.16 RN

The RN directive defines a register 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 listed in *Predefined register and coprocessor names* on page 3-15.

*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
```

## 6.8.17 ROUT

The ROUT directive marks the boundaries of the scope of local labels (see *Local labels* on page 3-22).

### Syntax

```
{name} ROUT
```

where:

*name* is the name to be assigned to the scope.

### Usage

Use the ROUT directive to limit the scope of local labels. This makes it easier for you to avoid referring to a wrong label by accident. The scope of local labels is the whole area if there are no ROUT directives in it (see *AREA* on page 6-64).

Use the *name* option to ensure that each reference is to the correct 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

```

routineA ; code
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

```



# Glossary

The items in this glossary are listed in alphabetical order, with any symbols and numerics appearing at the end.

**AAPCS** *See* Procedure Call Standard for The ARM Architecture.

**ABI for the ARM Architecture (base standard) (BSABI)**

The ABI for the ARM Architecture is a collection of specifications, some open and some specific to ARM architecture, that regulate the inter-operation of binary code in a range of ARM architecture-based execution environments. The base standard specifies those aspects of code generation that must be standardized to support inter-operation and is aimed at authors and vendors of C and C++ compilers, linkers, and runtime libraries.

**American National Standards Institute (ANSI)**

An organization that specifies standards for, among other things, computer software. This is superseded by the International Standards Organization.

**ANSI** *See* American National Standards Institute.

**Architecture** The term used to identify a group of processors that have similar characteristics.

**ARM instruction** A word that encodes an operation for an ARM processor operating in ARM state. ARM instructions must be word-aligned.

|                                                   |                                                                                                                                                                                                                                                                                                       |
|---------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>ARM state</b>                                  | <p>A processor that is executing ARM instructions is operating in ARM state. The processor switches to Thumb state (and to recognizing Thumb instructions) when directed to do so by a state-changing instruction such as BX, BLX.</p> <p><i>See also</i> Thumb state.</p>                            |
| <b>Big-endian</b>                                 | <p>Memory organization where the least significant byte of a word is at a higher address than the most significant byte.</p>                                                                                                                                                                          |
| <b>Byte</b>                                       | <p>A unit of memory storage consisting of eight bits.</p>                                                                                                                                                                                                                                             |
| <b>Canonical Frame Address (CFA)</b>              | <p>In DWARF, this is an address on the stack specifying where the call frame of an interrupted function is located.</p>                                                                                                                                                                               |
| <b>CFA</b>                                        | <p><i>See</i> Canonical Frame Address.</p>                                                                                                                                                                                                                                                            |
| <b>Coprocessor</b>                                | <p>An additional processor that is used for certain operations. Usually used for floating-point math calculations, signal processing, or memory management.</p>                                                                                                                                       |
| <b>CPSR</b>                                       | <p><i>See</i> Current Processor Status Register.</p>                                                                                                                                                                                                                                                  |
| <b>Current Processor Status Register (CPSR)</b>   | <p>A register containing the current state of control bits and flags.</p> <p><i>See also</i> Saved Processor Status Register.</p>                                                                                                                                                                     |
| <b>Debug With Arbitrary Record Format (DWARF)</b> | <p>ARM code generation tools generate debug information in DWARF2 format by default. From RVCT v2.2, you can optionally generate DWARF3 format (Draft Standard 9).</p>                                                                                                                                |
| <b>Debugger</b>                                   | <p>An application that monitors and controls the execution of a second application. Usually used to find errors in the application program flow.</p>                                                                                                                                                  |
| <b>Deprecated</b>                                 | <p>A deprecated option or feature is one that you are strongly discouraged from using. Deprecated options and features will not be supported in future versions of the product.</p>                                                                                                                   |
| <b>Doubleword</b>                                 | <p>A 64-bit unit of information. Contents are taken as being an unsigned integer unless otherwise stated.</p>                                                                                                                                                                                         |
| <b>DWARF</b>                                      | <p><i>See</i> Debug With Arbitrary Record Format.</p>                                                                                                                                                                                                                                                 |
| <b>ELF</b>                                        | <p><i>See</i> Executable and Linking Format.</p>                                                                                                                                                                                                                                                      |
| <b>Executable and Linking Format (ELF)</b>        | <p>The industry standard binary file format used by RealView Compilation Tools. ELF object format is produced by the ARM object producing tools such as armcc and armasm. The ARM linker accepts ELF object files and can output either an ELF executable file, or a partially linked ELF object.</p> |

