RealView Compilation Tools
Assembler Guide

Copyright © 2002-2010 ARM. All rights reserved.

Release Information

The following changes have been made to this book.

<table>
<thead>
<tr>
<th>Date</th>
<th>Issue</th>
<th>Confidential</th>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>August 2002</td>
<td>A</td>
<td>Non-Confidential</td>
<td>Release 1.2</td>
</tr>
<tr>
<td>January 2003</td>
<td>B</td>
<td>Non-Confidential</td>
<td>Release 2.0</td>
</tr>
<tr>
<td>September 2003</td>
<td>C</td>
<td>Non-Confidential</td>
<td>Release 2.0.1 for RealView Development Suite v2.0</td>
</tr>
<tr>
<td>January 2004</td>
<td>D</td>
<td>Non-Confidential</td>
<td>Release 2.1 for RealView Development Suite v2.1</td>
</tr>
<tr>
<td>December 2004</td>
<td>E</td>
<td>Non-Confidential</td>
<td>Release 2.2 for RealView Development Suite v2.2</td>
</tr>
<tr>
<td>May 2005</td>
<td>F</td>
<td>Non-Confidential</td>
<td>Release 2.2 for RealView Development Suite v2.2 SP1</td>
</tr>
<tr>
<td>March 2006</td>
<td>G</td>
<td>Non-Confidential</td>
<td>Release 3.0 for RealView Development Suite v3.0</td>
</tr>
<tr>
<td>March 2007</td>
<td>H</td>
<td>Non-Confidential</td>
<td>Release 3.1 for RealView Development Suite v3.1</td>
</tr>
<tr>
<td>September 2008</td>
<td>I</td>
<td>Non-Confidential</td>
<td>Release 4.0 for RealView Development Suite v4.0</td>
</tr>
<tr>
<td>23 January 2009</td>
<td>I</td>
<td>Non-Confidential</td>
<td>Update 1 for RealView Development Suite v4.0</td>
</tr>
<tr>
<td>10 December 2010</td>
<td>J</td>
<td>Non-Confidential</td>
<td>Update 2 for RealView Development Suite v4.0</td>
</tr>
</tbody>
</table>

Proprietary Notice

Words and logos marked with ® or ™ are registered trademarks or trademarks of ARM® in the EU and other countries, except as otherwise stated below in this proprietary notice. 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 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.

Where the term ARM is used it means “ARM or any of its subsidiaries as appropriate”.

Copyright © 2002-2010 ARM. All rights reserved. ARM DUI 0204J Non-Confidential, ID101213
Confidentiality Status

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

Unrestricted Access is an ARM internal classification.

Product Status

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

Web Address

http://www.arm.com
Contents

RealView Compilation Tools Assembler Guide

Preface
About this book ........................................................................................................... x
Feedback ................................................................................................................. xiv

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 ....................................................... 2-12
2.4 Conditional execution ................................................................................... 2-18
2.5 Loading constants into registers ................................................................. 2-25
2.6 Loading addresses into registers ................................................................. 2-33
2.7 Load and store multiple register instructions ............................................. 2-39
2.8 Using macros .............................................................................................. 2-46
2.9 Adding symbol versions ............................................................................ 2-50
2.10 Using frame directives ............................................................................ 2-51
2.11 Assembly language changes ................................................................ 2-52

Chapter 3  Assembler Reference
3.1 Command syntax .......................................................................................... 3-2
3.2 Format of source lines ................................................................. 3-22
3.3 Predefined register and coprocessor names .................................. 3-23
3.4 Built-in variables and constants .................................................. 3-25
3.5 Symbols .................................................................................... 3-27
3.6 Expressions, literals, and operators ............................................ 3-33
3.7 Diagnostic messages ................................................................. 3-45
3.8 Using the C preprocessor .......................................................... 3-47

Chapter 4 ARM and Thumb Instructions
4.1 Instruction summary ................................................................. 4-2
4.2 Memory access instructions ....................................................... 4-10
4.3 General data processing instructions ......................................... 4-41
4.4 Multiply instructions ............................................................... 4-71
4.5 Saturating instructions .............................................................. 4-93
4.6 Parallel instructions ................................................................. 4-98
4.7 Packing and unpacking instructions ......................................... 4-106
4.8 Branch and control instructions ................................................. 4-114
4.9 Coprocessor instructions ........................................................ 4-123
4.10 Miscellaneous instructions ...................................................... 4-131
4.11 Instruction width selection in Thumb ....................................... 4-148
4.12 ThumbEE instructions ............................................................. 4-150
4.13 Pseudo-instructions ............................................................... 4-154

Chapter 5 NEON and VFP Programming
5.1 Instruction summary ................................................................. 5-2
5.2 Architecture support for NEON and VFP ................................. 5-8
5.3 The extension register bank ...................................................... 5-9
5.4 Condition codes ........................................................................ 5-12
5.5 General information ............................................................... 5-14
5.6 Instructions shared by NEON and VFP ..................................... 5-20
5.7 NEON logical and compare operations .................................... 5-28
5.8 NEON general data processing instructions ............................ 5-36
5.9 NEON shift instructions .......................................................... 5-48
5.10 NEON general arithmetic instructions .................................... 5-54
5.11 NEON multiply instructions .................................................... 5-67
5.12 NEON load / store element and structure instructions ............ 5-72
5.13 NEON and VFP pseudo-instructions ..................................... 5-80
5.14 NEON and VFP system registers ........................................... 5-87
5.15 Flush-to-zero mode ............................................................... 5-92
5.16 VFP instructions ..................................................................... 5-94
5.17 VFP vector mode ................................................................. 5-104

Chapter 6 Wireless MMX Technology Instructions
6.1 Introduction ............................................................................... 6-2
6.2 ARM support for Wireless MMX Technology ......................... 6-3
6.3 Wireless MMX instructions ...................................................... 6-7
# Chapter 7  Directives Reference

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1</td>
<td>Alphabetical list of directives</td>
<td>7-2</td>
</tr>
<tr>
<td>7.2</td>
<td>Symbol definition directives</td>
<td>7-3</td>
</tr>
<tr>
<td>7.3</td>
<td>Data definition directives</td>
<td>7-15</td>
</tr>
<tr>
<td>7.4</td>
<td>Assembly control directives</td>
<td>7-31</td>
</tr>
<tr>
<td>7.5</td>
<td>Frame directives</td>
<td>7-41</td>
</tr>
<tr>
<td>7.6</td>
<td>Reporting directives</td>
<td>7-56</td>
</tr>
<tr>
<td>7.7</td>
<td>Instruction set and syntax selection directives</td>
<td>7-62</td>
</tr>
<tr>
<td>7.8</td>
<td>Miscellaneous directives</td>
<td>7-65</td>
</tr>
</tbody>
</table>
Preface

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

- About this book on page x
- Feedback on page xiv.
About this book

This book provides tutorial and reference information on the RealView® Compilation Tools assemblers (ARM 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.

Intended audience

This book is written for all developers who are producing applications using RealView Compilation Tools. 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 Essentials Guide.

Using this book

This book is organized into the following chapters:

Chapter 1 Introduction
Read this chapter for an introduction to the ARM 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 pre-Thumb-2 Thumb, and Thumb-2EE.

Chapter 5 NEON and VFP Programming
Read this chapter for reference material on the ARM NEON™ Technology and the VFP instruction set. This also describes other VFP-specific assembly language information.
Chapter 6 Wireless MMX Technology Instructions

Read this chapter for reference material on ARM support for Wireless MMX™ Technology.

Chapter 7 Directives Reference

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

This book assumes that the ARM software is installed 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:

- **monospace** Denotes text that can be entered at the keyboard, such as commands, file and program names, and source code.

- **monos**pace Denotes a permitted abbreviation for a command or option. The underlined text can be entered instead of the full command or option name.

- **monospace italic** Denotes arguments to commands and functions where the argument is to be replaced by a specific value.

- **monospace bold** Denotes language keywords when used outside example code.

- **italic** 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 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://infocenter.arm.com for current errata sheets and addenda, and the ARM Frequently Asked Questions (FAQs)

ARM publications

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

- RVCT Essentials Guide (ARM DUI 0202)
- RVCT Compiler User Guide (ARM DUI 0205)
- RVCT Compiler Reference Guide (ARM DUI 0348)
- RVCT Libraries and Floating Point Support Guide (ARM DUI 0349)
- RVCT Linker User Guide (ARM DUI 0206)
- RVCT Linker Reference Guide (ARM DUI 0381)
- RVCT Utilities Guide (ARM DUI 0382)
- RVCT Developer Guide (ARM DUI 0203).

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:

- ARMv7-M Architecture Reference Manual (ARM DDI 0403)
- ARMv6-M Architecture Reference Manual (ARM DDI 0419)
- ARM datasheet or technical reference manual for your hardware device.
Other publications


Feedback

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

Feedback on RealView Compilation Tools

If you have any problems with RealView Compilation Tools, 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. It contains the following section:

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

RealView Compilation Tools (RVCT) provide:

- A freestanding assembler, armasm, documented in this guide.
- An optimizing inline assembler and a non-optimizing embedded assembler built into the C and C++ compilers. These use the same syntax for assembly instructions, but are otherwise not documented in this guide. See the Mixing C, C++, and Assembly Language chapter in the Developer Guide for more information on the inline and embedded assemblers.

If you are upgrading to RVCT from a previous release, read the Essentials Guide for details about new features and enhancements in this release.

1.1.1 ARM assembly language

The current ARM and Thumb assembler language has superseded earlier versions of both the ARM and Thumb assembler languages. It is sometimes referred to as Unified Assembler Language (UAL).

Code written using UAL can be assembled for ARM, Thumb-2, or pre-Thumb-2 Thumb. The assembler faults the use of unavailable instructions.

1.1.2 Wireless MMX Technology instructions

The assembler supports Intel® Wireless MMX™ Technology instructions to assemble code to run on the PXA270 processor. This processor implements ARMv5TE architecture, with MMX extensions. RVCT supports Wireless MMX Technology Control and Single Instruction Multiple Data (SIMD) Data registers, and include new directives for Wireless MMX Technology development. There is also enhanced support for load and store instructions. See Chapter 6 Wireless MMX Technology Instructions for information about the Wireless MMX Technology support.

1.1.3 NEON technology

ARM NEON™ Technology is the implementation of the Advanced SIMD architecture extension. It is a 64/128 bit hybrid SIMD technology targeted at advanced media and signal processing applications and embedded processors. It is implemented as part of the ARM core, but has its own execution pipelines and a register bank that is distinct from the ARM core register bank.

NEON instructions are available in both ARM and Thumb-2 code. See Chapter 5 NEON and VFP Programming for details of NEON.
1.1.4 Using the examples

This book references examples provided with RealView Development Suite in the main examples directory `install_directory\RVDS\Examples`. See RealView Development Suite Getting Started Guide for a summary of the examples provided.
Introduction
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 2-12
- Conditional execution on page 2-18
- Loading constants into registers on page 2-25
- Loading addresses into registers on page 2-33
- Load and store multiple register instructions on page 2-39
- Using macros on page 2-46
- Adding symbol versions on page 2-50
- Using frame directives on page 2-51
- Assembly language changes on page 2-52.
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®, Thumb-2, NEON™, VFP, or Wireless MMX instruction sets. For this information see:

- Chapter 4 ARM and Thumb Instructions
- Chapter 5 NEON and VFP Programming
- Chapter 6 Wireless MMX Technology Instructions.

For more information, see ARM Architecture Reference Manual.

For the convenience of programmers who are familiar with the ARM and Thumb assembly languages accepted in RVCT v2.1 and earlier, this chapter includes a section outlining the differences between them and the latest version of the ARM assembly language. See Assembly language changes on page 2-52.

2.1.1 Code examples

There are a number of code examples in this chapter. Many of them are supplied in the install_directory\RVDS\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, with an appropriate debug target such as the RealView Instruction Set Simulator (RealView ISS).

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

fromelf -c filename.o


See the Application Binary Interface (ABI) documentation on
http://infocenter.arm.com for details on ELF and DWARF.
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 and 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, Thumb-2, and Thumb-2EE instruction sets
- ARM, Thumb, and ThumbEE state on page 2-4
- Processor mode on page 2-5
- Registers on page 2-6
- Instruction set overview on page 2-8
- Instruction capabilities on page 2-10.

2.2.1 Architecture versions

The information and examples in this book assume that you are using a processor that implements ARMv4 or above. 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, Thumb-2, and Thumb-2EE 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.

ARMv7 defines the Thumb-2 Execution Environment (Thumb-2EE). The Thumb-2EE instruction set is based on Thumb-2, with some changes and additions to make it a better target for dynamically generated code, that is, code compiled on the device either shortly before or during execution.

For more information, see Instruction set overview on page 2-8.
2.2.3 ARM, Thumb, and ThumbEE state

A processor that is executing ARM instructions is operating in ARM state. A processor that is executing Thumb instructions is operating in Thumb state. A processor that is executing ThumbEE instructions is operating in ThumbEE state. A processor can also operate in another state called the Jazelle state.

A processor in one state cannot execute instructions from another instruction set. For example, a processor in ARM state cannot execute Thumb instructions, and a processor in Thumb state cannot execute ARM instructions. You must ensure that the processor never receives instructions of the wrong instruction set for the current state.

Most ARM processors always start executing code in ARM state. However, some processors can only execute Thumb code, or can be configured to start in Thumb state.

Changing state

Each instruction set includes instructions to change processor state.

To change between ARM and Thumb states, you must switch the assembler mode to produce the correct opcodes using ARM or THUMB directives. To generate Thumb-2EE code, use THUMBX. (Assembler code using CODE32 and CODE16 can still be assembled by the assembler, but you are recommended to use ARM and THUMB for new code.)