|                                                   |                                                                                                                                                                                                                                                                            |
|---------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Global variables</b>                           | Variables that are accessible to all code in the application.<br><br><i>See also</i> Local variables.                                                                                                                                                                      |
| <b>Halfword</b>                                   | A 16-bit unit of information. Contents are taken as being an unsigned integer unless otherwise stated.                                                                                                                                                                     |
| <b>Hint</b>                                       | A hint instruction provides information to the hardware that the hardware can take advantage of. An implementation can choose whether to implement hint instructions or not. If they are not implemented, they execute as NOP.                                             |
| <b>Image</b>                                      | An executable file that has been loaded onto a processor for execution.<br><br>A binary execution file loaded onto a processor and given a thread of execution. An image can have multiple threads. An image is related to the processor on which its default thread runs. |
| <b>Implementation defined</b>                     | Means that the behavior is not architecturally defined, but must be defined and documented by individual implementations.                                                                                                                                                  |
| <b>International Standards Organization (ISO)</b> | An organization that specifies standards for, among other things, computer software. This supersedes the American National Standards Institute.                                                                                                                            |
| <b>Interrupt</b>                                  | A change in the normal processing sequence of an application caused by, for example, an external signal.                                                                                                                                                                   |
| <b>Interworking</b>                               | Producing an application that uses both ARM and Thumb code.                                                                                                                                                                                                                |
| <b>ISO</b>                                        | <i>See</i> International Standards Organization.                                                                                                                                                                                                                           |
| <b>IT block</b>                                   | A block of up to four instructions following an <i>If-Then</i> (IT) instruction. Each instruction in the block is conditional. The conditions for the instructions are either all the same, or some can be the inverse of others.                                          |
| <b>Jazelle</b>                                    | The Jazelle architecture extends the existing ARM architecture to enable direct execution of selected JVM (Java Virtual Machine) opcodes.                                                                                                                                  |
| <b>Library</b>                                    | A collection of assembler or compiler output objects grouped together into a single repository.                                                                                                                                                                            |
| <b>Linker</b>                                     | Software that produces a single image from one or more source assembler or compiler output objects.                                                                                                                                                                        |
| <b>Little-endian</b>                              | Memory organization where the least significant byte of a word is at a lower address than the most significant byte.                                                                                                                                                       |

|                                                                 |                                                                                                                                                                                                           |
|-----------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Local variable</b>                                           | A variable that is only accessible to the subroutine that created it.<br><br><i>See also</i> Global variables.                                                                                            |
| <b>Memory hint</b>                                              | A memory hint instruction enables you to provide advance information to memory systems about future memory accesses, without actually loading or storing any data.                                        |
| <b>MPCore</b>                                                   | An integrated Symmetric Multiprocessor system (SMP) delivered as a traditional uniprocessor core. The chip contains up to four ARM1136J-S based CPUs with cache coherency.                                |
| <b>PIC</b>                                                      | Position Independent Code.<br><br><i>See also</i> ROPI.                                                                                                                                                   |
| <b>PID</b>                                                      | Position Independent Data.<br><br><i>See also</i> RWPI.                                                                                                                                                   |
| <b>Procedure Call Standard for the ARM Architecture (AAPCS)</b> | <i>Procedure Call Standard for the ARM Architecture</i> defines how registers and the stack will be used for subroutine calls.                                                                            |
| <b>Processor Status Register (PSR)</b>                          | A register containing various control bits and flags.<br><br><i>See also</i> Current Processor Status Register<br><i>See also</i> Saved Processor Status Register.                                        |
| <b>PSR</b>                                                      | <i>See</i> Processor Status Register                                                                                                                                                                      |
| <b>Read-Only Position Independent (ROPI)</b>                    | Code and read-only data addresses can be changed at runtime.                                                                                                                                              |
| <b>Read Write Position Independent (RWPI)</b>                   | Read/write data addresses can be changed at runtime.                                                                                                                                                      |