See Instruction set and syntax selection directives on page 7-62 for details.
2.2.4 Processor mode

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

--- Note ---

ARMv6-M and ARMv7-M do not support the same modes as other ARM processors. This section does not apply to ARMv6-M and ARMv7-M.

<table>
<thead>
<tr>
<th>Processor mode</th>
<th>Architectures</th>
<th>Mode number</th>
</tr>
</thead>
<tbody>
<tr>
<td>User</td>
<td>All</td>
<td>0b10000</td>
</tr>
<tr>
<td>FIQ - Fast Interrupt Request</td>
<td>All</td>
<td>0b10001</td>
</tr>
<tr>
<td>IRQ - Interrupt Request</td>
<td>All</td>
<td>0b10010</td>
</tr>
<tr>
<td>Supervisor</td>
<td>All</td>
<td>0b10011</td>
</tr>
<tr>
<td>Abort</td>
<td>All</td>
<td>0b10111</td>
</tr>
<tr>
<td>Undefined</td>
<td>All</td>
<td>0b11011</td>
</tr>
<tr>
<td>System</td>
<td>ARMv4 and above</td>
<td>0b11111</td>
</tr>
<tr>
<td>Monitor</td>
<td>Security Extensions only</td>
<td>0b10110</td>
</tr>
</tbody>
</table>

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

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 Chapter 6 Handling Processor Exceptions in the Developer Guide).

On architectures that implement the Security Extensions, code can run in either a secure state or in a non-secure state. See the ARM Architecture Reference Manual for details.
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)
- The Application Program Status Register (APSR) on page 2-7
- Saved Program Status Registers (SPSRs) on page 2-7.

Thirty general-purpose, 32-bit registers

Fifteen general-purpose registers are visible at any one time, depending on the current processor mode. These are r0-r12, sp, lr.

sp (or r13) is the stack pointer. The C and C++ compilers always use sp as the stack pointer. In Thumb-2, sp is strictly defined as the stack pointer, so many instructions that are not useful for stack manipulation are unpredictable if they use sp. Use of sp as a general purpose register is discouraged.

In User mode, lr (or r14) is used as a link register to store the return address when a subroutine call is made. It can also be used as a general-purpose register if the return address is stored on the stack.

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

The Program Counter (PC)

The Program Counter is accessed as pc (or r15). It is incremented by one word (four bytes) for each instruction in ARM state, or by the size of the instruction executed 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, 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 Application Program Status Register (APSR)

The APSR holds copies of the Arithmetic Logic Unit (ALU) status flags. They are used to determine whether conditional instructions are executed or not. See Conditional execution on page 2-18 for more information.

On ARMv5TE, and ARMv6 and above, the APSR also holds the Q flag (see The ALU status flags on page 2-19).

On ARMv6 and above, the APSR also holds the GE flags (see Parallel add and subtract on page 4-99).

These flags are accessible in all modes, using MSR and MRS instructions. See MRS on page 4-134 and MSR on page 4-136 for details.

The Current Program Status Register (CPSR)

The CPSR holds:
- the APSR flags
- the current processor mode
- interrupt disable flags
- current processor state (ARM, Thumb, ThumbEE, or Jazelle)
- execution state bits for the IT block.

The execution state bits control conditional execution in the IT block (see IT on page 4-118) and are only available on ARMv6T2 and above.

Only the APSR flags are accessible in all modes. The remaining bits of the CPSR are accessible only in privileged modes, using MSR and MRS instructions. See MRS on page 4-134 and MSR on page 4-136 for details.

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 Chapter 6 Handling Processor Exceptions in the 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, Thumb-2, and Thumb-2EE instructions are either 16 or 32 bits long. Instructions are stored half-word aligned. Some instructions use the least significant bit of the address to determine whether the code being branched to is Thumb code or ARM code.

Before the introduction of 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.

In ARMv6 and above, all ARM and Thumb instructions are little-endian.

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 and control instructions
- Data processing instructions on page 2-9
- Register load and store instructions on page 2-9
- Multiple register load and store instructions on page 2-9
- Status register access instructions on page 2-9
- Coprocessor instructions on page 2-9.

Branch and control instructions

These instructions are used to:

- branch to subroutines
- branch backwards to form loops
- branch forward in conditional structures
- make following instructions conditional without branching
- 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.

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.

A few instructions are also defined that can load or store 64-bit doubleword values into two 32-bit registers.

Multiple register load and store 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 2-39 for a detailed description of these instructions.

Status register access instructions

These instructions move the contents of a status register 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 2-11.

Conditional execution

Almost all ARM instructions can be executed conditionally on the value of the ALU status flags in the APSR. 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.

In Thumb state on pre-Thumb-2 processors, 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.

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 non flag-setting instructions in between.

See Conditional execution on page 2-18 for a detailed description.

Register access

In ARM state, all instructions can access r0 to r14, and most can also access pc (r15). The MRS and MSR instructions can move the contents of a status register 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 4-136 for more information.

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

Most Thumb instructions on pre-Thumb-2 processors 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 many 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 2-25 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 pre-Thumb2 Thumb instruction set 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 pre-UAL Thumb assembly language. In this case, you must inform armasm using the --16 command-line option, or the CODE16 directive in the source code. The pre-UAL Thumb assembly language does not support Thumb-2 instructions.

This section describes:
- Layout of assembly language source files
- An example ARM assembly language module on page 2-15
- Calling subroutines on page 2-17.

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.

Some directives do not allow the use of a 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 followed by end-of-line sequence as white space.

--- Note ---

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

---

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

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 3-31 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
- floating-point, for example, 0.02, 123.0, or 3.14159.

Floating-point numbers are only available if your system has VFP, or NEON with floating-point.

**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\RVDS\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
  SVC #0x123456      ; ARM semihosting (formerly 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 Chapter 5 Using Scatter-loading Description Files in the Linker User 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 7-70 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 2-15 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 2-15 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 SVC (0x123456 by default), with the following parameters:

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

See Chapter 8 *Semihosting* in the *RVCT Developer Guide*.

**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 expression. See `B, BL, BX, BLX, and BXJ` on page 4-115 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 `BX lr` instruction to return. By convention, registers r0 to r3 are used to pass parameters to subroutines, and r0 is used to pass a result 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 Procedure Call Standard for the ARM Architecture specification, `aapcs.pdf`, in `install_directory\Documentation\Specifications\...` 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 `subrout.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    subrout, CODE, READONLY     ; Name this block of code
ENTRY                     ; Mark first instruction to execute
start   MOV     r0, #10           ; Set up parameters
        MOV     r1, #3
        BL      doadd             ; Call subroutine
stop    MOV     r0, #0x18         ; angel_SWIreason_ReportException
        LDR     r1, =0x20026      ; ADP_Stopped_ApplicationExit
        SVC     #0x123456         ; ARM semihosting (formerly SWI)
doadd   ADD     r0, r0, r1        ; Subroutine code
        BX      lr                ; Return from subroutine
END                       ; Mark end of file
```
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 Application Program Status Register (APSR) according to the result of the operation. Some instructions update all flags, and some instructions only update a subset. If a flag is not updated, the original value is preserved. The description of each instruction details the effect it has on the flags. Conditional instructions that are not executed have no effect on the flags.

In Thumb state on pre-Thumb-2 processors, most data processing instructions update the ALU status flags automatically. There is no option to leave the flags unchanged and not update them. Other instructions cannot update the flags.

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

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

Almost every ARM instruction can be executed conditionally on the state of the ALU status flags in the APSR. See Table 2-2 on page 2-19 for a list of the suffixes to add to instructions to make them conditional.

In Thumb state, a mechanism for conditional execution is available using a conditional branch.

In Thumb state on processors with Thumb-2, you can also make instructions conditional using a special IT (If-Then) instruction. You can also use the CBZ (Conditional Branch on Zero) and CBNZ instructions to compare the value of a register against zero and branch on the result.

This section describes:
- The ALU status flags on page 2-19
- Conditional execution on page 2-19
- Using conditional execution on page 2-20
- Example of the use of conditional execution on page 2-21
- The Q flag on page 2-24.
2.4.1 The ALU status flags

The APSR 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 zero
- 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

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

In Thumb state on pre-Thumb-2 processors, the `{cond}` field is only permitted on certain branch instructions.

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

<table>
<thead>
<tr>
<th>Suffix</th>
<th>Flags</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>EQ</td>
<td>Z set</td>
<td>Equal</td>
</tr>
<tr>
<td>NE</td>
<td>Z clear</td>
<td>Not equal</td>
</tr>
<tr>
<td>CS or HS</td>
<td>C set</td>
<td>Higher or same (unsigned &gt;= )</td>
</tr>
<tr>
<td>CC or L0</td>
<td>C clear</td>
<td>Lower (unsigned &lt; )</td>
</tr>
<tr>
<td>MI</td>
<td>N set</td>
<td>Negative</td>
</tr>
<tr>
<td>PL</td>
<td>N clear</td>
<td>Positive or zero</td>
</tr>
</tbody>
</table>
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

You can use conditional execution of ARM instructions to reduce the number of branch instructions in your code. This improves code density. The IT instruction in Thumb-2 achieves a similar improvement.

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

Some ARM processors, for example 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:

```c
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:

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

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

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

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

By using the conditional execution feature of the ARM instruction set, you can implement the gcd function in only four instructions.
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 \( r0 = 1 \) and \( r1 = 2 \). 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 \( r0 = r1 \). In all other cases, the conditional version of the code executes in fewer cycles.

**Table 2-3 Conditional branches only**

<table>
<thead>
<tr>
<th>( r0: a )</th>
<th>( r1: b )</th>
<th>Instruction</th>
<th>Cycles (ARM7)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>CMP ( r0, r1 )</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>BEQ end</td>
<td>1 (not executed)</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>BLT less</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>SUB ( r1, r1, r0 )</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>B gcd</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>CMP ( r0, r1 )</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>BEQ end</td>
<td>3</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td></td>
<td><strong>13</strong></td>
</tr>
</tbody>
</table>

**Table 2-4 All instructions conditional**

<table>
<thead>
<tr>
<th>( r0: a )</th>
<th>( r1: b )</th>
<th>Instruction</th>
<th>Cycles (ARM7)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>CMP ( r0, r1 )</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>SUBGT ( r0, r0, r1 )</td>
<td>1 (not executed)</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>SUBLT ( r1, r1, r0 )</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>BNE gcd</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>CMP ( r0, r1 )</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>SUBGT ( r0, r0, r1 )</td>
<td>1 (not executed)</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>SUBLT ( r1, r1, r0 )</td>
<td>1 (not executed)</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>BNE gcd</td>
<td>1 (not executed)</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td></td>
<td><strong>10</strong></td>
</tr>
</tbody>
</table>
Pre-Thumb-2 Thumb version of gcd

Because \(^{8}\) is the only pre-Thumb-2 Thumb instruction that can be executed conditionally, the gcd algorithm must be written with conditional branches.

Like the ARM conditional branch implementation, the pre-Thumb-2 Thumb code requires seven 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 this version runs \textit{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.

Thumb-2 version of gcd

You can convert the ARM version of this code into Thumb-2 code, by making the \texttt{SUB} instructions conditional using an \texttt{IT} instruction:

\begin{verbatim}
  gcd
    CMP     r0, r1
    ITE     GT
    SUBGT   r0, r0, r1
    SUBLE   r1, r1, r0
    BNE     gcd
\end{verbatim}

This assembles equally well to ARM or Thumb-2 code. The assembler checks the \texttt{IT} instructions, but omits them on assembly to ARM code. (You can omit the \texttt{IT} instructions. The assembler inserts them for you when assembling to Thumb-2 code.)

It requires one more instruction in Thumb-2 code than in ARM code, but the overall code size is 10 bytes in Thumb-2 code compared with 16 bytes in ARM code.

Execution speed

To optimize code for execution speed you must have detailed knowledge of the instruction timings, branch prediction logic, and cache behavior of your target system. See \textit{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 4-94), or when overflow has occurred in certain multiply instructions (see SMULxxy and SMLAxy on page 4-76 and SMULWy and SMLAWy on page 4-78).

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 4-136).

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 instruction without performing a data load from memory. This is because ARM and Thumb-2 instructions are only 32 bits long.

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.

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

In ARMv6T2 and above, you can 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 2-26.
- how to use the MOV32 pseudo-instruction to load any 32-bit constant. See Loading with MOV32 on page 2-30.
- how to use the LDR pseudo-instruction to load any 32-bit constant. See Loading with LDR Rd, =const on page 2-30.
- how to load floating-point constants. See Loading floating-point constants on page 2-32.
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 2-28 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 2-26 shows the range of 8-bit values that this provides (for data processing operations).

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

<table>
<thead>
<tr>
<th>Binary</th>
<th>Decimal</th>
<th>Step</th>
<th>Hexadecimal</th>
<th>MVN value</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000000000000000000000abcddefgh</td>
<td>0-255</td>
<td>1</td>
<td>0-0xFF</td>
<td>-1 to -256</td>
<td>-</td>
</tr>
<tr>
<td>0000000000000000000000000000000000abcddefgh00</td>
<td>0-1020</td>
<td>4</td>
<td>0-0x3FC</td>
<td>-4 to -1024</td>
<td>-</td>
</tr>
<tr>
<td>000000000000000000000000000000000000000000000abcddefgh0000</td>
<td>0-4080</td>
<td>16</td>
<td>0-0xFF0</td>
<td>-16 to -4096</td>
<td>-</td>
</tr>
</tbody>
</table>
### Table 2-5 ARM state immediate constants (8-bit) (continued)

<table>
<thead>
<tr>
<th>Binary</th>
<th>Decimal</th>
<th>Step</th>
<th>Hexadecimal</th>
<th>MVN value</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>abcdefgh000000000000000000000000</code></td>
<td>0-16320</td>
<td>64</td>
<td>0-0x3FC0</td>
<td>-64 to -16384</td>
<td>-</td>
</tr>
<tr>
<td><code>... ... ... ... ...</code></td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>-</td>
</tr>
<tr>
<td><code>abcdefgh000000000000000000000000</code></td>
<td>0-255 x $2^{24}$</td>
<td>$2^{24}$</td>
<td>0-0xFF000000</td>
<td>1-256 x $-2^{24}$</td>
<td>-</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Binary</th>
<th>Decimal</th>
<th>Step</th>
<th>Hexadecimal</th>
<th>MVN value</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>abcdefg</code> (bit pattern)</td>
<td>-</td>
<td>-</td>
<td>(bit pattern)</td>
<td>See b in Notes</td>
<td></td>
</tr>
<tr>
<td><code>efgh</code> (bit pattern)</td>
<td>-</td>
<td>-</td>
<td>(bit pattern)</td>
<td>See b in Notes</td>
<td></td>
</tr>
<tr>
<td><code>gh</code> (bit pattern)</td>
<td>-</td>
<td>-</td>
<td>(bit pattern)</td>
<td>See b in Notes</td>
<td></td>
</tr>
</tbody>
</table>

### Table 2-6 ARM state immediate constants in MOV instructions

<table>
<thead>
<tr>
<th>Binary</th>
<th>Decimal</th>
<th>Step</th>
<th>Hexadecimal</th>
<th>MVN value</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>abcdefghijkmnop</code></td>
<td>0-65535</td>
<td>1</td>
<td>0-0xFFF</td>
<td>-</td>
<td>See c in Notes</td>
</tr>
</tbody>
</table>

### Notes

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

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

**b** These values are available in ARM state only. All the other values in this table are also available in Thumb-2.

**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 0-0xFF (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 0-0xFFFF (0-65535). These values are not available as immediate operands in data processing operations.

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

Table 2-8 on page 2-29 shows the range of 16-bit values that this provides (for MOV instructions only).

<table>
<thead>
<tr>
<th>Binary</th>
<th>Decimal</th>
<th>Step</th>
<th>Hexadecimal</th>
<th>MVN value</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>000...000abcdefgh</td>
<td>0-255</td>
<td>1</td>
<td>0-0xFF</td>
<td>-1 to -256</td>
<td>-</td>
</tr>
<tr>
<td>000...000abcdefgh0</td>
<td>0-510</td>
<td>2</td>
<td>0-0x1FE</td>
<td>-2 to -512</td>
<td>-</td>
</tr>
<tr>
<td>000...000abcdefgh00</td>
<td>0-1020</td>
<td>4</td>
<td>0-0x3FC</td>
<td>-4 to -1024</td>
<td>-</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0abcdefgh000...000</td>
<td>0-255 x 2^{23}</td>
<td>2^{23}</td>
<td>0-0x7F800000</td>
<td>1-256 x -2^{23}</td>
<td>-</td>
</tr>
<tr>
<td>abcdefgh000...000</td>
<td>0-255 x 2^{24}</td>
<td>2^{24}</td>
<td>0-0xF8000000</td>
<td>1-256 x -2^{24}</td>
<td>-</td>
</tr>
<tr>
<td>abcdefghabcdefgh</td>
<td>(bit pattern)</td>
<td></td>
<td>0xYYYYYYYY</td>
<td>0xYYYYYYYY</td>
<td>-</td>
</tr>
</tbody>
</table>
**Table 2-7 Thumb-2 immediate constants (continued)**

<table>
<thead>
<tr>
<th>Binary</th>
<th>Decimal</th>
<th>Step</th>
<th>Hexadecimal</th>
<th>MVN value</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000abcdefgh00000000abcdefgh (bit pattern)</td>
<td>-</td>
<td>0x00XY00XY</td>
<td>0xFFXYFFXY</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>abcdefgh00000000abcdefgh00000000 (bit pattern)</td>
<td>-</td>
<td>0xXY00XY00</td>
<td>0xYFFXYFF</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0000000000000000abcdefghijkl</td>
<td>0-4095</td>
<td>1</td>
<td>0-0xFFF</td>
<td>-</td>
<td>See b in Notes on page 2-29</td>
</tr>
</tbody>
</table>

**Table 2-8 Thumb-2 immediate constants in MOV instructions**

<table>
<thead>
<tr>
<th>Binary</th>
<th>Decimal</th>
<th>Step</th>
<th>Hexadecimal</th>
<th>MVN value</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000000000000000abcdefghijklmnop</td>
<td>0-65535</td>
<td>1</td>
<td>0-0xFFF</td>
<td>-</td>
<td>See c in Notes</td>
</tr>
</tbody>
</table>

**Notes**

These notes give extra information on Table 2-7 on page 2-28 and Table 2-8.

- **a**  The MVN values are only available directly as operands in MVN instructions.
- **b**  These values are available directly as operands in ADD, SUB, and MOV instructions, but not in MVN or any other data processing instructions.
- **c**  These values are only available in MOV instructions.
2.5.2 Loading with MOV32

In ARMv6T2 and above, 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 4-157 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 single MOV or MVN instruction, the assembler generates the appropriate instruction.
- If the constant cannot be constructed with a single MOV or MVN instruction, the assembler:
  - places the value in a literal pool (a portion of memory embedded in the code to hold constant values)
  - generates an LDR instruction with a program-relative address that reads the constant from the literal pool.

For example:

```
LDR      r n, [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 2-31 for more information.

See LDR pseudo-instruction on page 4-159 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 or Thumb-2 code, but can be in either direction
- forward and less than 1KB in pre-Thumb-2 Thumb code, or when using the 16-bit instruction in Thumb-2 code.

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 ±4KB (ARM, 32-bit Thumb-2) or in the range 0 to +1KB (pre-Thumb-2 Thumb, 16-bit Thumb-2). See LTORG on page 7-17 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 on page 2-32 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

```assembly
AREA     Loadcon, CODE, READONLY
ENTRY                              ; Mark first instruction to execute
start
BL       func1                      ; Branch to first subroutine
BL       func2                      ; Branch to second subroutine
stop
MOV      r0, #0x18                 ; angel_SWIreason_ReportException
LDR      r1, =0x20026              ; ADP_Stopped_ApplicationExit
SVC  #0x123456                 ; ARM semihosting (formerly SWI)
  func1
LDR      r0, =42                   ; => MOV R0, #42
LDR      r1, =0x55555555           ; => LDR R1, [PC, #offset to
  ; Literal Pool 1]
LDR      r2, =0xFFFFFFFF           ; => MVN R2, #0
BX       lr
LTORG                              ; Literal Pool 1 contains
  LTORG ; literal Ox55555555
func2
LDR      r3, =0x55555555           ; => LDR R3, [PC, #offset to
  ; Literal Pool 1]
; LDR r4, =0x66666666666       ; 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

In the NEON and VFPv3 instruction sets, there are instructions to load a limited range of floating-point constants as immediate constants. See:

- `VMOV, VMVN (immediate)` on page 5-41 for details of the NEON instructions
- `VMOV` on page 5-103 for details of the VFPv3 instruction.

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

See `VLDR pseudo-instruction` on page 5-81 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 2-36.

You can also load an address into a register using the MOV32 pseudo-instruction, see MOV32 pseudo-instruction on page 4-157.

2.6.1 Direct loading with ADR and ADRL

The ADR instruction and the ADRL pseudo-instruction 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.

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

ADRL is not available in Thumb state on pre-Thumb-2 processors.

ADR

The available range depends on the instruction set:

<table>
<thead>
<tr>
<th>Instruction Set</th>
<th>Available Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARM</td>
<td>±255 bytes to a byte or halfword-aligned address.</td>
</tr>
<tr>
<td></td>
<td>±1020 bytes to a word-aligned address.</td>
</tr>
<tr>
<td>32-bit Thumb-2</td>
<td>±4095 bytes to a byte, halfword, or word-aligned address.</td>
</tr>
<tr>
<td>16-bit Thumb</td>
<td>0 to 1020 bytes. label must be word-aligned. You can use the ALIGN directive to ensure this.</td>
</tr>
</tbody>
</table>

See ADR on page 4-23 for details.
ADRL

The assembler converts an ADRL \texttt{rn},\texttt{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:

\begin{itemize}
\item \textbf{ARM}±64KB to a byte or halfword-aligned address.
\item ±256KB to a word-aligned address.
\item \textbf{32-bit Thumb-2}±1MB to a byte, halfword, or word-aligned address.
\item \textbf{16-bit Thumb}ADRL is not available.
\end{itemize}

See \textit{Loading addresses with LDR Rd, =label} on page 2-36 for information on loading addresses that are outside the range of the ADRL pseudo-instruction.

Implementing a jump table with ADR

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

\begin{verbatim}
Example 2-5 Implementing a jump table (ARM)

\begin{verbatim}
AREA    Jump, CODE, READONLY     ; Name this block of code
        ARM                              ; Following code is ARM code
num     EQU     2                        ; Number of entries in jump table
ENTRY                            ; Mark first instruction to execute
        start                                    ; First instruction to call
        MOV     r0, #0                   ; Set up the three parameters
        MOV     r1, #3
        MOV     r2, #2
        BL      arithfunc                ; Call the function
        CMP     r0, #num                 ; Treat function code as unsigned
        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

        stop
        MOV     r0, #0x18                ; angel_SWIreason_ReportException
        LDR     r1, =0x20026             ; ADP_Stopped_ApplicationExit
        SVC  #0x123456                ; ARM semihosting (formerly SWI)
        CMP     r0, #num
        BR     arithfunc
        BXHS    lr
        ADR     r3, JumpTable
        LDR     pc, [r3,r0,LSL#2]
\end{verbatim}
\end{verbatim}

\end{verbatim}
In Example 2-5 on page 2-34, 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:

- \( \text{argument1}=0 \)  \hspace{1cm} \text{Result} = \text{argument2} + \text{argument3}.
- \( \text{argument1}=1 \)  \hspace{1cm} \text{Result} = \text{argument2} – \text{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 2-34 it assigns the value 2 to \( \text{num} \). When \( \text{num} \) is used elsewhere in the code, the value 2 is substituted. Using EQU in this way is similar to using \#define to define a constant in C.

- **DCD**: Declares one or more words of store. In Example 2-5 on page 2-34 each DCD stores the address of a routine that handles a particular clause of the jump table.

- **LDR**: The \text{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 numeric constant into a register (see Loading with LDR Rd, =const on page 2-30). It also accepts program-relative expressions such as labels, and labels with offsets. See LDR pseudo-instruction on page 4-159 for a description of the syntax.

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:

  \[
  \text{LDR } \text{n } [\text{pc, #offset to literal pool}]
  \]

  \[
  ; \text{load register n with one word}
  ; \text{from the address [pc + offset]}
  \]

You must ensure that there is a literal pool within range (see Placing literal pools on page 2-31 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

<table>
<thead>
<tr>
<th>AREA</th>
<th>LDRlabel, CODE,READONLY</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENTRY</td>
<td></td>
</tr>
<tr>
<td>start</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BL</td>
</tr>
<tr>
<td>stop</td>
<td>BL</td>
</tr>
<tr>
<td></td>
<td>MOV</td>
</tr>
<tr>
<td></td>
<td>LDR</td>
</tr>
<tr>
<td></td>
<td>SVC</td>
</tr>
<tr>
<td></td>
<td>LDR</td>
</tr>
<tr>
<td></td>
<td>=&gt; LDR R0,[PC, #offset into</td>
</tr>
</tbody>
</table>
LDR r1, =Darea + 12
   ; => LDR R1, [PC, #offset into
   ;   Literal Pool 1]
LDR r2, =Darea + 6000
   ; => LDR R2, [PC, #offset into
   ;   Literal Pool 1]
BX  lr
   ; Return
LTORG
   ; Literal Pool 1

func2
LDR r3, =Darea + 6000
   ; => LDR r3, [PC, #offset into
   ;   Literal Pool 1]
   ; (sharing with previous literal)
; LDR r4, =Darea + 6004
   ; If uncommented produces an error
   ; 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 7-21 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      strcopy                   ; Call subroutine to do copy
stop
  MOV     r0, #0x18                 ; angel_SWIreason_ReportException
  LDR     r1, =0x20026              ; ADP_Stopped_ApplicationExit
  SVC    #0x123456                 ; ARM semihosting (formerly SWI)
strcopy
  LDRB    r2, [r1], #1               ; Load byte and update address
  STRB    r2, [r0], #1               ; Store byte and update address
  CMP     r2, #0                    ; Check for zero terminator
  BNE     strcopy                   ; Keep going if not
  MOV     pc, lr                     ; Return