| <b>RealView Compilation Tools (RVCT)</b>                        | RealView Compilation Tools is a suite of tools, together with supporting documentation and examples, that enables you to write and build applications for the ARM family of <i>RISC</i> processors.       |
| <b>RealView Developer Suite (RVDS)</b>                          | The latest suite of software development applications, together with supporting documentation and examples, that enable you to write and debug applications for the ARM family of <i>RISC</i> processors. |
| <b>ROPI</b>                                                     | <i>See</i> Read-Only Position Independent.                                                                                                                                                                |



|                                                 |                                                                                                                                                                                                                                                                                                                 |
|-------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>RVCT</b>                                     | <i>See</i> RealView Compilation Tools.                                                                                                                                                                                                                                                                          |
| <b>RVDS</b>                                     | <i>See</i> RealView Developer Suite.                                                                                                                                                                                                                                                                            |
| <b>RWPI</b>                                     | <i>See</i> Read Write Position Independent.                                                                                                                                                                                                                                                                     |
| <b>Saved Processor Status Register (SPSR)</b>   | SPSR. A register that holds a copy of what was in the Current Processor Status Register before the most recent exception. Each exception mode has its own SPSR.                                                                                                                                                 |
| <b>Scope</b>                                    | The accessibility of a function or variable at a particular point in the application code. Symbols that have global scope are always accessible. Symbols with local or private scope are only accessible to code in the same subroutine or object.                                                              |
| <b>Section</b>                                  | A block of software code or data for an Image.                                                                                                                                                                                                                                                                  |
| <b>Semihosting</b>                              | A mechanism whereby the target communicates I/O requests made in the application code to the host system, rather attempting to support the I/O itself.                                                                                                                                                          |
| <b>SIMD</b>                                     | <i>See</i> Single Instruction, Multiple Data.                                                                                                                                                                                                                                                                   |
| <b>Single Instruction, Multiple Data (SIMD)</b> | Single Instruction, Multiple Data (SIMD) instructions perform similar operations on four 8-bit, or two 16-bit, data items held in 32-bit registers.                                                                                                                                                             |
| <b>Software Interrupt (SWI)</b>                 | An instruction that causes the processor to call a programmer-specified subroutine. Used by ARM to handle semihosting.                                                                                                                                                                                          |
| <b>SPSR</b>                                     | <i>See</i> Saved Processor Status Register.                                                                                                                                                                                                                                                                     |
| <b>Stack</b>                                    | The portion of computer memory that is used to record the address of code that calls a subroutine. The stack can also be used for parameters and temporary variables.                                                                                                                                           |
| <b>SWI</b>                                      | <i>See</i> Software Interrupt.                                                                                                                                                                                                                                                                                  |
| <b>Target</b>                                   | The actual target processor, (real or simulated), on which the target application is running.<br><br>The fundamental object in any debugging session. The basis of the debugging system. The environment in which the target software will run. It is essentially a collection of real or simulated processors. |
| <b>Thumb instruction</b>                        | One halfword or two halfwords that encode an operation for an ARM processor operating in Thumb state. Thumb instructions must be halfword-aligned.                                                                                                                                                              |

|                                    |                                                                                                                                                                                                                                                                   |
|------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Thumb state</b>                 | A processor that is executing Thumb instructions is operating in Thumb state. The processor switches to ARM state (and to recognizing ARM instructions) when directed to do so by a state-changing instruction such as BX, BLX.<br><br><i>See also</i> ARM state. |
| <b>TrustZone</b>                   | ARM technology-optimized software that provides a secure execution environment to enable trusted programs and data to be separated from the operating system and applications.                                                                                    |
| <b>Undefined</b>                   | An attempt to execute an instruction that is not permitted causes an Undefined Instruction exception.                                                                                                                                                             |