AREA    Strings, DATA, READWRITE
srcstr  DCB     "First string - source",0
dststr  DCB     "Second string - destination",0
END
```
2.7 Load and store multiple register instructions

The ARM, Thumb-2, and pre-Thumb-2 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 --diag_warning 1206 assembler command-line option to check that registers in register lists are specified in increasing order.

This section describes:

• Load and store multiple instructions available in ARM and Thumb on page 2-40
• Implementing stacks with LDM and STM on page 2-41
• Block copy with LDM and STM on page 2-44.
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.

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

In **PUSH** and **POP** instructions:

- The stack pointer (sp) is the base register, and is always updated.

- The address is incremented after each transfer in **POP** instructions, and decremented before each transfer in **PUSH** instructions.

- The list of registers loaded or stored can include:
  - in 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, sp. This means that you can use these instructions to implement push and pop operations for any number of registers in a single instruction.

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

Descending or ascending

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

Full or empty

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

To make it easier for the programmer, stack-oriented suffixes can be used instead of the increment or decrement, and before or after suffixes. Table 2-9 shows the stack-oriented suffixes and their equivalent addressing mode suffixes for load and store instructions.

<table>
<thead>
<tr>
<th>Stack-oriented suffix</th>
<th>For store or push instructions</th>
<th>For load or pop instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>FD (Full Descending stack)</td>
<td>DB (Decrement Before)</td>
<td>IA (Increment After)</td>
</tr>
<tr>
<td>FA (Full Ascending stack)</td>
<td>IB (Increment Before)</td>
<td>DA (Decrement After)</td>
</tr>
<tr>
<td>ED (Empty Descending stack)</td>
<td>DA (Decrement After)</td>
<td>IB (Increment Before)</td>
</tr>
<tr>
<td>EA (Empty Ascending stack)</td>
<td>IA (Increment After)</td>
<td>DB (Decrement Before)</td>
</tr>
</tbody>
</table>
Table 2-10 shows the load and store multiple instructions with the stack-oriented suffixes for the various stack types.

<table>
<thead>
<tr>
<th>Stack type</th>
<th>Store</th>
<th>Load</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full descending</td>
<td>STMFD (STMDB, Decrement Before)</td>
<td>LDMFD (LDM, increment after)</td>
</tr>
<tr>
<td>Full ascending</td>
<td>STMFA (STMIB, Increment Before)</td>
<td>LDMFA (LDMIA, Decrement After)</td>
</tr>
<tr>
<td>Empty descending</td>
<td>STMED (STMDA, Decrement After)</td>
<td>LDMED (LDMIB, Increment Before)</td>
</tr>
<tr>
<td>Empty ascending</td>
<td>STMEA (STM, increment after)</td>
<td>LDMEA (LDMDB, Decrement Before)</td>
</tr>
</tbody>
</table>

For example:

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

Note

The Procedure Call Standard for the ARM Architecture (AAPCS), and 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 these cases you must pop the address into a temporary register and use the BX instruction.

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

See Chapter 5 Interworking ARM and Thumb in the 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_directoryRVDSExamples. 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

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

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 2-45 lists the block copy module rewritten to use LDM and STM for copying.
Example 2-9 Block copy using LDM and STM

```
AREA    Block, CODE, READONLY    ; name this block of code
num         EQU     20                       ; set number of words to be copied
ENTRY                            ; mark the first instruction called
start
LDR     r0, =src                 ; r0 = pointer to source block
LDR     r1, =dst                 ; r1 = pointer to destination block
MOV     r2, #num                 ; r2 = number of words to copy
MOV     sp, #0x400               ; Set up stack pointer (sp)
blockcopy
MOVS    r3,r2, LSR #3            ; Number of eight word multiples
BEQ     copywords                ; 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
        ; originals
copywords
ANDS    r2, r2, #7               ; Number of odd words to copy
BEQ     stop                     ; No words left to copy?
wordcopy
LDR     r3, [r0], #4             ; Load a word from the source and
STR     r3, [r1], #4             ; store it to the destination
SUBS    r2, r2, #1               ; Decrement the counter
BNE     wordcopy                 ; ... copy more
stop
MOV     r0, #0x18                ; angel_SWIreason_ReportException
LDR     r1, =0x20026             ; ADP_Stopped_ApplicationExit
SVC #0x123456                ; ARM semihosting (formerly SWI)
AREA    BlockData, DATA, READWRITE
src         DCD     1,2,3,4,5,6,7,8,1,2,3,4,5,6,7,8,1,2,3,4
dst         DCD     0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
END
```
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 7-32 for more details.

This section describes:

- `Test-and-branch macro example` on page 2-47
- `Unsigned integer division macro example` on page 2-47.
2.8.1 Test-and-branch macro example

In ARM code, and Thumb on a pre-Thumb-2 processor, a test-and-branch operation requires two ARM instructions to implement.

You can define a macro definition such as this:

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

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

This macro can be invoked as follows:

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

After substitution this becomes:

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

2.8.2 Unsigned integer division macro example

Example 2-10 on page 2-48 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

MACRO
$Lab DivMod $Div,$Top,$Bot,$Temp
   ASSERT $Top <> $Bot ; Produce an error message if the
   ASSERT $Top <> $Temp ; registers supplied are
   ASSERT $Bot <> $Temp ; not all different
   IF "$Div" <> "" ; These three only matter if $Div
      ASSERT $Div <> $Top ; is not null ("")
      ASSERT $Div <> $Bot
      ASSERT $Div <> $Temp ;
   ENDIF
90   $Lab
   MOV $Temp, $Bot ; Put divisor in $Temp
   CMP $Temp, $Top, LSR #1 ; double it until
   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
   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
91   CMP $Top, $Temp ; Halve $Temp,
   SUBCS $Top, $Top,$Temp ; and loop until
   ADC $Div, $Div, $Div
   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 2-13 for more
information.

Example 2-11 on page 2-49 shows the code that this macro produces if it is invoked as
follows:

    ratio DivMod r0,r5,r4,r2
Example 2-11

```
ASSERT r5 <> r4                   ; Produce an error if the
ASSERT r5 <> r2                   ; registers supplied are
ASSERT r4 <> r2                   ; not all different
ASSERT r0 <> r5                   ; These three only matter if $Div
ASSERT r0 <> r4                   ; is not null (""")
ASSERT r0 <> r2                   ;
ratio
MOV     r2, r4                     ; Put divisor in $Temp
CMP     r2, r5, LSR #1             ; double it until
90      MOVLS   r2, r2, LSL #1             ; 2 * r2 > r5
CMP     r2, r5, LSR #1             ;
BLS     %b90                       ; The b means search backwards
MOV     r0, #0                     ; Initialize quotient
91      CMP     r5, r2                     ; Can we subtract r2?
SUBCS   r5, r5, r2                 ; If we can, do so
ADC     r0, r0, r0                 ; Double r0
MOV     r2, r2, LSR #1             ; Halve r2,
CMP     r2, r4                     ; and loop until
BHS     %b91                       ; less than divisor
```
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 Chapter 4 Accessing Image Symbols in the Linker User 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 7-41 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 Procedure Call Standard for the ARM Architecture specification, aapcs.pdf, in install_directory\Documentation\Specifications\... for more information about stack checking qualifiers.

Be aware of the following:

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

Table 2-11 shows the main differences between UAL and the earlier ARM assembler language. The pre-UAL ARM syntax is accepted by the assembler.

<table>
<thead>
<tr>
<th>Change</th>
<th>Pre-UAL ARM syntax</th>
<th>Preferred syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>The default addressing mode for LDM and STM is IA</td>
<td>LDMIA, STMIA</td>
<td>LDM, STM</td>
</tr>
<tr>
<td>You can use the PUSH and POP mnemonics for full, descending stack operations in ARM as well as Thumb.</td>
<td>STMFD spl!, {reglist}</td>
<td>PUSH {reglist}</td>
</tr>
<tr>
<td>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.</td>
<td>MOV Rd, Rn, LSL shift</td>
<td>LSL Rd, Rn, shift</td>
</tr>
<tr>
<td></td>
<td>MOV Rd, Rn, LSR shift</td>
<td>LSR Rd, Rn, shift</td>
</tr>
<tr>
<td></td>
<td>MOV Rd, Rn, ASR shift</td>
<td>ASR Rd, Rn, shift</td>
</tr>
<tr>
<td></td>
<td>MOV Rd, Rn, ROR shift</td>
<td>ROR Rd, Rn, shift</td>
</tr>
<tr>
<td></td>
<td>MOV Rd, Rn, RRX</td>
<td>RRX Rd, Rn</td>
</tr>
<tr>
<td>You can use the label form for PC-relative addressing. Do not use the offset form in new code.</td>
<td>LDR Rd, [pc, #offset]</td>
<td>LDR Rd, label</td>
</tr>
<tr>
<td>Specify both registers for doubleword memory accesses. You must still obey rules about the register combinations you can use.</td>
<td>LDRD Rd, addr_mode</td>
<td>LDRD Rd, Rd2, addr_mode</td>
</tr>
<tr>
<td>{cond}, if used, is always the last element of all instructions.</td>
<td>ADD{cond}S</td>
<td>ADDS{cond}</td>
</tr>
<tr>
<td>LDR{cond}SB</td>
<td>LDRSB{cond}</td>
<td></td>
</tr>
<tr>
<td>You can use both ARM {cond} 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. If you omit the IT instructions, the assembler inserts them for you in Thumb-2 code.</td>
<td>ADDEQ r1, r2, r3</td>
<td>ITEQ E; optional</td>
</tr>
<tr>
<td>LDRNE r1, [r2, r3]</td>
<td>ADDEQ r1, r2, r3</td>
<td></td>
</tr>
<tr>
<td>LDRNE r1, [r2, r3]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In addition, some flexibility is permitted that was not permitted in previous assemblers (see Table 2-12).

<table>
<thead>
<tr>
<th>Relaxation</th>
<th>Permitted syntax</th>
<th>Preferred syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>If the destination register is the same as the first operand, you can use a two register form of the instruction.</td>
<td>ADD r1, r3</td>
<td>ADD r1, r1, r3</td>
</tr>
</tbody>
</table>

You can write source code for pre-Thumb-2 Thumb processors using UAL.
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 2-13 shows the main differences between pre-UAL Thumb assembly language and UAL. The assembler accepts the pre-UAL Thumb syntax only if it is preceded by a CODE16 directive, or if the source file is assembled with the --16 command-line option.

<table>
<thead>
<tr>
<th>Change</th>
<th>Pre-UAL Thumb syntax</th>
<th>UAL syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>The default addressing mode for LDM and STM is IA</td>
<td>LDMIA, STMIA</td>
<td>LDM, STM</td>
</tr>
</tbody>
</table>
| 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  
SUB r4, r5, #6  
MOV r0, #1  
LSR r1, r2, #1 | ADDS r1, r2, r3  
SUBS r4, r5, #6  
MOVS r0, #1  
LSRS r1, r2, #1 |
| The preferred form for ALU instructions specifies three registers, even if the destination register is the same as the first operand. | ADD r7, r8  
SUB r1, #80 | ADD r7, r7, r8  
SUBS r1, r1, #80 |
| If Rd and Rn are both Lo registers, MOV Rd, Rn is disassembled as ADDS Rd, Rn, #0. | MOV r2, r3  
MOV r8, r9  
CPY r0, r1  
LSL r2, r3, #0 | ADDS r2, r3, #0  
MOV r8, r9  
MOV r0, r1  
MOVS r2, r3 |
| NEG Rd, Rm is disassembled as RSBS Rd, Rm, #0. | NEG Rd, Rm | RSBS Rd, Rm, #0 |
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 3-22
- Predefined register and coprocessor names on page 3-23
- Built-in variables and constants on page 3-25
- Symbols on page 3-27
- Expressions, literals, and operators on page 3-33
- Diagnostic messages on page 3-45
- Using the C preprocessor on page 3-47.

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 and embedded 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. The ARM assembler uses the normal command line ordering rules as described in Ordering command-line options on page 2-10 in the Compiler User Guide. Therefore, if the command line contains options that conflict with each other, then the last one found always takes precedence.

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

- **--32**
  Is a synonym for --arm.

- **--apcs qualifier**
  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 3-10 for details.

- **--arm**
  Instructs the assembler to interpret instructions as ARM instructions. It does not, however, guarantee ARM-only code in the object file. This is the default.

- **--arm_only**
  Instructs the assembler to only generate ARM code. The resulting object file does not contain or permit any Thumb code.

- **--bi**
  Is a synonym for --bigend.

- **--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 3-18.
--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.

This option is deprecated. Use --diag_warning 1206 instead (see Controlling the output of diagnostic messages on page 3-18).

--compatible=name

Enables code generated by the assembler to be compatible with multiple processors or architectures. name is the CPU or architecture name that the code is to be compatible with, see CPU names on page 3-12.

--cpreproc

Instructs the assembler to call armcc to preprocess the input file before assembling it. See Using the C preprocessor on page 3-47.

--cpreproc_opts=options

Enables the assembler to pass compiler options to armcc when using the C preprocessor. See Using the C preprocessor on page 3-47. options is a comma-separated list of options and their values.

--cpu name

Sets the target CPU. See CPU names on page 3-12.

--debug

Instructs the assembler to generate DWARF debug tables. --debug is a synonym for -g.

The default is DWARF 3.

Note

Local symbols are not preserved with --debug. You must specify --keep if you want to preserve the local symbols to aid debugging.

--depend dependfile

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

--depend_format=string

Changes the format of output dependency files to UNIX-style format, for compatibility with some UNIX make programs.

The value of string can be one of:

unix Generates dependency files with UNIX-style path separators.
unix_escaped Is the same as unix, but escapes spaces with backslash.
unix_quoted

Is the same as unix, but surrounds path names with double quotes.

--device=list

Lists the supported device names that can be used with the --device=name option.

--device=name

Selects a specific device and sets the associated processor settings. See --device=name on page 2-44 in the Compiler Reference Guide.

--diag_error, --diag_remark, --diag_warning, --diag_suppress, --diag_style

See Controlling the output of diagnostic messages on page 3-18.

--dllexport_all

Gives all exported global symbols STV_PROTECTED visibility in ELF rather than STV_HIDDEN, unless overridden by source directives (see EXPORT or GLOBAL on page 7-78).

--dwarf2

Use with --debug, to instruct the assembler to generate DWARF 2 debug tables.

--dwarf3

Use with --debug, to instruct the assembler to generate DWARF 3 debug tables. This is the default if --debug is specified.

--errors errorfile

Instructs the assembler to output error messages to errorfile.

--exceptions

See Controlling exception table generation on page 3-20.

--exceptions_unwind

See Controlling exception table generation on page 3-20.

--execstack

Generates a .note.GNU-stack section marking the stack as executable.

You can use the AREA directive to generate an executable .note.GNU-stack section:

\[
\text{AREA } \cdot\text{note.GNU-stack},\text{ALIGN=0,READONLY,NOALLOC, CODE}
\]

In the absence of --execstack and --no_execstack, the .note.GNU-stack section is not generated unless it is specified by the AREA directive.
If both the command line option and source directive are used and are different, then the stack is marked as executable, see Table 3-1 on page 3-7.

See AREA on page 7-70.

`--fpmode model`

Specifies the floating-point conformance, and sets library attributes and floating-point optimizations. See Floating-point model on page 3-12.

`--fpu name`

Selects the target floating-point unit (FPU) architecture. See FPU names on page 3-14.

`-g`

Is a synonym for --debug.

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

Adds directories to the source file have to be fully qualified (see GET or INCLUDE on page 7-81).

`--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 7-85).

`--length`

See Listing output to a file on page 3-17

`--li`

Is a synonym for --littleend.

`--library_type=lib`

Enables the relevant library selection to be used at link time.

Where `lib` can be one of:

- `standardlib` Specified that the full ARM runtime libraries are selected at link time. This is the default.
- `microlib` Specified that the C micro-library (microlib) is selected at link time.

--- Note ---

This option can be used with the compiler, assembler or linker when use of the libraries require more specialized optimizations.

Use this option with the linker to override all other --library_type options.

---

For more information see:

- Building an application with microlib on page 3-4 in the Libraries Guide
- --library_type=lib on page 2-81 in the Compiler Reference Guide.
--licretry  
Instructs the assembler to retry checking out licenses up to 10 times, approximately 10 seconds apart, upon encountering certain FLEXnet error codes.

--list file  
Instructs the assembler to output a detailed listing of the assembly language produced by the assembler to file. See Listing output to a file on page 3-17 for details.

--littleend  
Instructs the assembler to assemble code suitable for a little-endian ARM.

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

--maxcache n  
Sets the maximum source cache size to n bytes. The default is 8MB. armasm gives a warning if size is less than 8MB.

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

--memaccess attributes  
Specifies memory access attributes of the target memory system. See Memory access attributes on page 3-16.

Note  
The --memaccess option is deprecated.

--no_code_gen  
Instructs the assembler to exit after pass 1. No object file is generated.

--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 3-20.

--no_exceptions_unwind  
See Controlling exception table generation on page 3-20.

--no_execstack  
Generates a .note.GNU-stack section marking the stack as non-executable.

You can use the AREA directive to generate a non executable .note.GNU-stack section:

```
AREA | .note.GNU-stack |,ALIGN=0,READONLY,NOALLOC
```
In the absence of \texttt{--execstack} and \texttt{--no_execstack}, the \_note\_GNU-stack section is not generated unless it is specified by the \texttt{AREA} directive.

If both the command line option and source directive are used and are different, then the stack is marked as executable.

\begin{table}[h]
\centering
\begin{tabular}{lll}
\hline
 & \texttt{--execstack} & \texttt{--no_execstack} \\
\hline
\texttt{execstack} AREA directive & execstack & execstack \\
\texttt{no_execstack} AREA directive & execstack & no\_execstack \\
\hline
\end{tabular}
\caption{Specifying a command line option and an \texttt{AREA} directive for GNU-stack sections}
\end{table}

See \texttt{AREA} on page 7-70.

\texttt{--no\_hide\_all}

Gives all exported and imported global symbols STV\_DEFAULT visibility in ELF rather than STV\_HIDDEN, unless overridden by source directives (see \textit{EXPORT or GLOBAL} on page 7-78 and \textit{IMPORT and EXTERN} on page 7-82).

\texttt{--no\_regs}

Instructs the assembler not to redefine register names. See \textit{Predefined register and coprocessor names} on page 3-23 for a list of predefined register names.

This option is deprecated. Use \texttt{--regnames=none} instead.

\texttt{--no\_terse}

See \textit{Listing output to a file} on page 3-17

\texttt{--no\_unaligned\_access}

Instructs the assembler to set an attribute in the object file indicating that unaligned accesses are not used.

\texttt{--no\_warn}

Turns off warning messages.

\texttt{-o filename}

Names the output object file. If this option is not specified, the assembler creates an object filename of the form \textit{inputfilename.o}. This option is case-sensitive.

\texttt{--pd}

Is a synonym for \texttt{--predefine}. 

--predefine "directive"
Instructs the assembler to pre-execute one of the SET directives. See
Pre-executing a SET directive on page 3-16 for details. This is useful for
conditional assembly (see Assembly conditional on a variable being
declared on page 7-39).

--project=filename, --no_project=filename
See Project template options on page 3-18.

--reduce_paths, --no_reduce_paths
Enables or disables the elimination of redundant pathname information
in file paths. This option is valid for Windows systems only.
Windows systems impose a 260 character limit on file paths. Where
relative pathnames exist whose absolute names expand to longer than 260
characters, you can use the --reduce_paths option to reduce absolute
pathname length by matching up directories with corresponding
instances of .. and eliminating the directory/.. sequences in pairs.

Note
It is recommended that you avoid using long and deeply nested file paths,
in preference to minimizing path lengths using the --reduce_paths option.

See --reduce_paths, --no_reduce_paths on page 2-109 in the Compiler
Reference Guide for more information.

--regnames=none
Instructs the assembler not to predefine register names. See Predefined
register and coprocessor names on page 3-23 for a list of predefined
register names.

--regnames=callstd
Defines additional register names based on the AAPCS variant that you
are using as specified by the --apcs option (see AAPCS on page 3-10 for
details).

--regnames=all
Defines all AAPCS registers regardless of the value of --apcs (see
AAPCS on page 3-10 for details).

--reinitialize_workdir
See Project template options on page 3-18.
--report-if-not-wysiwyg

Instructs the assembler to report when the assembler outputs an encoding that was not directly requested in the source code. This can happen when the assembler:

- uses a pseudo-instruction that is not available in other assemblers, for example MOV32
- outputs an encoding that does not directly match the instruction mnemonic, for example if the assembler outputs the MVN encoding when assembling the MOV instruction
- inserts additional instructions where necessary for instruction syntax semantics, for example the assembler can insert a missing IT instruction before a conditional Thumb instruction.

--show_cmdline

Shows how the assembler has processed the command line. The commands are shown normalized, and the contents of any via files are expanded.

--split_ldm

Instructs the assembler to fault long LDM and STM instructions. See Splitting long LDMs and STMs on page 3-16 for details. Use of this option is deprecated.

--thumb

Instructs the assembler to interpret instructions as Thumb instructions, using the UAL syntax. This is equivalent to a THUMB directive at the head of the source file.

--unaligned_access

Instructs the assembler to set an attribute in the object file indicating the use of unaligned accesses.

--unsafe

Enables instructions from differing architectures to be assembled without error. See Controlling the output of diagnostic messages on page 3-18).

--untyped_local_labels

Forces the assembler not to set the Thumb bit when referencing labels in Thumb code. See LDR pseudo-instruction on page 4-159 for details.

--via file

Instructs the assembler to open file and read in command-line arguments to the assembler. For more information see Appendix A Via File Syntax in the Compiler Reference Guide.

--width

See Listing output to a file on page 3-17.
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 Specifying command-line options with an environment variable

You can specify command-line options by setting the value of the RVCT40_ASMOPT environment variable. The syntax is identical to the command line syntax. The assembler reads the value of RVCT40_ASMOPT and inserts it at the front of the command string. This means that options specified in RVCT40_ASMOPT can be overridden by arguments on the command-line.

3.1.3 AAPCS

There is an option to specify whether you are using the Procedure Call Standard for the ARM Architecture (AAPCS):

```
--apcs qualifier
```

The AAPCS forms part of the Base Standard Application Binary Interface for the ARM Architecture (BSABI) specification. By writing code that adheres to the AAPCS, you can ensure that separately compiled and assembled modules can work together.

The --apcs option 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\...

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 inputfile does not use AAPCS. AAPCS registers are not set up. Other qualifiers are not permitted if you use none.

/interwork  Specifies that the code in inputfile is suitable for ARM and Thumb interworking. See Chapter 5 Interworking ARM and Thumb in the Developer Guide for information.

/nointerwork  Specifies that the code in inputfile is not suitable for ARM and Thumb interworking. This is the default.

/inter  Is a synonym for /interwork.

/nointer  Is a synonym for /nointerwork.

/ropi  Specifies that the content of inputfile is read-only position-independent.

/noropi  Specifies that the content of inputfile 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 inputfile is read-write position-independent.

/norwpi  Specifies that the content of inputfile 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 inputfile is read-only position-independent code that requires FPIC addressing.

--- Note ---
If you specify more than one qualifier, ensure that there are no spaces or commas between the individual qualifiers in the list.
3.1.4 Floating-point model

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

```
--fpmode model
```

Selects the target floating-point model and sets attributes to select the most suitable library when linking.

--- Note ---
This does not cause any changes to the code that you write.

`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.
- **ieee_fixed**: IEEE standard with round-to-nearest and no inexact exception.
- **ieee_no_fenv**: IEEE standard with round-to-nearest and no exceptions. This mode is compatible with the Java floating-point arithmetic model.
- **std**: IEEE finite values with denormals flushed to zero, round-to-nearest and no exceptions. It is C and C++ compatible. This is the default option. Finite values are as predicted by the IEEE standard. It is not guaranteed that NaNs and infinities are produced in all circumstances defined by the IEEE model, or that when they are produced, they have the same sign. Also, it is not guaranteed that the sign of zero is that predicted by the IEEE model.
- **fast**: Some value altering optimizations, where accuracy is sacrificed to fast execution. This is not IEEE compatible, and is not standard C.

3.1.5 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 3-18).

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®.
Note

Using --cpu=7 generates code that is supported by the ARMv7-A, ARMv7-R, and ARMv7-M architectures. This means that the assembler is restricted to producing only the Thumb instructions that are available on the ARMv7A, ARMv7-R, and ARMv7-M architectures.

There is an option to specify another CPU or architecture name that the code is to be compatible with:

--compatible=name

Specifies a second processor or architecture, name, for which the assembler generates compatible code.

When you specify a processor or architecture name using --compatible, valid values of name for both the --cpu and --compatible options are restricted to those shown in Table 3-2 and must not be from the same group. For example:

```
armasm --cpu=arm7tdmi --compatible=cortex-m3 myprog.asm
```

Table 3-2 Compatible processor or architecture combinations

<table>
<thead>
<tr>
<th>Group 1</th>
<th>ARM7TDMI, 4T</th>
</tr>
</thead>
<tbody>
<tr>
<td>Group 2</td>
<td>Cortex-M0, Cortex-M1, Cortex-M3, Cortex-M4, 7-M, 6-M, 6S-M</td>
</tr>
</tbody>
</table>

Specify --compatible=NONE to turn off all previous instances of the option on the command line.

See the Linker User 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 and architecture names by invoking the assembler with the following command:

```
armasm --cpu list
```
3.1.6 FPU names

There is an option to specify the FPU name:

```bash
--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. The assembler produces an error if the FPU you specify explicitly is incompatible with the CPU. Floating-point instructions also 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 object files built with any FPU.
- `vfpv3` Selects hardware floating-point unit conforming to architecture VFPv3.
- `vfpv3_fp16` Selects hardware floating-point unit conforming to architecture VFPv3 with half-precision floating-point extension.
- `vfpv3_d16` Selects hardware floating-point unit conforming to architecture VFPv3-D16.
- `vfpv3_d16_fp16` Selects hardware floating-point unit conforming to architecture VFPv3-D16 with half-precision floating-point extension.
- `vfpv4` Selects hardware floating-point unit conforming to architecture VFPv4.
- `vfpv4_d16` Selects hardware floating-point unit conforming to architecture VFPv4-D16.
- `fpv4_sp` Selects hardware floating-point unit conforming to the single precision variant of architecture FPv4.
- `vfpv2` Selects hardware floating-point unit conforming to architecture VFPv2.
- `softvfp` Selects software floating-point linkage. This is the default if you do not specify a `--fpu` option and the `--cpu` option selected does not imply a particular FPU.
- `softvfp+vfpv2` Selects a floating-point library with software floating-point linkage that uses VFP instructions.
This is otherwise equivalent to using --fpu vfpv2.

softvfp+vfpv3 Selects a floating-point library with software floating-point
linkage that uses VFP instructions.
This is otherwise equivalent to using --fpu vfpv3.

softvfp+vfpv3_fp16 Selects a floating-point library with software floating-point
linkage that uses VFP instructions.
This is otherwise equivalent to using --fpu vfpv3_fp16.

softvfp+vfpv3_d16 Selects a floating-point library with software floating-point
linkage that uses VFP instructions.
This is otherwise equivalent to using --fpu vfpv3_d16.

softvfp+vfpv3_d16_fp16 Selects a floating-point library with software floating-point
linkage that uses VFP instructions.
This is otherwise equivalent to using --fpu vfpv3_d16_fp16.

softvfp+vfpv4 Selects a floating-point library with software floating-point
linkage that uses VFP instructions.
This is otherwise equivalent to using --fpu vfpv4.

softvfp+vfpv4_d16 Selects a floating-point library with software floating-point
linkage that uses VFP instructions.
This is otherwise equivalent to using --fpu vfpv4_d16.

softvfp+fpv4_sp Selects a floating-point library with software floating-point
linkage that uses VFP instructions.
This is otherwise equivalent to using --fpu fpv4_sp.