| <b>Unpredictable</b>               | The result of an instruction that cannot be relied upon. Unpredictable instructions or results must not represent security holes. Unpredictable instructions must not halt or hang the processor, or any parts of the system.                                     |
| <b>Vector Floating Point (VFP)</b> | A standard for floating-point coprocessors where several data values can be processed by a single instruction.                                                                                                                                                    |
| <b>Veneer</b>                      | A small block of code used with subroutine calls when there is a requirement to change processor state or branch to an address that cannot be reached in the current processor state.                                                                             |
| <b>VFP</b>                         | <i>See</i> Vector Floating Point.                                                                                                                                                                                                                                 |
| <b>Word</b>                        | A 32-bit unit of information. Contents are taken as being an unsigned integer unless otherwise stated.                                                                                                                                                            |
| <b>Zero initialized (ZI)</b>       | R/W memory used to hold variables that do not have an initial value. The memory is normally set to zero on reset.                                                                                                                                                 |
| <b>ZI</b>                          | <i>See</i> Zero Initialized.                                                                                                                                                                                                                                      |

# Index

## A

### AAPCS

- about 3-5
- specifying variants 3-5
- /adsabi qualifier 3-6

### ABI 3-5

Absolute addresses 3-21

ADC ARM instruction 4-45

ADC Thumb 16-bit instruction 4-45

ADC Thumb-2 instruction 4-45

ADD ARM instruction 4-45

ADD Thumb 16-bit instruction 4-45

ADD Thumb-2 instruction 4-45

### Addresses

- direct loading with ADR 2-31
- direct loading with ADRL 2-31
- loading into registers 2-31
- loading with LDR 2-34

ADR ARM pseudo-instruction 4-145

ADRL ARM pseudo-instruction 4-147

ALIGN directive 6-62

ALU status flags 2-17, 2-18

AND ARM instruction 4-50

AND Thumb 16-bit instruction 4-50

AND Thumb-2 instruction 4-50

AREA directive 2-12, 2-14, 6-64

AREA directive (literal pools) 2-29

### ARM

instruction set, overview 2-7

state 2-4

ARM directive 6-58

ARM instructions

summary 4-2

conditional execution 2-17

Q flag 2-22

second operand 4-42

ADC 4-45

ADD 4-45

AND 4-50

B 4-116

BFC 4-53

BFI 4-53

BIC 4-50

BKPT 4-132

BL 4-116

CDP 4-122

CDP2 4-122

CLREX 4-39

CLZ 4-54

CMN 4-55

CMP 4-55

CPS 4-137

EOR 4-50

LDC 4-127

LDC2 4-129

LDM 4-27

LDR 4-9, 4-13, 4-16, 4-19, 4-22

LDREX 4-36

MAR 4-143

MCR 4-123

MCR2 4-123

MCR2 4-123

MIA 4-93

MIAPH 4-93

MIAXy 4-93

MLA 4-74

MLS 4-74

- MOV 4-57
- MOVT 4-60
- MRA 4-143
- MRC 4-125
- MRC2 4-125
- MRS 4-134
- MSR 4-135
- MUL 4-74
- MVN 4-57
- ORN 4-50
- ORR 4-50
- PKHBT 4-113
- PKHTB 4-113
- PLD 4-25
- POP 4-30
- PUSH 4-30
- QADD 4-96
- QADDSUBX 4-101
- QADD16 4-101
- QADD8 4-101
- QDADD 4-96
- QDSUB 4-96
- QSUB 4-96
- QSUBADDX 4-101
- QSUB16 4-101
- QSUB8 4-101
- RBIT 4-65
- REV 4-65
- REVSH 4-65
- REV16 4-65
- RFE 4-32
- RSB 4-45
- RSC 4-45
- SADDSUBX 4-101
- SADD16 4-101
- SADD8 4-101
- SBC 4-45
- SBFX 4-109
- SEL 4-63
- SETEND 4-140
- SEV 4-131, 4-141
- SHADDSUBX 4-101
- SHADD16 4-101
- SHADD8 4-101
- SHSUBADDX 4-101
- SHSUB16 4-101
- SHSUB8 4-101
- SMT 4-139
- SMLAD 4-88
- SMLAL 4-76
- SMLALBB 4-82
- SMLALBT 4-82
- SMLALD 4-90
- SMLALTB 4-82
- SMLALTT 4-82
- SMLSD 4-88
- SMLSLD 4-90
- SMMUL 4-86
- SMUAD 4-84
- SMULBB 4-78
- SMULBT 4-78
- SMULL 4-76
- SMULTB 4-78
- SMULTT 4-78
- SMULWB 4-80
- SMULWT 4-80
- SMUSD 4-84
- SRS 4-34
- SSAT 4-98
- SSAT16 4-106
- SSUBADDX 4-101
- SSUB16 4-101
- SSUB8 4-101
- STC 4-127
- STC2 4-129
- STM 4-27
- STR 4-9, 4-13, 4-16, 4-19
- STREX 4-36
- SUB 4-45
- SWI 4-133
- SWP 4-40
- SWPB 4-40
- SXT 4-110
- SXTA 4-110
- TBB 4-120
- TEQ 4-61
- TST 4-61
- UADDSUBX 4-101
- UADD16 4-101
- UADD8 4-101
- UBFX 4-109
- UHADDSUBX 4-101
- UHADD16 4-101
- UHADD8 4-101
- UHSUBADDX 4-101
- UHSUB16 4-101
- UHSUB8 4-101
- UMAAL 4-92
- UMLAL 4-76
- UMULL 4-76
- UQADDSUBX 4-101
- UQADD16 4-101
- UQADD8 4-101
- UQSUBADDX 4-101
- UQSUB16 4-101
- UQSUB8 4-101
- USADA8 4-104
- USAD8 4-104
- USAT 4-98
- USAT16 4-106
- USUBADDX 4-101
- USUB16 4-101
- USUB8 4-101
- UXT 4-110
- UXTA 4-110
- ARM pseudo-instructions
  - ADR 4-145
  - ADRL 4-147
  - LDR 4-151
  - MOV32 4-149
- armasm
  - command syntax 3-2
- ASR Thumb-2 instruction 4-67
- Assembler syntax, diagnostics options 3-11
- Assembler syntax, options
  - apcs 3-2
  - bigend 3-2
  - brief\_diagnostics 3-2
  - checkreglist 3-2
  - cpu 3-2
  - depend 3-3
  - diag\_error 3-3
  - dllexport\_all 3-3
  - dwarf2 3-3
  - dwarf3 3-3
  - errors 3-3
  - exceptions 3-3
  - exceptions\_unwind 3-3
  - fpmode 3-3
  - fpu 3-3
  - g 3-2
  - i 3-3
  - keep 3-3
  - list 3-4
  - littleend 3-2
  - m 3-3

- maxcache 3-4
- md 3-3
- memaccess 3-4
- no\_cache 3-4
- no\_esc 3-4
- no\_exceptions 3-4
- no\_exceptions\_unwind 3-4
- no\_hide\_all 3-4
- no\_regs 3-4
- no\_warn 3-4
- o 3-4
- predefine 3-4
- split\_ldm 3-4
- thumb 3-5
- unsafe 3-5
- via 3-5
- 16 3-2
- 32 3-2
- Assembly language
  - absolute addresses 3-21
  - binary operators 3-31
  - block copy 2-41
  - Boolean constants 2-13
  - built-in variables 3-16
  - case rules 2-11
  - character constants 2-13
  - comments 2-12
  - condition code suffixes 2-18
  - conditional execution 2-17
  - constants 2-13
  - coprocessor names 3-15
  - defining macros 6-29
  - ELF sections 2-14
  - entry point 2-15, 6-67
  - examples 2-2, 2-14, 2-16
  - floating-point literals 3-27
  - format of source lines 3-14
  - global variables 6-5, 6-8
  - jump tables 2-32
  - labels 2-12, 3-21
  - line format 2-11
  - line length 2-11
  - literal pools 2-29
  - loading addresses 2-31
  - loading addresses with LDR 2-34
  - loading addresses, direct loading 2-31
  - loading constants 2-23
  - local labels 2-12, 3-22
  - logical
    - expressions 3-28
    - literals 3-28
    - variables 3-19
  - macros 2-43
  - multiple register transfers 2-37
  - multiplicative operators 3-32
  - nesting subroutines 2-40
  - numeric constants 2-13, 3-19
  - numeric expressions 3-25
  - numeric literals 3-26
  - numeric variables 3-19
  - operator precedence 3-28
  - operator precedence in C 3-29
  - PC 2-6, 2-40, 3-21, 3-28
  - Program Counter 2-6, 3-21, 3-28
  - program-relative 2-12
    - expressions 3-28
  - program-relative labels 3-21
  - register names 3-15
  - register-relative
    - expressions 3-28
    - labels 3-21
  - relational operators 3-34
  - shift operators 3-33
  - stacks 2-39
  - string
    - expressions 3-24
    - manipulation 3-32
    - variables 3-19
  - string constants 2-13
  - string literals 3-25
  - subroutines 2-16
  - symbol naming rules 3-18
  - symbols 3-18
  - unary operators 3-30
  - variable substitution 3-19
  - variables 3-19
    - built-in 3-16
    - global 6-5, 6-8
    - local 6-7, 6-8
  - VFP directives and notation 5-37
- ASSERT directive 6-52
- B**
  - B ARM instruction 4-116
  - B Thumb instruction 2-17
  - B Thumb 16-bit instruction 4-116
  - B (conditional) Thumb-2 instruction 4-116
  - B (unconditional) Thumb-2 instruction 4-116
  - Barrel shifter, ARM 2-10
  - :BASE: operator 3-30
  - BFC ARM instruction 4-53
  - BFC Thumb-2 instruction 4-53
  - BFI ARM instruction 4-53
  - BFI Thumb-2 instruction 4-53
  - BIC ARM instruction 4-50
  - BIC Thumb 16-bit instruction 4-50
  - BIC Thumb-2 instruction 4-50
  - Binary operators, assembly 3-31
  - BKPT ARM instruction 4-132
  - BKPT Thumb 16-bit instruction 4-132
  - BL ARM instruction 2-16, 4-116
  - BL Thumb instruction 2-16
  - BL Thumb 16-bit instruction 4-116
  - BL Thumb-2 instruction 4-116
  - Block copy, ARM state example 2-41
  - Block copy, assembly language 2-41
  - Boolean constants, assembly language 2-13
  - Branch instructions, ARM 2-7
  - BSABI 3-5
- C**
  - Callgraph
    - stack usage rules 6-37
  - Case rules, assembly language 2-11
  - CBNZ Thumb-2 16-bit instruction 4-119
  - CBZ Thumb-2 16-bit instruction 4-119
  - CDP ARM instruction 4-122
  - CDP Thumb-2 instruction 4-122
  - CDP2 ARM instruction 4-122
  - CDP2 Thumb-2 instruction 4-122
  - Character constants, assembly language 2-13
  - :CHR: operator 3-30
  - CLREX ARM instruction 4-39
  - CLREX Thumb-2 instruction 4-39
  - CLZ ARM instruction 4-54
  - CLZ Thumb-2 instruction 4-54
  - CMN ARM instruction 4-55
  - CMN Thumb 16-bit instruction 4-55

- CMN Thumb-2 instruction 4-55
  - CMP ARM instruction 4-55
  - CMP Thumb 16-bit instruction 4-55
  - CMP Thumb-2 instruction 4-55
  - CN directive 6-10
  - CODE16 directive 6-60
  - CODE32 directive 6-58
  - Command syntax
    - armasm 3-2
  - Comments
    - assembly language 2-12
  - COMMON directive 6-27
  - Condition code suffixes 2-18
  - Conditional execution, ARM state
    - 2-17, 2-19
  - Conditional execution, ARM state
    - example 2-20
  - Conditional execution, assembly 2-17
  - Conditional execution, Thumb example
    - 2-22
  - Constants, assembly 2-13
  - Coprocessor instructions, ARM 2-8
  - Coprocessor names, assembly 3-15
  - CP directive 6-11
  - CPS ARM instruction 4-137
  - CPS Thumb 16-bit instruction 4-137
  - CPS Thumb-2 instruction 4-137
  - CPSR 2-6, 2-17, 2-18, 4-27, 4-32, 4-33, 4-46, 4-49, 4-51, 4-58, 4-63
  - Current Program Status Register 2-6
- ## D
- DATA directive 6-27
  - Data processing instructions, ARM 2-8
  - DCB directive 6-19
  - DCDO directive 6-21
  - DCD, DCDU directives 6-20
  - DCFD, DCFDU directives 6-22
  - DCFS, DCFSU directives 6-23
  - DCQ, DCQU directives 6-25
  - DCW, DCWU directives 6-26
  - Default version
    - symbol versioning 2-46
  - Diagnostic messages 3-36
    - arm style 3-11
    - controlling 3-11
    - ide style 3-11
    - severity 3-12
  - Directives
    - ENDP 6-37
    - FRAME 6-37
    - PROC 6-37
  - Directives, assembly language
    - using 2-47
    - summary 6-2
    - nesting 6-28
    - ALIGN 6-62
    - AREA 2-12, 2-14, 6-64
    - AREA (literal pools) 2-29
    - ARM 6-58
    - ASSERT 6-52
    - CN 6-10
    - CODE16 6-59, 6-60
    - CODE32 2-4, 6-58
    - COMMON 6-27
    - CP 6-11
    - DATA 6-27
    - DCB 6-19
    - DCDO 6-21
    - DCD, DCDU 6-20
    - DCFD, DCFDU 6-22
    - DCFS, DCFSU 6-23
    - DCQ, DCQU 6-25
    - DCW, DCWU 6-26
    - DN 6-12
    - ELIF 6-33
    - ELSE 6-33
    - END 2-15, 6-66
    - END (literal pools) 2-29
    - ENDFUNC 6-51
    - ENDIF 6-33
    - ENDP 6-51
    - ENTRY 2-15, 6-67
    - EQU 3-19, 6-68
    - EXPORT 6-69
    - EXPORTAS 6-71
    - EXTERN 6-72
    - FIELD 6-17
    - FRAME ADDRESS 6-39
    - FRAME POP 6-40
    - FRAME PUSH 6-41
    - FRAME REGISTER 6-43
    - FRAME RESTORE 6-44
    - FRAME RETURN ADDRESS 6-45
    - FRAME SAVE 6-46
    - FRAME STATE REMEMBER 6-47
    - FRAME STATE RESTORE 6-48, 6-49
    - FUNCTION 6-50
    - GBLA 3-9, 3-19, 6-5, 6-54
    - GBLL 3-9, 3-19, 6-5, 6-54
    - GBLS 3-9, 3-19, 6-5, 6-54
    - GET 6-74
    - GLOBAL 6-69
    - IF 6-32, 6-33, 6-36
    - IMPORT 6-75
    - INCBIN 6-77
    - INCLUDE 6-74
    - INFO 6-53
    - KEEP 6-78
    - LCLA 3-19, 6-7, 6-54
    - LCLL 3-19, 6-54
    - LCLS 3-19, 6-54
    - LTORG 6-15
    - MACRO 2-43, 6-29
    - MAP 6-16
    - MEND 6-29, 6-54
    - MEXIT 6-32
    - NOFP 6-79
    - OPT 6-54
    - PRESERVE8 6-80
    - PROC 6-50
    - REQUIRE 6-79
    - REQUIRE8 6-80
    - RLIST 6-9
    - RN 6-82
    - ROUT 2-12, 3-22, 3-23, 6-83
    - SETA 3-9, 3-17, 3-19, 6-8, 6-54
    - SETL 3-9, 3-17, 3-19, 6-8, 6-54
    - SETS 3-9, 3-17, 3-19, 6-8, 6-54
    - SN 6-12
    - SPACE 6-18
    - SUBT 6-56
    - TTL 6-56
    - VFPASSERT SCALAR 5-38
    - VFPASSERT VECTOR 5-39
    - WEND 6-36
    - WHILE 6-32, 6-36
    - ! 6-53
    - # 6-17
    - % 6-18
    - & 6-20
    - \* 6-68
    - = 6-19
    - [ 6-33
    - ] 6-33

^ 6-16  
| 6-33  
DN directive 6-12

## E

ELIF directive 6-33  
ELSE directive 6-33  
END directive 2-15, 6-66  
END directive (literal pools) 2-29  
ENDFUNC directive 6-51  
ENDIF directive 6-33  
ENDP  
    directive 6-37  
ENDP directive 6-51  
ENTRY directive 2-15, 6-67  
Entry point  
    assembly 6-67  
Entry point, assembly 2-15  
EOR ARM instruction 4-50  
EOR Thumb 16-bit instruction 4-50  
EOR Thumb-2 instruction 4-50  
EQU directive 3-19, 6-68  
Examples  
    main directory 1-2  
EXPORT directive 6-69  
EXPORTAS directive 6-71  
EXTERN directive 6-72

## F

FABS VFP instruction 5-16  
FADD VFP instruction 5-17  
FCMP VFP instruction 5-18  
FCPY VFP instruction 5-16  
FCVTDS VFP instruction 5-19  
FCVTSV VFP instruction 5-20  
FDIV VFP instruction 5-21  
FIELD directive 6-17  
FLD VFP instruction 5-22  
FLD VFP pseudo-instruction 5-36  
FLDM VFP instruction 5-24  
FMAC VFP instruction 5-26  
FMDHR VFP instruction 5-28  
FMDLR VFP instruction 5-28  
FMDRR VFP instruction 5-27  
FMRDH VFP instruction 5-28

FMRDL VFP instruction 5-28  
FMRRD VFP instruction 5-27  
FMRRS VFP instruction 5-30  
FMRS VFP instruction 5-29  
FMRX VFP instruction 5-31  
FMSC VFP instruction 5-26  
FMSR VFP instruction 5-29  
FMSRR VFP instruction 5-30  
FMSTAT VFP instruction 5-31  
FMUL VFP instruction 5-32  
FMXR VFP instruction 5-31  
FNEG VFP instruction 5-16  
FNMAL VFP instruction 5-26  
FNMSC VFP instruction 5-26  
FNMUL VFP instruction 5-32  
FRAME  
    directive 6-37  
FRAME ADDRESS directive 6-39  
FRAME POP directive 6-40  
FRAME PUSH directive 6-41  
FRAME REGISTER directive 6-43  
FRAME RESTORE directive 6-44  
FRAME RETURN ADDRESS directive 6-45  
FRAME SAVE directive 6-46  
FRAME STATE REMEMBER directive 6-47  
FRAME STATE RESTORE directive 6-48, 6-49  
FSITO VFP instruction 5-33  
FSQRT VFP instruction 5-34  
FST VFP instruction 5-22  
FSTM VFP instruction 5-24  
FSUB VFP instruction 5-17  
FTOSI VFP instruction 5-35  
FTOUI VFP instruction 5-35  
FUITO VFP instruction 5-33  
FUNCTION directive 6-50

## G

GBLA directive 3-9, 3-19, 6-5, 6-54  
GBLL directive 3-9, 3-19, 6-5, 6-54  
GBLS directive 3-9, 3-19, 6-5, 6-54  
GET directive 6-74  
GLOBAL directive 6-69

## H

### Halfwords

in load and store instructions, ARM 2-8

## I

IF directive 6-32, 6-33, 6-36  
IMPORT directive 6-75  
INCBIN directive 6-77  
INCLUDE directive 6-74  
:INDEX: operator 3-30  
INFO directive 6-53  
Installation directory  
    default location ix  
Instruction set  
    ARM, overview 2-7  
Instructions  
    ADC, ARM 4-45  
    ADC, Thumb 16-bit 4-45  
    ADC, Thumb-2 4-45  
    ADD, ARM 4-45  
    ADD, Thumb 16-bit 4-45  
    ADD, Thumb-2 4-45  
    AND, ARM 4-50  
    AND, Thumb 16-bit 4-50  
    AND, Thumb-2 4-50  
    ASR, Thumb-2 4-67  
    B (conditional), Thumb-2 4-116  
    B (unconditional), Thumb-2 4-116  
    BFC, ARM 4-53  
    BFC, Thumb-2 4-53  
    BFI, ARM 4-53  
    BFI, Thumb-2 4-53  
    BIC, ARM 4-50  
    BIC, Thumb 16-bit 4-50  
    BIC, Thumb-2 4-50  
    BKPT, ARM 4-132  
    BKPT, Thumb 16-bit 4-132  
    BL, ARM 4-116  
    BL, Thumb 16-bit 4-116  
    BL, Thumb-2 4-116  
    B, ARM 4-116  
    B, Thumb 16-bit 4-116  
    CBNZ, Thumb-2 16-bit 4-119  
    CBZ, Thumb-2 16-bit 4-119  
    CDP2, ARM 4-122

- CDP2, Thumb-2 4-122
- CDP, ARM 4-122
- CDP, Thumb-2 4-122
- CLREX, ARM 4-39
- CLREX, Thumb-2 4-39
- CLZ, ARM 4-54
- CLZ, Thumb-2 4-54
- CMN, ARM 4-55
- CMN, Thumb 16-bit 4-55