See the Linker User 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
3.1.7 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.

**Note**
The `--memaccess` option is deprecated.

3.1.8 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 7-7. The assembler also executes a corresponding `GBLL`, `GBLS`, 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`.

3.1.9 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.

### 3.1.10 Listing output to a file

Use the following option to list output to a file:

```
--list file
```

This instructs the assembler to output a detailed listing of the assembly language produced by the assembler to `file`.

If `-` is given as `file`, listing is sent to `stdout`.

If no `file` is given, use `--list=` to send the output to `inputfile.lst`.

________ Note

You can use `--list` without a filename to send the output to `inputfile.lst`. However, this syntax is deprecated and the assembler issues a warning. This syntax will be removed in a later release. Use `--list=` instead.

________

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 unpaged 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.

### 3.1.11 Project template options

Project templates are files containing project information such as command-line options for a particular configuration. These files are stored in the project template working directory. The following options control the use of project templates.

---project=filename, --no_project=filename  Enables or disables the use of a project template file.

---reinitialize_workdir  Enables you to reinitialize the project template working directory.

---workdir=directory  Enables you to provide a working directory for a project template.

For more information on each of these options, see the following in the Compiler Reference Guide:

- --project=filename, --no_project on page 2-107
- --reinitialize_workdir on page 2-110
- --workdir=directory on page 2-136.

### 3.1.12 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|gnu}  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.

gnu Display messages using the GNU style.

Choosing the option --diag_style=idc implicitly selects the option --brief_diagnostics. Explicitly selecting --no_brief_diagnostics on the command line overrides the selection of --brief_diagnostics implied by --diag_style=idc.

Selecting either the option --diag_style=arm or the option --diag_style=gnu does not imply any selection of --brief_diagnostics.

--diag_error tag{,tag,...}

Sets the diagnostic messages that have the specified tag(s) to the error severity (see Table 3-3 on page 3-20).

--diag_remark tag{,tag,...}

Sets the diagnostic messages that have the specified tag(s) to the remark severity (see Table 3-3 on page 3-20).

--diag_warning tag{,tag,...}

Sets the diagnostic messages that have the specified tag(s) to the warning severity (see Table 3-3 on page 3-20).

--diag_suppress tag{,tag,...}

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

--unsafe Enables instructions from differing architectures to be assembled without error. It changes corresponding error messages to warning messages. It also suppresses warnings about operator precedence (see Binary operators on page 3-40).

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 ...

The assembler prefix A can be used with --diag_error, --diag_remark, and --diag_warning, or when suppressing messages, for example:

armasm --diag_suppress A1293,A187 ...

Diagnostic messages can be cut and paste directly into a command line. Using the prefix letter is optional. However, if a prefix letter is included, it must match the armasm identification letter. If another prefix is found, the assembler ignores the message number.
Table 3-3 explains the meaning of the term **severity** used in the option descriptions.

<table>
<thead>
<tr>
<th>Severity</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Catastrophic error</td>
<td>Catastrophic errors indicate problems that cause the assembly to stop. These errors include command-line errors, internal errors, and missing include files.</td>
</tr>
<tr>
<td>Error</td>
<td>Errors indicate violations in the syntactic or semantic rules of assembly language. Assembly continues, but object code is not generated.</td>
</tr>
<tr>
<td>Warning</td>
<td>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.</td>
</tr>
<tr>
<td>Remark</td>
<td>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.</td>
</tr>
</tbody>
</table>

### 3.1.13 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. This is the default.

--exceptions_unwind

Instructs the assembler to produce *unwind* tables for functions where possible. This is the default.

--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 7-53 and FRAME UNWIND OFF on page 7-53.

### 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 function 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 instructions, pseudo-instructions, or directives in all uppercase, as in this manual. Alternatively, you can write them in all lowercase. You must not write an instruction, pseudo-instruction, or 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 3-30 and Local labels on page 3-31). 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 3-27).
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, $r9$)
- 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 extension register names

The following extension register names are predeclared:

- $q0$-$q15$ and $Q0$-$Q15$ (NEON™ Quadword registers)
- $d0$-$d31$ and $D0$-$D31$ (NEON Doubleword registers, VFP double-precision registers)
- $s0$-$s31$ and $S0$-$S31$ (VFP single-precision registers).

3.3.3 Predeclared XScale register names

The following register names are predeclared when assembling for an Intel XScale CPU:

- $acc0$-$acc7$ and $ACC0$-$ACC7$ (XScale accumulators).

The following register names are predeclared when assembling for an Intel XScale CPU with Wireless MMX:

- $wr0$-$wr15$, $wr0$-$wr15$, and $wr0$-$wr15$
- $wC0$-$wC15$, $wC0$-$wC15$, and $wC0$-$wC15$
- $w CID$, $wcid$, and $w CID$
- $wCON$, $wcon$, and $wCON$
- $wCSSF$, $wcssf$, and $wCSSF$
- $wCASF$, $wcasf$, and $wCASF$. 
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 3-4 lists the built-in variables defined by the ARM assembler.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ARCHITECTURE}</td>
<td>Holds the name of the selected ARM architecture.</td>
</tr>
<tr>
<td>{AREANAME}</td>
<td>Holds the name of the current AREA.</td>
</tr>
<tr>
<td>{ARMASM_VERSION}</td>
<td>Holds an integer that increases with each version of armasm.</td>
</tr>
<tr>
<td>{ads$version}</td>
<td>Has the same value as {ARMASM_VERSION}.</td>
</tr>
<tr>
<td>{CODESIZE}</td>
<td>Is a synonym for {CONFIG}.</td>
</tr>
<tr>
<td>{COMMANDLINE}</td>
<td>Holds the contents of the command line.</td>
</tr>
<tr>
<td>{CONFIG}</td>
<td>Has the value 32 if the assembler is assembling ARM code, or 16 if it is assembling Thumb code.</td>
</tr>
<tr>
<td>{CPU}</td>
<td>Holds the name of the selected CPU. The default is &quot;ARM7TDMI&quot;. If an architecture was specified in the command line --cpu option, (CPU) holds the value &quot;Generic ARM&quot;.</td>
</tr>
<tr>
<td>{ENDIAN}</td>
<td>Has the value &quot;big&quot; if the assembler is in big-endian mode, or &quot;little&quot; if it is in little-endian mode.</td>
</tr>
<tr>
<td>{FPIC}</td>
<td>Has the boolean value True if /fpic is set. The default is False.</td>
</tr>
<tr>
<td>{FPU}</td>
<td>Holds the name of the selected FPU. The default is &quot;SoftVFP&quot;.</td>
</tr>
<tr>
<td>{INPUTFILE}</td>
<td>Holds the name of the current source file.</td>
</tr>
<tr>
<td>{INTER}</td>
<td>Has the boolean value True if /inter is set. The default is False.</td>
</tr>
<tr>
<td>{LINENUM}</td>
<td>Holds an integer indicating the line number in the current source file.</td>
</tr>
<tr>
<td>{OPT}</td>
<td>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.</td>
</tr>
<tr>
<td>{PC} or .</td>
<td>Address of current instruction.</td>
</tr>
<tr>
<td>{PCSTOREOFFSET}</td>
<td>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.</td>
</tr>
<tr>
<td>{ROPI}</td>
<td>Has the boolean value True if /ropi is set. The default is False.</td>
</tr>
<tr>
<td>{RWPI}</td>
<td>Has the boolean value True if /rwpi is set. The default is False.</td>
</tr>
<tr>
<td>{VAR} or @</td>
<td>Current value of the storage area location counter.</td>
</tr>
</tbody>
</table>
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"
```

The built-in variable `|ads$version|` must be all in lowercase. The names of the other built-in variables can be in uppercase, lowercase, or mixed. For example:

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

___ Note ___

All built-in string variables contain case-sensitive values. See CPU names on page 3-12 for valid values for `{CPU}` and `{ARCHITECTURE}`. See FPU names on page 3-14 for valid values for `{FPU}`. Relational operations on these built-in variables will not match with strings that contain an incorrect case.

Table 3-5 lists the built-in Boolean constants defined by the ARM assembler.

<table>
<thead>
<tr>
<th>Boolean Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>{FALSE}</td>
<td>Logical constant false.</td>
</tr>
<tr>
<td>{TRUE}</td>
<td>Logical constant true.</td>
</tr>
</tbody>
</table>

### 3.4.1 Detecting versions of armasm

You can use the built-in variable `{ARMASM_VERSION}` to distinguish between versions of armasm. The format of the version number is `PVtbbbb` where:

- `P` is the major version
- `V` is the minor version
- `t` is the patch release
- `bbbb` is the build number

The ARM assembler did not have the built-in variable `|ads$version|` before ADS and RVCT. If you have to build versions of your code for legacy development tools, you can test for the built-in variable `|ads$version|`. Use code similar to the following:

```
IF :DEF: |ads$version|
  ; code for RealView 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 3-28
• Numeric constants on page 3-28
• Assembly time substitution of variables on page 3-28
• Labels on page 3-30
• Local labels on page 3-31.

3.5.1 Symbol naming rules

The following general rules apply to symbol names:

• Symbol names must be unique within their scope.
• You can use uppercase letters, lowercase letters, numeric characters, or the underscore character in symbol names. Symbol names are case-sensitive, and all characters in the symbol name are significant.
• Do not use numeric characters for the first character of symbol names, except in local labels (see Local labels on page 3-31).
• Symbols must not use the same name as built-in variable names or predefined symbol names (see Predefined register and coprocessor names on page 3-23 and Built-in variables and constants on page 3-25).
• If you use the same name as an instruction mnemonic or directive, use double bars to delimit the symbol name. For example:

  ||ASSERT||

  The bars are not part of the symbol.
• You must not use the symbols |$a|, |$t|, |$t.x|, or |$d| as program labels. These are mapping symbols used to mark ARM, Thumb, ThumbEE, and data within the object file.

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 3-34).

The possible values of a logical variable are \{TRUE\} or \{FALSE\} (see Logical expressions on page 3-37).

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 3-33).

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

- GBLA, GBLL, and GBLS on page 7-4
- LCLA, LCLL, and LCLS on page 7-6
- SETA, SETL, and SETS on page 7-7.

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 $2^{31} - 1$. 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 7-77). You cannot change the value of a numeric constant after you define it. You can create numeric constant expressions by combining numeric constants and binary operators.

See also Numeric expressions on page 3-34 and Numeric literals on page 3-35.

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 \text{T} or \text{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 \textit{Symbol naming rules} on page 3-27). 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.

\textbf{Examples}

; straightforward substitution
  GBLS add4ff

; elaborate substitution
  GBLS s1
  GBLS s2
  GBLS fixup
  GBLA count
  ...
  count SETA 14
  s1 SETS "a$s$count" ; s1 now has value a$s0000000E
  s2 SETS "abc"
  fixup SETS "|xy$s2.z|" ; fixup now has value |xyabcz|
  |CSScode| MOV r4,#16 ; but the label here is CSScode
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. Labels are local to the source file unless you make them global using the `EXPORT` directive, see `EXPORT` or `GLOBAL` on page 7-78.

**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 7-21
- `DCD` and `DCDU` on page 7-22
- `DCFD` and `DCFDU` on page 7-24
- `DCFS` and `DCFSU` on page 7-25
- `DCI` on page 7-26
- `DCQ` and `DCQU` on page 7-27
- `DCW` and `DCWU` on page 7-28.

You can also use the section name of an `AREA` directive as a label for program-relative addresses. In this case the label points to the start of the specified `AREA`, see `AREA` on page 7-70. Do not use `AREA` names as branch targets.

**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 7-18
- `SPACE` or `FILL` on page 7-20
- `DCDO` on page 7-23
- `EQU` on page 7-77.

**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 area.

A local label can be used in place of symbol in source lines in an assembly language module (see Format of source lines on page 3-22):

- on its own, that is, where there is no instruction or directive
- on a line that contains an instruction
- on a line that contains a code- or data-generating directive.

A local label is generally used where you might use a program-relative label (see Labels on page 3-30).

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 7-32).

Use the ROUT directive to limit the scope of local labels (see ROUT on page 7-88). 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:

\[ n\{\text{routname}\} \]

The syntax of a reference to a local label is:

\[ %\{F|B\}{A|T}\{n\{\text{routname}\} \]

where:

- \( n \) is the number of the local label.
- \( \text{routname} \) 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 \( \text{routname} \) is specified in either a label or a reference to a label, the assembler checks it against the name of the nearest preceding \( \text{ROUT} \) 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 3-34
- Numeric expressions on page 3-34
- Numeric literals on page 3-35
- Floating-point literals on page 3-36
- Register-relative and program-relative expressions on page 3-37
- Logical expressions on page 3-37
- Logical literals on page 3-37
- Operator precedence on page 3-37
- Unary operators on page 3-39
- Binary operators on page 3-40.

3.6.1 String expressions

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

- Variables on page 3-28
- String literals on page 3-34
- Unary operators on page 3-39
- String manipulation operators on page 3-41
- SETA, SETL, and SETS on page 7-7.

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 3-22).

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 3-28
• Variables on page 3-28
• Numeric literals on page 3-35
• Binary operators on page 3-40
• SETA, SETL, and SETS on page 7-7.