- CMN, Thumb-2 4-55
- CMP, ARM 4-55
- CMP, Thumb 16-bit 4-55
- CMP, Thumb-2 4-55
- CPS, ARM 4-137
- CPS, Thumb 16-bit 4-137
- CPS, Thumb-2 4-137
- EOR, ARM 4-50
- EOR, Thumb 16-bit 4-50
- EOR, Thumb-2 4-50
- FABS, VFP 5-16
- FADD, VFP 5-17
- FCMP, VFP 5-18
- FCPY, VFP 5-16
- FCVTDS, VFP 5-19, 5-20
- FDIV, VFP 5-21
- FLDM, VFP 5-24
- FLD, VFP 5-22
- FMAC, VFP 5-26
- FMDHR, VFP 5-28
- FMDLR, VFP 5-28
- FMDRR, VFP 5-27
- FMRDH, VFP 5-28
- FMRDL, VFP 5-28
- FMRRD, VFP 5-27
- FMRRS, VFP 5-30
- FMRS, VFP 5-29
- FMRX, VFP 5-31
- FMSC, VFP 5-26
- FMSRR, VFP 5-30
- FMSR, VFP 5-29
- FMSTAT, VFP 5-31
- FMUL, VFP 5-32
- FMXR, VFP 5-31
- FNEG, VFP 5-16
- FNMAC, VFP 5-26
- FNMSC, VFP 5-26
- FNMUL, VFP 5-32
- FSITO, VFP 5-33
- FSQRT, VFP 5-34
- FSTM, VFP 5-24
- FST, VFP 5-22
- FSUB, VFP 5-17
- FTOSI, VFP 5-35
- FTOUI, VFP 5-35
- FUITO, VFP 5-33
- IT, Thumb-2 16-bit 4-70
- LDC2, ARM 4-129
- LDC2, Thumb-2 4-129
- LDC, ARM 4-127
- LDC, Thumb-2 4-127
- LDM, ARM 4-27
- LDM, Thumb-2 4-27
- LDREX, ARM 4-36
- LDREX, Thumb-2 4-36
- LDR, ARM 4-9, 4-13, 4-16, 4-19, 4-22
- LSL, Thumb-2 4-67
- LSR, Thumb-2 4-67
- MAR, ARM 4-143
- MCRR2, ARM 4-123
- MCRR2, Thumb-2 4-123
- MCRR, ARM 4-123
- MCRR, Thumb-2 4-123
- MCR2, ARM 4-123
- MCR2, Thumb-2 4-123
- MCR, ARM 4-123
- MCR, Thumb-2 4-123
- MIAPH, ARM 4-93
- MIAXy, ARM 4-93
- MIA, ARM 4-93
- MLA, ARM 4-74
- MLA, Thumb-2 4-74
- MLS, ARM 4-74
- MLS, Thumb-2 4-74
- MOVt, ARM 4-60
- MOVt, Thumb-2 4-60
- MOV, ARM 4-57
- MOV, Thumb 16-bit 4-57
- MOV, Thumb-2 4-57
- MRA, ARM 4-143
- MRC2, ARM 4-125
- MRC2, Thumb-2 4-125
- MRC, ARM 4-125
- MRC, Thumb-2 4-125
- MRS, ARM 4-134
- MRS, Thumb-2 4-134
- MSR, ARM 4-135
- MSR, Thumb-2 4-135
- MUL, ARM 4-74
- MUL, Thumb 16-bit 4-74
- MUL, Thumb-2 4-74
- MVN, ARM 4-57
- MVN, Thumb 16-bit 4-57
- MVN, Thumb-2 4-57
- ORN, ARM 4-50
- ORN, Thumb-2 4-50
- ORR, ARM 4-50
- ORR, Thumb 16-bit 4-50
- ORR, Thumb-2 4-50
- PKHBT, ARM 4-113
- PKHBT, Thumb-2 4-113
- PKHTB, ARM 4-113
- PKHTB, Thumb-2 4-113
- PLD, ARM 4-25
- POP, ARM 4-30
- POP, Thumb-2 4-30
- PUSH, ARM 4-30
- PUSH, Thumb-2 4-30
- QADDSUBX, ARM 4-101
- QADDSUBX, Thumb-2 4-101
- QADD16, ARM 4-101
- QADD16, Thumb-2 4-101
- QADD8, ARM 4-101
- QADD8, Thumb-2 4-101
- QADD, ARM 4-96
- QADD, Thumb-2 4-96
- QDADD, ARM 4-96
- QDADD, Thumb-2 4-96
- QDSUB, ARM 4-96
- QDSUB, Thumb-2 4-96
- QSUBADDX, ARM 4-101
- QSUBADDX, Thumb-2 4-101
- QSUB16, ARM 4-101
- QSUB16, Thumb-2 4-101
- QSUB8, ARM 4-101
- QSUB8, Thumb-2 4-101
- QSUB, ARM 4-96
- QSUB, Thumb-2 4-96
- RBIT, ARM 4-65
- RBIT, Thumb-2 4-65
- REVSH, ARM 4-65
- REVSH, Thumb 16-bit 4-65
- REVSH, Thumb-2 4-65
- REV16, ARM 4-65
- REV16, Thumb 16-bit 4-65
- REV16, Thumb-2 4-65
- REV, ARM 4-65



- REV, Thumb 16-bit 4-65
- REV, Thumb-2 4-65
- RFE, ARM 4-32
- RFE, Thumb-2 4-32
- ROR, Thumb-2 4-67
- RRX, Thumb-2 4-67
- RSB, ARM 4-45
- RSB, Thumb-2 4-45
- RSC, ARM 4-45
- SADDSUBX, ARM 4-101
- SADDSUBX, Thumb-2 4-101
- SADD16, ARM 4-101
- SADD16, Thumb-2 4-101
- SADD8, ARM 4-101
- SADD8, Thumb-2 4-101
- SBC, ARM 4-45
- SBC, Thumb 16-bit 4-45
- SBC, Thumb-2 4-45
- SBFX, ARM 4-109
- SBFX, Thumb-2 4-109
- SEL, ARM 4-63
- SEL, Thumb-2 4-63
- SETEND, ARM 4-140
- SETEND, Thumb 16-bit 4-140
- SEV, ARM 4-131, 4-141
- SEV, Thumb-2 4-131, 4-141
- SEV, Thumb-2 16-bit 4-141
- SHADDSUBX, ARM 4-101
- SHADDSUBX, Thumb-2 4-101
- SHADD16, ARM 4-101
- SHADD16, Thumb-2 4-101
- SHADD8, ARM 4-101
- SHADD8, Thumb-2 4-101
- SHSUBADDX, ARM 4-101
- SHSUBADDX, Thumb-2 4-101
- SHSUB16, ARM 4-101
- SHSUB16, Thumb-2 4-101
- SHSUB8, ARM 4-101
- SHSUB8, Thumb-2 4-101
- SMI, ARM 4-139
- SMI, Thumb-2 4-139
- SMLAD, ARM 4-88
- SMLAD, Thumb-2 4-88
- SMLALBB, ARM 4-82
- SMLALBB, Thumb-2 4-82
- SMLALBT, ARM 4-82
- SMLALBT, Thumb-2 4-82
- SMLALD, ARM 4-90
- SMLALD, Thumb-2 4-90
- SMLALTB, ARM 4-82
- SMLALTB, Thumb-2 4-82
- SMLALTT, ARM 4-82
- SMLALTT, Thumb-2 4-82
- SMLAL, ARM 4-76
- SMLAL, Thumb-2 4-76
- SMLSD, ARM 4-88
- SMLSD, Thumb-2 4-88
- SMLSLD, ARM 4-90
- SMLSLD, Thumb-2 4-90
- SMMUL, ARM 4-86
- SMMUL, Thumb-2 4-86
- SMUAD, ARM 4-84
- SMUAD, Thumb-2 4-84
- SMULBB, ARM 4-78
- SMULBB, Thumb-2 4-78
- SMULBT, ARM 4-78
- SMULBT, Thumb-2 4-78
- SMULL, ARM 4-76
- SMULL, Thumb-2 4-76
- SMULTB, ARM 4-78
- SMULTB, Thumb-2 4-78
- SMULTT, ARM 4-78
- SMULTT, Thumb-2 4-78
- SMULWB, ARM 4-80
- SMULWB, Thumb-2 4-80
- SMULWT, ARM 4-80
- SMULWT, Thumb-2 4-80
- SMUSD, ARM 4-84
- SMUSD, Thumb-2 4-84
- SRS, ARM 4-34
- SRS, Thumb-2 4-34
- SSAT16, ARM 4-106
- SSAT16, Thumb-2 4-106
- SSAT, ARM 4-98
- SSAT, Thumb-2 4-98
- SSUBADDX, ARM 4-101
- SSUBADDX, Thumb-2 4-101
- SSUB16, ARM 4-101
- SSUB16, Thumb-2 4-101
- SSUB8, ARM 4-101
- SSUB8, Thumb-2 4-101
- STC2, ARM 4-129
- STC2, Thumb-2 4-129
- STC, ARM 4-127
- STC, Thumb-2 4-127
- STM, ARM 4-27
- STM, Thumb-2 4-27
- STREX, ARM 4-36
- STREX, Thumb-2 4-36
- STR, ARM 4-9, 4-13, 4-16, 4-19
- SUBS PC, LR, Thumb-2 4-49
- SUB, ARM 4-45
- SUB, Thumb 16-bit 4-45
- SUB, Thumb-2 4-45
- SWI, ARM 4-133
- SWI, Thumb 16-bit 4-133
- SWI, Thumb-2 4-133
- SWPB, ARM 4-40
- SWP, ARM 4-40
- SXTA, ARM 4-110
- SXTA, Thumb-2 4-110
- SXT, ARM 4-110
- SXT, Thumb 16-bit 4-110
- SXT, Thumb-2 4-110
- TBB, ARM 4-120
- TBB, Thumb-2 4-120
- TBH, Thumb-2 4-120
- TEQ, ARM 4-61
- TEQ, Thumb-2 4-61
- TST, ARM 4-61
- TST, Thumb 16-bit 4-61
- TST, Thumb-2 4-61
- UADDSUBX, ARM 4-101
- UADDSUBX, Thumb-2 4-101
- UADD16, ARM 4-101
- UADD16, Thumb-2 4-101
- UADD8, ARM 4-101
- UADD8, Thumb-2 4-101
- UBFX, ARM 4-109
- UBFX, Thumb-2 4-109
- UHADDSUBX, ARM 4-101
- UHADDSUBX, Thumb-2 4-101
- UHADD16, ARM 4-101
- UHADD16, Thumb-2 4-101
- UHADD8, ARM 4-101
- UHADD8, Thumb-2 4-101
- UHSUBADDX, ARM 4-101
- UHSUBADDX, Thumb-2 4-101
- UHSUB16, ARM 4-101
- UHSUB16, Thumb-2 4-101
- UHSUB8, ARM 4-101
- UHSUB8, Thumb-2 4-101
- UMAAL, ARM 4-92
- UMAAL, Thumb-2 4-92
- UMLAL, ARM 4-76
- UMLAL, Thumb-2 4-76
- UMULL, ARM 4-76

UMULL, Thumb-2 4-76  
 UQADDSUBX, ARM 4-101  
 UQADDSUBX, Thumb-2 4-101  
 UQADD16, ARM 4-101  
 UQADD16, Thumb-2 4-101  
 UQADD8, ARM 4-101  
 UQADD8, Thumb-2 4-101  
 UQSUBADDX, ARM 4-101  
 UQSUBADDX, Thumb-2 4-101  
 UQSUB16, ARM 4-101  
 UQSUB16, Thumb-2 4-101  
 UQSUB8, ARM 4-101  
 UQSUB8, Thumb-2 4-101  
 USADA8, ARM 4-104  
 USADA8, Thumb-2 4-104  
 USAD8, ARM 4-104  
 USAD8, Thumb-2 4-104  
 USAT16, ARM 4-106  
 USAT16, Thumb-2 4-106  
 USAT, ARM 4-98  
 USAT, Thumb-2 4-98  
 USUBADDX, ARM 4-101  
 USUBADDX, Thumb-2 4-101  
 USUB16, ARM 4-101  
 USUB16, Thumb-2 4-101  
 USUB8, ARM 4-101  
 USUB8, Thumb-2 4-101  
 UXTA, ARM 4-110  
 UXTA, Thumb-2 4-110  
 UXT, ARM 4-110  
 UXT, Thumb 16-bit 4-110  
 UXT, Thumb-2 4-110  
 Interlocks 3-36  
 IT block 4-70  
 IT Thumb-2 16-bit instruction 4-70

## J

Jump tables, ARM state example 2-32  
 Jump tables, assembly language 2-32

## K

KEEP directive 6-78

## L

Labels, assembly 3-21  
 Labels, assembly language 2-12  
 Labels, local, assembly 3-22  
 LCLA directive 3-19, 6-7, 6-54  
 LCLL directive 3-19, 6-54  
 LCLS directive 3-19, 6-54  
 LDC ARM instruction 4-127  
 LDC Thumb-2 instruction 4-127  
 LDC2 ARM instruction 4-129  
 LDC2 Thumb-2 instruction 4-129  
 LDM ARM instruction 4-27  
 LDM Thumb-2 instruction 4-27  
 LDR ARM instruction 4-9, 4-13, 4-16, 4-19, 4-22  
 LDR ARM pseudo-instruction 4-151  
 LDR pseudo-instruction  
     literal pools 2-29  
 LDREX ARM instruction 4-36  
 LDREX Thumb-2 instruction 4-36  
 :LEFT: operator 3-32  
 :LEN: operator 3-30  
 Line format, assembly language 2-11  
 Line length, assembly language 2-11  
 Link register 2-5, 2-16  
 Linking  
     assembly language labels 2-12  
 Literal pools, ARM state example 2-29  
 Literal pools, assembly language 2-28, 2-29  
 Loading addresses, ARM state example 2-35  
 Loading constants, assembly language 2-23  
 Local  
     labels, assembly 3-22  
     variables, assembly 6-7, 6-8  
 Local labels, assembly language 2-12  
 Logical  
     expressions, assembly 3-28  
     variable, assembly 3-19  