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 \(\geq\) use the unsigned interpretation. This means that 0 \(\geq\) \(-1\) 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

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>SETA</td>
<td>34906</td>
</tr>
<tr>
<td>addr</td>
<td>DCD</td>
<td>0xA10E</td>
</tr>
<tr>
<td></td>
<td>LDR</td>
<td>r4,=&amp;1000000F</td>
</tr>
<tr>
<td></td>
<td>DCD</td>
<td>2_11001010</td>
</tr>
<tr>
<td>c3</td>
<td>SETA</td>
<td>8_74007</td>
</tr>
<tr>
<td></td>
<td>DCQ</td>
<td>0x0123456789abcdef</td>
</tr>
<tr>
<td></td>
<td>LDR</td>
<td>r1,'A'   ; pseudo-instruction loading 65 into r1</td>
</tr>
<tr>
<td></td>
<td>ADD</td>
<td>r3,r2,#&quot;&quot; ; add 39 to contents of r2, result to r3</td>
</tr>
</tbody>
</table>
3.6.5 Floating-point literals

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

\{-\}digitsE\{-\}digits
\{-\}digits\}.digits
\{-\}digits\}.digitsE\{-\}digits
0\hexdigits
&\hexdigits
0f_hexdigits
0d_hexdigits

where:

digits  Are sequences of characters using only the digits 0 to 9. You can write E in uppercase or lowercase. These forms correspond to normal floating-point notation.

\hexdigits  Are sequences of characters using only the digits 0 to 9 and the letters A to F or a to f. These forms correspond to the internal representation of the numbers in the computer. Use these forms to enter infinities and NaNs, or if you want to be sure of the exact bit patterns you are using.

The 0x and & forms allow the floating-point bit pattern to be specified by any number of hex digits.

The 0f_ form requires the floating-point bit pattern to be specified by exactly 8 hex digits.

The 0d_ form requires the floating-point bit pattern to be specified by exactly 16 hex digits.

The range for single-precision floating-point values is:

- maximum 3.40282347e+38
- minimum 1.17549435e–38.

The range for double-precision floating-point values is:

- maximum 1.79769313486231571e+308
- minimum 2.22507385850720138e–308.

**Examples**

```assembly
DCFD    1E308,-4E-100
DCFS    1.0
DCFD    3.725e15
DCFS    0x7FC00000              ; Quiet NaN
DCFD    &FF00000000000000       ; 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 7-18).

A program relative address is expressed as an offset from the current Program Counter (PC). It is normally a label combined with a numeric expression.

The following steps shows what the program-relative address evaluates to:
1. the address of the instruction following the currently executing instruction
2. bitwise OR with 0xFFFFFFFFC (this makes no difference in ARM code)
3. plus or minus the numeric constant.

Example

```
LDR r4,=data+4*n ; n is an assembly-time variable
; code
MOV pc,lr
data DCD value_0
; n-1 DCD directives
DCD value_n ; data+4*n points here
; more DCD directives
```

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 3-44).

Relations consist of combinations of variables, literals, constants, or expressions with appropriate relational operators (see Relational operators on page 3-42).

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 3-39 and Binary operators on page 3-40).
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, and you are not using the --unsafe option, armasm normally gives a warning:

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

Table 3-6 shows the order of precedence of operators in armasm, and a comparison with the order in C (see Table 3-7 on page 3-39).

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>
<thead>
<tr>
<th>armasm precedence</th>
<th>equivalent C operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>unary operators</td>
<td>unary operators</td>
</tr>
<tr>
<td>* / :MOD:</td>
<td>* / %</td>
</tr>
<tr>
<td>string manipulation</td>
<td>n/a</td>
</tr>
<tr>
<td>:SHL: :SHR: :ROR: :ROL:</td>
<td>&lt;&lt; &gt;&gt;</td>
</tr>
<tr>
<td>= &gt; &gt;= &lt; &lt;= /= &lt;&gt;</td>
<td>== &gt; &gt;= &lt; &lt;= !=</td>
</tr>
</tbody>
</table>
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 3-8 lists the unary operators that return strings.

Table 3-8 Unary operators that return strings

<table>
<thead>
<tr>
<th>Operator</th>
<th>Usage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>:CHR:</td>
<td>:CHR:A</td>
<td>Returns the character with ASCII code A.</td>
</tr>
<tr>
<td>:LOWERCASE:</td>
<td>:LOWERCASE:string</td>
<td>Returns the given string, with all uppercase characters converted to lowercase.</td>
</tr>
<tr>
<td>:REVERSE_CC:</td>
<td>:REVERSE_CC:cond_code</td>
<td>Returns the inverse of the condition code in cond_code, or an error if cond_code does not contain a valid condition code.</td>
</tr>
<tr>
<td>:STR:</td>
<td>:STR:A</td>
<td>Returns an 8-digit hexadecimal string corresponding to a numeric expression, or the string &quot;T&quot; or &quot;F&quot; if used on a logical expression.</td>
</tr>
<tr>
<td>:UPPERCASE:</td>
<td>:UPPERCASE:string</td>
<td>Returns the given string, with all lowercase characters converted to uppercase.</td>
</tr>
</tbody>
</table>

Table 3-7 Operator precedence in C

<table>
<thead>
<tr>
<th>C precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>unary operators</td>
</tr>
<tr>
<td>* / %</td>
</tr>
<tr>
<td>+ - (as binary operators)</td>
</tr>
<tr>
<td>&lt;&lt; &gt;&gt;</td>
</tr>
<tr>
<td>&lt; &lt;= &gt; &gt;=</td>
</tr>
<tr>
<td>== !=</td>
</tr>
<tr>
<td>&amp;</td>
</tr>
<tr>
<td>^</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>&amp;&amp;</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
Table 3-9 lists the unary operators that return numeric values.

### Table 3-9 Unary operators that return numeric or logical values

<table>
<thead>
<tr>
<th>Operator</th>
<th>Usage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>?</td>
<td>?A</td>
<td>Number of bytes of executable code generated by line defining symbol A.</td>
</tr>
<tr>
<td>+ and -</td>
<td>+A, -A</td>
<td>Unary plus. Unary minus. + and – can act on numeric and program-relative expressions.</td>
</tr>
<tr>
<td>:BASE:</td>
<td>:BASE:A</td>
<td>If A is a PC-relative or register-relative expression, :BASE: returns the number of its register component. :BASE: is most useful in macros.</td>
</tr>
<tr>
<td>:CC_ENCODING:</td>
<td>:CC_ENCODING:cond_code</td>
<td>Returns the numeric value of the condition code in cond_code, or an error if cond_code does not contain a valid condition code.</td>
</tr>
<tr>
<td>:DEF:</td>
<td>:DEF:A</td>
<td>{TRUE} if A is defined, otherwise {FALSE}.</td>
</tr>
<tr>
<td>:INDEX:</td>
<td>:INDEX:A</td>
<td>If A is a register-relative expression, :INDEX: returns the offset from that base register. :INDEX: is most useful in macros.</td>
</tr>
<tr>
<td>:LEN:</td>
<td>:LEN:A</td>
<td>Length of string A.</td>
</tr>
<tr>
<td>:LNOT:</td>
<td>:LNOT:A</td>
<td>Logical complement of A.</td>
</tr>
<tr>
<td>:NOT:</td>
<td>:NOT:A</td>
<td>Bitwise complement of A (- is an alias, for example ~A).</td>
</tr>
<tr>
<td>:RCONST:</td>
<td>:RCONST:Rn</td>
<td>Number of register, 0-15 corresponding to r0-r15.</td>
</tr>
</tbody>
</table>

### 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 3-38.

---

**Multiplicative operators**

Multiplicative operators have the highest precedence of all binary operators. They act only on numeric expressions.
Table 3-10 shows the multiplicative operators.

**Table 3-10 Multiplicative operators**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Alias</th>
<th>Usage</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>A*B</td>
<td>Multiply</td>
<td></td>
</tr>
<tr>
<td>/</td>
<td>A/B</td>
<td>Divide</td>
<td></td>
</tr>
<tr>
<td>:MOD:</td>
<td>%</td>
<td>A:MOD:B A modulo B</td>
<td></td>
</tr>
</tbody>
</table>

**String manipulation operators**

Table 3-11 shows the string manipulation operators. In CC, both A and B must be strings. In the slicing operators LEFT and RIGHT:
- A must be a string
- B must be a numeric expression.

**Table 3-11 String manipulation operators**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Usage</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>:CC:</td>
<td>A:CC:B</td>
<td>B concatenated onto the end of A</td>
</tr>
<tr>
<td>:LEFT:</td>
<td>A:LEFT:B</td>
<td>The left-most B characters of A</td>
</tr>
<tr>
<td>:RIGHT:</td>
<td>A:RIGHT:B</td>
<td>The right-most B characters of A</td>
</tr>
</tbody>
</table>

**Shift operators**

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

Table 3-12 shows the shift operators.

**Table 3-12 Shift operators**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Alias</th>
<th>Usage</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>:ROL:</td>
<td>A:ROL:B</td>
<td>Rotate A left by B bits</td>
<td></td>
</tr>
<tr>
<td>:ROR:</td>
<td>A:ROR:B</td>
<td>Rotate A right by B bits</td>
<td></td>
</tr>
<tr>
<td>:SHL:</td>
<td>&lt;&lt; A:SHL:B</td>
<td>Shift A left by B bits</td>
<td></td>
</tr>
<tr>
<td>:SHR:</td>
<td>&gt;&gt; A:SHR:B</td>
<td>Shift A right by B bits</td>
<td></td>
</tr>
</tbody>
</table>
--- 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 3-13 shows addition, subtraction, and logical operators.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Alias</th>
<th>Usage</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>A+B</td>
<td>A+B</td>
<td>Add A to B</td>
</tr>
<tr>
<td>-</td>
<td>A-B</td>
<td>A-B</td>
<td>Subtract B from A</td>
</tr>
<tr>
<td>:EOR:</td>
<td>^</td>
<td>A:EOR:B</td>
<td>Bitwise Exclusive OR of A and B</td>
</tr>
<tr>
<td>:OR:</td>
<td>^</td>
<td>A:OR:B</td>
<td>Bitwise OR of A and B</td>
</tr>
</tbody>
</table>

The use of | as an alias for :OR: is deprecated.

Relational operators

Table 3-14 on page 3-43 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>
<thead>
<tr>
<th>Operator</th>
<th>Alias</th>
<th>Usage</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>==</td>
<td>A=B</td>
<td>A equal to B</td>
</tr>
<tr>
<td>&gt;</td>
<td></td>
<td>A&gt;B</td>
<td>A greater than B</td>
</tr>
<tr>
<td>&gt;=</td>
<td></td>
<td>A&gt;=B</td>
<td>A greater than or equal to B</td>
</tr>
<tr>
<td>&lt;</td>
<td></td>
<td>A&lt;B</td>
<td>A less than B</td>
</tr>
<tr>
<td>&lt;=</td>
<td></td>
<td>A&lt;=B</td>
<td>A less than or equal to B</td>
</tr>
<tr>
<td>/=</td>
<td>&lt;&gt;</td>
<td>A/=B</td>
<td>A not equal to B</td>
</tr>
</tbody>
</table>
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 3-15 shows the Boolean operators.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Alias</th>
<th>Usage</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>:LAND:</td>
<td>&amp;&amp;</td>
<td>A:LAND:B</td>
<td>Logical AND of A and B</td>
</tr>
<tr>
<td>:LEOR:</td>
<td></td>
<td>A:LEOR:B</td>
<td>Logical Exclusive OR of A and B</td>
</tr>
<tr>
<td>:LOR:</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
3.7 Diagnostic messages

The assembler can give a range of additional diagnostic messages. By default, these diagnostic messages are not displayed. However, you can control what messages the assembler gives using command-line options. See Controlling the output of diagnostic messages on page 3-18 for details.

This section contains the following subsections:

- Interlocks
- IT block generation
- Thumb branch target alignment on page 3-46.

3.7.1 Interlocks

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

```
armasm --diag_warning 1563
```

**Note**

Where the --cpu option specifies a multi-issue processor such as Cortex-A8, the assembler warnings are unpredictable.

3.7.2 IT block generation

If you write:

```
AREA x, CODE
THUMB
MOVNE r0, r1 ; (1)
NOP
IT NE
MOVNE r0, r1 ; (2)
END
```

the assembler generates an IT instruction before the first MOVNE instruction.

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

```
armasm --diag_warning 1763
```
3.7.3 Thumb branch target alignment

On some processors, non word-aligned Thumb instructions sometimes take one or more additional cycles to execute in loops. This means that it can be an advantage to ensure that branch targets are word-aligned. The assembler can issue warnings when branch targets in Thumb code are not word-aligned. To do this, use the following command-line option when invoking the assembler:

```
armasm --diag_warning 1604
```
3.8 Using the C preprocessor

You can use C preprocessor commands in your assembly language source file. If you do this, you must use the --cpreproc command-line option when invoking the assembler. This causes armasm to call armcc to preprocess the file before assembling it.

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

armasm passes certain options to armcc if present on the command-line. These are shown in Table 3-16. Some of these options are converted to the armcc equivalent before passing to armcc. These are shown in Table 3-17.

<table>
<thead>
<tr>
<th>Table 3-16 Command-line options</th>
</tr>
</thead>
<tbody>
<tr>
<td>--16 --arm_only --diag_error --diag_warning --li</td>
</tr>
<tr>
<td>--32 --bi --diag_remark --fpu --library_type</td>
</tr>
<tr>
<td>--apcs --cpu --diag_style --fpumode --thumb</td>
</tr>
<tr>
<td>--arm --device --diag_suppress --i --unaligned_access --no_unaligned_access</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 3-17 armcc equivalent command-line options</th>
</tr>
</thead>
<tbody>
<tr>
<td>armasm</td>
</tr>
<tr>
<td>--16</td>
</tr>
<tr>
<td>--32</td>
</tr>
<tr>
<td>--i</td>
</tr>
</tbody>
</table>

To pass other simple compiler options, such as the preprocessor option `-D`, you must use the --cpreproc_opts command-line option. See the Compiler User Guide for more information. armasm correctly interprets the preprocessed `#line` commands. It can generate error messages and debug line tables using the information in the `#line` commands.

Example 3-1 on page 3-48 shows the commands you write to preprocess and assemble a file, `source.s`. The example also passes the compiler options to define a macro called RELEASE, and to undefine a macro called ALPHA.
Example 3-1 Preprocessing an assembly language source file