 Logical literals, assembly 3-28  
 LSL Thumb-2 instruction 4-67  
 LSR Thumb-2 instruction 4-67  
 LTORG directive 6-15

## M

MACRO directive 2-43, 6-29  
 Main examples directory 1-2  
 MAP directive 6-16  
 MAR ARM instruction 4-143  
 MCR ARM instruction 4-123  
 MCR Thumb-2 instruction 4-123  
 MCRR ARM instruction 4-123  
 MCRR Thumb-2 instruction 4-123  
 MCRR2 ARM instruction 4-123  
 MCRR2 Thumb-2 instruction 4-123  
 MCR2 ARM instruction 4-123  
 MCR2 Thumb-2 instruction 4-123  
 MEND directive 6-29, 6-54  
 MEXIT directive 6-32  
 MIA ARM instruction 4-93  
 MIAPH ARM instruction 4-93  
 MIAxy ARM instruction 4-93  
 MLA ARM instruction 4-74  
 MLA Thumb-2 instruction 4-74  
 MLS ARM instruction 4-74  
 MLS Thumb-2 instruction 4-74  
 MOV ARM instruction 4-57  
 MOV Thumb 16-bit instruction 4-57  
 MOV Thumb-2 instruction 4-57  
 MOVT ARM instruction 4-60  
 MOVT Thumb-2 instruction 4-60  
 MOV32 ARM pseudo-instruction 4-149  
 MPCore 2-3  
 MRA ARM instruction 4-143  
 MRC ARM instruction 4-125  
 MRC Thumb-2 instruction 4-125  
 MRC2 ARM instruction 4-125  
 MRC2 Thumb-2 instruction 4-125  
 MRS ARM instruction 4-134  
 MRS Thumb-2 instruction 4-134  
 MSR ARM instruction 4-135  
 MSR Thumb-2 instruction 4-135  
 MUL ARM instruction 4-74  
 MUL Thumb 16-bit instruction 4-74  
 MUL Thumb-2 instruction 4-74  
 Multiple register transfers 2-37  
 Multiplicative operators, assembly 3-32  
 MVN ARM instruction 4-57  
 MVN Thumb 16-bit instruction 4-57  
 MVN Thumb-2 instruction 4-57

## N

Nesting directives 6-28  
 Nesting subroutines, assembly language 2-40  
 NOPF directive 6-79  
 NOP-compatible hints  
   SEV, ARM 4-131, 4-141  
   SEV, Thumb-2 4-131, 4-141  
   SEV, Thumb-2 16-bit 4-131, 4-141  
 Numeric constants, assembly 3-19  
 Numeric constants, assembly language 2-13  
 Numeric expressions, assembly 3-25  
 Numeric literals, assembly 3-26  
 Numeric variable, assembly 3-19

## O

Operator precedence, assembly 3-28  
 Operator precedence, assembly, C 3-29  
 OPT directive 6-54  
 ORN ARM instruction 4-50  
 ORN Thumb-2 instruction 4-50  
 ORR ARM instruction 4-50  
 ORR Thumb 16-bit instruction 4-50  
 ORR Thumb-2 instruction 4-50

## P

Parallel instructions 4-100  
 Parameters, assembly macros 2-43  
 PC, assembly language 2-40, 3-21, 3-28  
 pc, assembly language 2-6  
 PKHBT ARM instruction 4-113  
 PKHBT Thumb-2 instruction 4-113  
 PKHTB ARM instruction 4-113  
 PKHTB Thumb-2 instruction 4-113  
 PLD ARM instruction 4-25  
 POP ARM instruction 4-30  
 POP Thumb-2 instruction 4-30  
 PRESERVE8 directive 6-80  
 PROC  
   directive 6-37  
 PROC directive 6-50  
 Processor mode

  about 2-4  
   privileged 2-5  
 Program counter, assembly 3-21, 3-28  
 Program Counter, assembly language 2-6  
 Program-relative  
   expressions 3-28  
   labels 3-21  
 Program-relative address 2-12  
 Prototype statement 2-43  
 Pseudo-instructions  
   ADRL, ARM 4-147  
   ADR, ARM 4-145  
   FLD, VFP 5-36  
   LDR, ARM 4-151  
   MOV32, ARM 4-149  
 Pseudo-instructions, assembly language  
   LDR (literal pools) 2-29  
   LDR (loading constants) 2-28  
 PSR 4-135  
 PUSH ARM instruction 4-30  
 PUSH Thumb-2 instruction 4-30

## Q

QADD ARM instruction 4-96  
 QADD Thumb-2 instruction 4-96  
 QADDSUBX ARM instruction 4-101  
 QADDSUBX Thumb-2 instruction 4-101  
 QADD16 ARM instruction 4-101  
 QADD16 Thumb-2 instruction 4-101  
 QADD8 ARM instruction 4-101  
 QADD8 Thumb-2 instruction 4-101  
 QDADD ARM instruction 4-96  
 QDADD Thumb-2 instruction 4-96  
 QDSUB ARM instruction 4-96  
 QDSUB Thumb-2 instruction 4-96  
 QSUB ARM instruction 4-96  
 QSUB Thumb-2 instruction 4-96  
 QSUBADDX ARM instruction 4-101  
 QSUBADDX Thumb-2 instruction 4-101  
 QSUB16 ARM instruction 4-101  
 QSUB16 Thumb-2 instruction 4-101  
 QSUB8 ARM instruction 4-101  
 QSUB8 Thumb-2 instruction 4-101

## R

RBIT ARM instruction 4-65  
 RBIT Thumb-2 instruction 4-65  
 :RCONST: operator 3-30  
 Register  
   names, assembly 3-15  
 Register banks 2-5  
 Register-relative  
   expressions 3-28  
 Register-relative labels 3-21  
 Registers 2-5  
 Relational operators, assembly 3-34  
 REQUIRE directive 6-79  
 REQUIRE8 directive 6-80  
 REV ARM instruction 4-65  
 REV Thumb 16-bit instruction 4-65  
 REV Thumb-2 instruction 4-65  
 REVSH ARM instruction 4-65  
 REVSH Thumb 16-bit instruction 4-65  
 REVSH Thumb-2 instruction 4-65  
 REV16 ARM instruction 4-65  
 REV16 Thumb 16-bit instruction 4-65  
 REV16 Thumb-2 instruction 4-65  
 RFE ARM instruction 4-32  
 RFE Thumb-2 instruction 4-32  
 :RIGHT: operator 3-32  
 RLIST directive 6-9  
 RN directive 6-82  
 ROR Thumb-2 instruction 4-67  
 ROUT directive 2-12, 3-22, 3-23, 6-83  
 RRX Thumb-2 instruction 4-67  
 RSB ARM instruction 4-45  
 RSB Thumb-2 instruction 4-45  
 RSC ARM instruction 4-45

## S

SADDSUBX ARM instruction 4-101  
 SADDSUBX Thumb-2 instruction 4-101  
 SADD16 ARM instruction 4-101  
 SADD16 Thumb-2 instruction 4-101  
 SADD8 ARM instruction 4-101  
 SADD8 Thumb-2 instruction 4-101  
 Saturating instructions 4-95  
 SBC ARM instruction 4-45  
 SBC Thumb 16-bit instruction 4-45  
 SBC Thumb-2 instruction 4-45

- SBFX ARM instruction 4-109
  - SBFX Thumb-2 instruction 4-109
  - Scope, assembly language 2-12
  - SEL ARM instruction 4-63
  - SEL Thumb-2 instruction 4-63
  - SETA directive 3-9, 3-17, 3-19, 6-8, 6-54
  - SETEND ARM instruction 4-140
  - SETEND Thumb 16-bit instruction 4-140
  - SETL directive 3-9, 3-17, 3-19, 6-8, 6-54
  - SETS directive 3-9, 3-17, 3-19, 6-8, 6-54
  - SEV ARM instruction 4-131, 4-141
  - SEV Thumb-2 instruction 4-131, 4-141
  - SEV Thumb-2 16-bit instruction 4-141
  - SHADDSUBX ARM instruction 4-101
  - SHADDSUBX Thumb-2 instruction 4-101
  - SHADD16 ARM instruction 4-101
  - SHADD16 Thumb-2 instruction 4-101
  - SHADD8 ARM instruction 4-101
  - SHADD8 Thumb-2 instruction 4-101
  - Shift operators, assembly 3-33
  - SHSUBADDX ARM instruction 4-101
  - SHSUBADDX Thumb-2 instruction 4-101
  - SHSUB16 ARM instruction 4-101
  - SHSUB16 Thumb-2 instruction 4-101
  - SHSUB8 ARM instruction 4-101
  - SHSUB8 Thumb-2 instruction 4-101
  - SMI ARM instruction 4-139
  - SMI Thumb-2 instruction 4-139
  - SMLAD ARM instruction 4-88
  - SMLAD Thumb-2 instruction 4-88
  - SMLAL ARM instruction 4-76
  - SMLAL Thumb-2 instruction 4-76
  - SMLALBB ARM instruction 4-82
  - SMLALBB Thumb-2 instruction 4-82
  - SMLALBT ARM instruction 4-82
  - SMLALBT Thumb-2 instruction 4-82
  - SMLALD ARM instruction 4-90
  - SMLALD Thumb-2 instruction 4-90
  - SMLALTB ARM instruction 4-82
  - SMLALTB Thumb-2 instruction 4-82
  - SMLALTT ARM instruction 4-82
  - SMLALTT Thumb-2 instruction 4-82
  - SMLSD ARM instruction 4-88
  - SMLSD Thumb-2 instruction 4-88
  - SMLSLD ARM instruction 4-90
  - SMLSLD Thumb-2 instruction 4-90
  - SMMUL ARM instruction 4-86
  - SMMUL Thumb-2 instruction 4-86
  - SMUAD ARM instruction 4-84
  - SMUAD Thumb-2 instruction 4-84
  - SMULBB ARM instruction 4-78
  - SMULBB Thumb-2 instruction 4-78
  - SMULBT ARM instruction 4-78
  - SMULBT Thumb-2 instruction 4-78
  - SMULL ARM instruction 4-76
  - SMULL Thumb-2 instruction 4-76
  - SMULTB ARM instruction 4-78
  - SMULTB Thumb-2 instruction 4-78
  - SMULTT ARM instruction 4-78
  - SMULTT Thumb-2 instruction 4-78
  - SMULWB ARM instruction 4-80
  - SMULWB Thumb-2 instruction 4-80
  - SMULWT ARM instruction 4-80
  - SMULWT Thumb-2 instruction 4-80
  - SMUSD ARM instruction 4-84
  - SMUSD Thumb-2 instruction 4-84
  - SN directive 6-12
  - SPACE directive 6-18
  - SPSR 4-27, 4-34, 4-35, 4-46, 4-49, 4-51, 4-58, 4-71
  - SRS ARM instruction 4-34
  - SRS Thumb-2 instruction 4-34
  - SSAT ARM instruction 4-98
  - SSAT Thumb-2 instruction 4-98
  - SSAT16 ARM instruction 4-106
  - SSAT16 Thumb-2 instruction 4-106
  - SSUBADDX ARM instruction 4-101
  - SSUBADDX Thumb-2 instruction 4-101
  - SSUB16 ARM instruction 4-101
  - SSUB16 Thumb-2 instruction 4-101
  - SSUB8 ARM instruction 4-101
  - SSUB8 Thumb-2 instruction 4-101
  - Stack pointer 2-5
  - Stacks, assembly language 2-39
  - State
    - ARM 2-4
    - Thumb 2-4
  - Status flags 2-17, 2-18
  - STC ARM instruction 4-127
  - STC Thumb-2 instruction 4-127
  - STC2 ARM instruction 4-129
  - STC2 Thumb-2 instruction 4-129
  - STM ARM instruction 4-27
  - STM Thumb-2 instruction 4-27
  - STR ARM instruction 4-9, 4-13, 4-16, 4-19
  - :STR: operator 3-30
  - STREX ARM instruction 4-36
  - STREX Thumb-2 instruction 4-36
  - String
    - expressions, assembly 3-24
    - manipulation, assembly 3-32
    - variable, assembly 3-19
  - String constants, assembly language 2-13
  - String copying, ARM state example 2-36
  - String literals, assembly 3-25
  - SUB ARM instruction 4-45
  - SUB Thumb 16-bit instruction 4-45
  - SUB Thumb-2 instruction 4-45
  - Subroutines, assembly language 2-16
  - SUBS PC, LR Thumb-2 instruction 4-49
  - SUBT directive 6-56
  - SWI ARM instruction 4-133
  - SWI Thumb 16-bit instruction 4-133
  - SWI Thumb-2 instruction 4-133
  - SWP ARM instruction 4-40
  - SWPB ARM instruction 4-40
  - SXT ARM instruction 4-110
  - SXT Thumb 16-bit instruction 4-110
  - SXT Thumb-2 instruction 4-110
  - SXTA ARM instruction 4-110
  - SXTA Thumb-2 instruction 4-110
  - Symbol versioning
    - creating versioned symbols 2-46
    - default version 2-46
    - version 2-46
  - Symbols
    - assembly language 3-18
    - assembly language, naming rules 3-18
- ## T
- TBB ARM instruction 4-120
  - TBB Thumb-2 instruction 4-120
  - TBH Thumb-2 instruction 4-120
  - TEQ ARM instruction 4-61
  - TEQ Thumb-2 instruction 4-61
  - Thumb
    - popping PC 2-40
    - state 2-4
  - THUMB directive 6-59
  - Thumb instructions
    - summary 4-2

- Q flag 2-22
- Thumb 16-bit instructions
  - ADC 4-45
  - ADD 4-45
  - AND 4-50
  - B 4-116
  - BIC 4-50
  - BKPT 4-132
  - BL 4-116
  - CMN 4-55
  - CMP 4-55
  - CPS 4-137
  - EOR 4-50
  - MOV 4-57
  - MUL 4-74
  - MVN 4-57
  - ORR 4-50
  - REV 4-65
  - REVSH 4-65
  - REV16 4-65
  - SBC 4-45
  - SETEND 4-140
  - SUB 4-45
  - SWI 4-133
  - SXT 4-110
  - TST 4-61
  - UXT 4-110
- Thumb-2 instructions
  - summary 4-2
  - second operand 4-42
  - ADC 4-45
  - ADD 4-45
  - AND 4-50
  - ASR 4-67
  - B (conditional) 4-116
  - B (unconditional) 4-116
  - BFC 4-53
  - BFI 4-53
  - BIC 4-50
  - BL 4-116
  - CDP 4-122
  - CDP2 4-122
  - CLREX 4-39
  - CLZ 4-54
  - CMN 4-55
  - CMP 4-55
  - CPS 4-137
  - EOR 4-50
  - LDC 4-127
  - LDC2 4-129
  - LDM 4-27
  - LDREX 4-36
  - LSL 4-67
  - LSR 4-67
  - MCR 4-123
  - MCRR 4-123
  - MCRR2 4-123
  - MCR2 4-123
  - MLA 4-74
  - MLS 4-74
  - MOV 4-57
  - MOVT 4-60
  - MRC 4-125
  - MRC2 4-125
  - MRS 4-134
  - MSR 4-135
  - MUL 4-74
  - MVN 4-57
  - ORN 4-50
  - ORR 4-50
  - PKHBT 4-113
  - PKHTB 4-113
  - POP 4-30
  - PUSH 4-30
  - QADD 4-96
  - QADDSUBX 4-101
  - QADD16 4-101
  - QADD8 4-101
  - QDADD 4-96
  - QDSUB 4-96
  - QSUB 4-96
  - QSUBADDX 4-101
  - QSUB16 4-101
  - QSUB8 4-101
  - RBIT 4-65
  - REV 4-65
  - REVSH 4-65
  - REV16 4-65
  - RFE 4-32
  - ROR 4-67
  - RRX 4-67
  - RSB 4-45
  - SADDSUBX 4-101
  - SADD16 4-101
  - SADD8 4-101
  - SBC 4-45
  - SBFX 4-109
  - SEL 4-63
  - SEV 4-131, 4-141
  - SHADDSUBX 4-101
  - SHADD16 4-101
  - SHADD8 4-101
  - SHSUBADDX 4-101
  - SHSUB16 4-101
  - SHSUB8 4-101
  - SMI 4-139
  - SMLAD 4-88
  - SMLAL 4-76
  - SMLALBB 4-82
  - SMLALBT 4-82
  - SMLALD 4-90
  - SMLALTB 4-82
  - SMLALTT 4-82
  - SMLSD 4-88
  - SMLSLD 4-90
  - SMMUL 4-86
  - SMUAD 4-84
  - SMULBB 4-78
  - SMULBT 4-78
  - SMULL 4-76
  - SMULTB 4-78
  - SMULTT 4-78
  - SMULWB 4-80
  - SMULWT 4-80
  - SMUSD 4-84
  - SRS 4-34
  - SSAT 4-98
  - SSAT16 4-106
  - SSUBADDX 4-101
  - SSUB16 4-101
  - SSUB8 4-101
  - STC 4-127
  - STC2 4-129
  - STM 4-27
  - STREX 4-36
  - SUB 4-45
  - SUBS PC, LR 4-49
  - SWI 4-133
  - SXT 4-110
  - SXTA 4-110
  - TBB 4-120
  - TBH 4-120
  - TEQ 4-61
  - TST 4-61
  - UADDSUBX 4-101
  - UADD16 4-101
  - UADD8 4-101

UBFX 4-109  
 UHADDSUBX 4-101  
 UHADD16 4-101  
 UHADD8 4-101  
 UHSUBADDX 4-101  
 UHSUB16 4-101  
 UHSUB8 4-101  
 UMAAL 4-92  
 UMLAL 4-76  
 UMULL 4-76  
 UQADDSUBX 4-101  
 UQADD16 4-101  
 UQADD8 4-101  
 UQSUBADDX 4-101  
 UQSUB16 4-101  
 UQSUB8 4-101  
 USADA8 4-104  
 USAD8 4-104  
 USAT 4-98  
 USAT16 4-106  
 USUBADDX 4-101  
 USUB16 4-101  
 USUB8 4-101  
 UXT 4-110  
 UXTA 4-110  
 Thumb-2 16-bit instructions  
   CBNZ 4-119  
   CBZ 4-119  
   IT 4-70  
   SEV 4-141  
 TrustZone  
   about 2-3  
 TST ARM instruction 4-61  
 TST Thumb 16-bit instruction 4-61  
 TST Thumb-2 instruction 4-61  
 TTL directive 6-56

## U

UADDSUBX ARM instruction 4-101  
 UADDSUBX Thumb-2 instruction 4-101  
 UADD16 ARM instruction 4-101  
 UADD16 Thumb-2 instruction 4-101  
 UADD8 ARM instruction 4-101  
 UADD8 Thumb-2 instruction 4-101  
 UBFX ARM instruction 4-109  
 UBFX Thumb-2 instruction 4-109  
 UHADDSUBX ARM instruction 4-101  
 UHADDSUBX Thumb-2 instruction 4-101  
 UHADD16 ARM instruction 4-101  
 UHADD16 Thumb-2 instruction 4-101  
 UHADD8 ARM instruction 4-101  
 UHADD8 Thumb-2 instruction 4-101  
 UHSUBADDX ARM instruction 4-101  
 UHSUBADDX Thumb-2 instruction 4-101  
 UHSUB16 ARM instruction 4-101  
 UHSUB16 Thumb-2 instruction 4-101  
 UHSUB8 ARM instruction 4-101  
 UHSUB8 Thumb-2 instruction 4-101  
 UMAAL ARM instruction 4-92  
 UMAAL Thumb-2 instruction 4-92  
 UMLAL ARM instruction 4-76  
 UMLAL Thumb-2 instruction 4-76  
 UMULL ARM instruction 4-76  
 UMULL Thumb-2 instruction 4-76  
 Unary operators, assembly 3-30  
 UQADDSUBX ARM instruction 4-101  
 UQADDSUBX Thumb-2 instruction 4-101  
 UQADD16 ARM instruction 4-101  
 UQADD16 Thumb-2 instruction 4-101  
 UQADD8 ARM instruction 4-101  
 UQADD8 Thumb-2 instruction 4-101  
 UQSUBADDX ARM instruction 4-101  
 UQSUBADDX Thumb-2 instruction 4-101  
 UQSUB16 ARM instruction 4-101  
 UQSUB16 Thumb-2 instruction 4-101  
 UQSUB8 ARM instruction 4-101  
 UQSUB8 Thumb-2 instruction 4-101  
 USADA8 ARM instruction 4-104  
 USADA8 Thumb-2 instruction 4-104  
 USAD8 ARM instruction 4-104  
 USAD8 Thumb-2 instruction 4-104  
 USAT ARM instruction 4-98  
 USAT Thumb-2 instruction 4-98  
 USAT16 ARM instruction 4-106  
 USAT16 Thumb-2 instruction 4-106  
 USUBADDX ARM instruction 4-101  
 USUBADDX Thumb-2 instruction 4-101  
 USUB16 ARM instruction 4-101  
 USUB16 Thumb-2 instruction 4-101  
 USUB8 ARM instruction 4-101  
 USUB8 Thumb-2 instruction 4-101  
 UXT ARM instruction 4-110  
 UXT Thumb 16-bit instruction 4-110  
 UXT Thumb-2 instruction 4-110  
 UXTA ARM instruction 4-110  
 UXTA Thumb-2 instruction 4-110

## V

Variables, assembly 3-19  
   built-in 3-16  
   global 6-5, 6-8  
   local 6-7, 6-8  
   substitution 3-19  
 Version  
   symbol versioning 2-46  
 VFP directives and notation 5-37  
 VFP instructions  
   summary 5-15  
   FABS 5-16  
   FADD 5-17  
   FCMP 5-18  
   FCPY 5-16  
   FCVTDS 5-19  
   FCVTSD 5-20  
   FDIV 5-21  
   FLD 5-22  
   FLDM 5-24  
   FMAC 5-26  
   FMDHR 5-28  
   FMDLR 5-28  
   FMDRR 5-27  
   FMRDH 5-28  
   FMRDL 5-28  
   FMRRD 5-27  
   FMRRS 5-30  
   FMRS 5-29  
   FMRX 5-31  
   FMSC 5-26  
   FMSR 5-29  
   FMSRR 5-30  
   FMSTAT 5-31  
   FMUL 5-32  
   FMXR 5-31  
   FNEG 5-16  
   FNMAC 5-26  
   FNMSC 5-26  
   FNMUL 5-32  
   FSITO 5-33  
   FSQRT 5-34  
   FST 5-22  
   FSTM 5-24  
   FSUB 5-17  
   FTOSI 5-35  
   FTOUI 5-35  
   FUITO 5-33

VFP pseudo-instructions  
    FLD 5-36  
VFPPASSERT SCALAR directive 5-38  
VFPPASSERT VECTOR directive 5-39  
Visual C++  
    diagnostic message style 3-11

## W

WEAK symbol 6-72, 6-75  
WEND directive 6-36  
WHILE directive 6-32, 6-36

## Symbols

! directive 6-53  
# directive 6-17  
% directive 6-18  
& directive 6-20  
\* directive 6-68  
= directive 6-19  
[ directive 6-33  
] directive 6-33  
^ directive 6-16  
| directive 6-33