```
armasm --cpreproc --cpreproc_opts=-D,RELEASE,-U,ALPHA source.s
```

If you want to use complex preprocessor options, you must manually call armcc to preprocess the file before calling armasm. Example 3-2 shows the commands you write to manually preprocess and assemble a file, source.s. In this example, the preprocessor outputs a file called preprocessed.s, and armasm assembles preprocessed.s.

Example 3-2 Preprocessing an assembly language source file manually

```
armcc -E source.s > preprocessed.s
armasm preprocessed.s
```
Chapter 4
ARM and Thumb Instructions

This chapter describes the ARM®, Thumb® (all versions), and ThumbEE instructions supported by the ARM assembler. It contains the following sections:

- Instruction summary on page 4-2
- Memory access instructions on page 4-10
- General data processing instructions on page 4-41
- Multiply instructions on page 4-71
- Saturating instructions on page 4-93
- Parallel instructions on page 4-98
- Packing and unpacking instructions on page 4-106
- Branch and control instructions on page 4-114
- Coprocessor instructions on page 4-123
- Miscellaneous instructions on page 4-131
- Instruction width selection in Thumb on page 4-148
- ThumbEE instructions on page 4-150
- Pseudo-instructions on page 4-154.

Some instruction sections have an Architectures subsection. Instructions that do not have an Architecture subsection are available in all versions of the ARM instruction set, and all versions of the Thumb instruction set.
4.1 Instruction summary

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

--- Note ---
Unless stated otherwise, ThumbEE instructions are identical to Thumb instructions.

Table 4-1 Location of instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Brief description</th>
<th>Page</th>
<th>Arch. *</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC, ADD</td>
<td>Add with Carry, Add</td>
<td>page 4-45</td>
<td>All</td>
</tr>
<tr>
<td>ADR</td>
<td>Load program or register-relative address (short range)</td>
<td>page 4-23</td>
<td>All</td>
</tr>
<tr>
<td>ADRL pseudo-instruction</td>
<td>Load program or register-relative address (medium range)</td>
<td>page 4-155</td>
<td>x6M</td>
</tr>
<tr>
<td>AND</td>
<td>Logical AND</td>
<td>page 4-51</td>
<td>All</td>
</tr>
<tr>
<td>ASR</td>
<td>Arithmetic Shift Right</td>
<td>page 4-67</td>
<td>All</td>
</tr>
<tr>
<td>B</td>
<td>Branch</td>
<td>page 4-115</td>
<td>All</td>
</tr>
<tr>
<td>BFC, BFI</td>
<td>Bit Field Clear and Insert</td>
<td>page 4-107</td>
<td>T2</td>
</tr>
<tr>
<td>BIC</td>
<td>Bit Clear</td>
<td>page 4-51</td>
<td>All</td>
</tr>
<tr>
<td>BKP</td>
<td>Breakpoint</td>
<td>page 4-132</td>
<td>5</td>
</tr>
<tr>
<td>BL</td>
<td>Branch with Link</td>
<td>page 4-115</td>
<td>All</td>
</tr>
<tr>
<td>BLX</td>
<td>Branch with Link, change instruction set</td>
<td>page 4-115</td>
<td>T</td>
</tr>
<tr>
<td>BX</td>
<td>Branch, change instruction set</td>
<td>page 4-115</td>
<td>T</td>
</tr>
<tr>
<td>BXJ</td>
<td>Branch, change to Jazelle</td>
<td>page 4-115</td>
<td>J, x7M</td>
</tr>
<tr>
<td>CBZ, CBNZ</td>
<td>Compare and Branch if {Non}Zero</td>
<td>page 4-121</td>
<td>T2</td>
</tr>
<tr>
<td>CDP</td>
<td>Coprocessor Data Processing operation</td>
<td>page 4-124</td>
<td>x6M</td>
</tr>
<tr>
<td>CDP2</td>
<td>Coprocessor Data Processing operation</td>
<td>page 4-124</td>
<td>5, x6M</td>
</tr>
<tr>
<td>CHKA</td>
<td>Check array</td>
<td>page 4-152</td>
<td>EE</td>
</tr>
<tr>
<td>CLREX</td>
<td>Clear Exclusive</td>
<td>page 4-39</td>
<td>K, x6M</td>
</tr>
<tr>
<td>Mnemonic</td>
<td>Brief description</td>
<td>Page</td>
<td>Arch.</td>
</tr>
<tr>
<td>---------------</td>
<td>--------------------------------------------------------</td>
<td>------------</td>
<td>--------</td>
</tr>
<tr>
<td>CLZ</td>
<td>Count leading zeros</td>
<td>page 4-54</td>
<td>5, x6M</td>
</tr>
<tr>
<td>CMN, CMP</td>
<td>Compare Negative, Compare</td>
<td>page 4-55</td>
<td>All</td>
</tr>
<tr>
<td>CPS</td>
<td>Change Processor State</td>
<td>page 4-138</td>
<td>6</td>
</tr>
<tr>
<td>DBG</td>
<td>Debug</td>
<td>page 4-144</td>
<td>7</td>
</tr>
<tr>
<td>DMB, DSB</td>
<td>Data Memory Barrier, Data Synchronization Barrier</td>
<td>page 4-144</td>
<td>7, 6M</td>
</tr>
<tr>
<td>ENTERX, LEAVEX</td>
<td>Change state to or from ThumbEE</td>
<td>page 4-151</td>
<td>EE</td>
</tr>
<tr>
<td>EOR</td>
<td>Exclusive OR</td>
<td>page 4-51</td>
<td>All</td>
</tr>
<tr>
<td>HB, HBL, HBLP, HBP</td>
<td>Handler Branch, branches to a specified handler</td>
<td>page 4-153</td>
<td>EE</td>
</tr>
<tr>
<td>ISB</td>
<td>Instruction Synchronization Barrier</td>
<td>page 4-144</td>
<td>7, 6M</td>
</tr>
<tr>
<td>IT</td>
<td>If-Then</td>
<td>page 4-118</td>
<td>T2</td>
</tr>
<tr>
<td>LDC</td>
<td>Load Coprocessor</td>
<td>page 4-129</td>
<td>x6M</td>
</tr>
<tr>
<td>LDC2</td>
<td>Load Coprocessor</td>
<td>page 4-129</td>
<td>5, x6M</td>
</tr>
<tr>
<td>LDM</td>
<td>Load Multiple registers</td>
<td>page 4-27</td>
<td>All</td>
</tr>
<tr>
<td>LDR</td>
<td>Load Register with word</td>
<td>page 4-10</td>
<td>All</td>
</tr>
<tr>
<td>LDR pseudo-instruction</td>
<td>Load Register pseudo-instruction</td>
<td>page 4-159</td>
<td>All</td>
</tr>
<tr>
<td>LDRB</td>
<td>Load Register with byte</td>
<td>page 4-10</td>
<td>All</td>
</tr>
<tr>
<td>LDRST</td>
<td>Load Register with byte, user mode</td>
<td>page 4-10</td>
<td>x6M</td>
</tr>
<tr>
<td>LDRD</td>
<td>Load Registers with two words</td>
<td>page 4-10</td>
<td>5E, x6M</td>
</tr>
<tr>
<td>LDREX</td>
<td>Load Register Exclusive</td>
<td>page 4-36</td>
<td>6, x6M</td>
</tr>
<tr>
<td>LDREXB, LDREXH</td>
<td>Load Register Exclusive Byte, Halfword</td>
<td>page 4-36</td>
<td>K, x6M</td>
</tr>
<tr>
<td>LDREXD</td>
<td>Load Register Exclusive Doubleword</td>
<td>page 4-36</td>
<td>K, x7M</td>
</tr>
<tr>
<td>LDRH</td>
<td>Load Register with halfword</td>
<td>page 4-10</td>
<td>All</td>
</tr>
<tr>
<td>LDRHT</td>
<td>Load Register with halfword, user mode</td>
<td>page 4-10</td>
<td>T2</td>
</tr>
<tr>
<td>LDRSB</td>
<td>Load Register with signed byte</td>
<td>page 4-10</td>
<td>All</td>
</tr>
<tr>
<td>LDRSBT</td>
<td>Load Register with signed byte, user mode</td>
<td>page 4-10</td>
<td>T2</td>
</tr>
</tbody>
</table>
### Table 4-1 Location of instructions (continued)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Brief description</th>
<th>Page</th>
<th>Arch.</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDRSH</td>
<td>Load Register with signed halfword</td>
<td>page 4-10</td>
<td>All</td>
</tr>
<tr>
<td>LDRSHT</td>
<td>Load Register with signed halfword, user mode</td>
<td>page 4-10</td>
<td>T2</td>
</tr>
<tr>
<td>LDRT</td>
<td>Load Register with word, user mode</td>
<td>page 4-10</td>
<td>x6M</td>
</tr>
<tr>
<td>LSL, LSR</td>
<td>Logical Shift Left, Logical Shift Right</td>
<td>page 4-67</td>
<td>All</td>
</tr>
<tr>
<td>MAR</td>
<td>Move from Registers to 40-bit Accumulator</td>
<td>page 4-147</td>
<td>XScale</td>
</tr>
<tr>
<td>MCR</td>
<td>Move from Register to Coprocessor</td>
<td>page 4-125</td>
<td>x6M</td>
</tr>
<tr>
<td>MCR2</td>
<td>Move from Register to Coprocessor</td>
<td>page 4-125</td>
<td>5, x6M</td>
</tr>
<tr>
<td>MCRR</td>
<td>Move from Registers to Coprocessor</td>
<td>page 4-125</td>
<td>5E, x6M</td>
</tr>
<tr>
<td>MCR2</td>
<td>Move from Registers to Coprocessor</td>
<td>page 4-125</td>
<td>6, x6M</td>
</tr>
<tr>
<td>MIA, MIAx, MIAxy</td>
<td>Multiply with Internal 40-bit Accumulate</td>
<td>page 4-91</td>
<td>XScale</td>
</tr>
<tr>
<td>MLA</td>
<td>Multiply Accumulate</td>
<td>page 4-72</td>
<td>x6M</td>
</tr>
<tr>
<td>MLS</td>
<td>Multiply and Subtract</td>
<td>page 4-72</td>
<td>T2</td>
</tr>
<tr>
<td>MOV</td>
<td>Move</td>
<td>page 4-57</td>
<td>All</td>
</tr>
<tr>
<td>MOVT</td>
<td>Move Top</td>
<td>page 4-60</td>
<td>T2</td>
</tr>
<tr>
<td>MOV32</td>
<td>Move 32-bit constant to register</td>
<td>page 4-157</td>
<td>T2</td>
</tr>
<tr>
<td>MRA</td>
<td>Move from 40-bit Accumulator to Registers</td>
<td>page 4-147</td>
<td>XScale</td>
</tr>
<tr>
<td>MRC</td>
<td>Move from Coprocessor to Register</td>
<td>page 4-127</td>
<td>x6M</td>
</tr>
<tr>
<td>MRC2</td>
<td>Move from Coprocessor to Register</td>
<td>page 4-127</td>
<td>5, x6M</td>
</tr>
<tr>
<td>MRRC</td>
<td>Move from Coprocessor to Registers</td>
<td>page 4-127</td>
<td>5E, x6M</td>
</tr>
<tr>
<td>MRRC2</td>
<td>Move from Coprocessor to Registers</td>
<td>page 4-127</td>
<td>6, x6M</td>
</tr>
<tr>
<td>MRS</td>
<td>Move from PSR to register</td>
<td>page 4-134</td>
<td>All</td>
</tr>
<tr>
<td>MSR</td>
<td>Move from register to PSR</td>
<td>page 4-136</td>
<td>All</td>
</tr>
<tr>
<td>MUL</td>
<td>Multiply</td>
<td>page 4-72</td>
<td>All</td>
</tr>
<tr>
<td>MVN</td>
<td>Move Not</td>
<td>page 4-57</td>
<td>All</td>
</tr>
<tr>
<td>NOP</td>
<td>No Operation</td>
<td>page 4-142</td>
<td>All</td>
</tr>
<tr>
<td>Mnemonic</td>
<td>Brief description</td>
<td>Page</td>
<td>Arch.</td>
</tr>
<tr>
<td>-------------------</td>
<td>--------------------------------------------------</td>
<td>--------</td>
<td>-------</td>
</tr>
<tr>
<td>ORN</td>
<td>Logical OR NOT</td>
<td>4-51</td>
<td>T2</td>
</tr>
<tr>
<td>ORR</td>
<td>Logical OR</td>
<td>4-51</td>
<td>All</td>
</tr>
<tr>
<td>PKHBT, PKHTB</td>
<td>Pack Halfwords</td>
<td>4-112</td>
<td>6, x7M</td>
</tr>
<tr>
<td>PLD</td>
<td>Preload Data</td>
<td>4-25</td>
<td>5E, x6M</td>
</tr>
<tr>
<td>PLDW</td>
<td>Preload Data with intent to Write</td>
<td>4-25</td>
<td>7MP</td>
</tr>
<tr>
<td>PLI</td>
<td>Preload Instruction</td>
<td>4-25</td>
<td>7</td>
</tr>
<tr>
<td>PUSH, POP</td>
<td>PUSH registers to stack, POP registers from stack</td>
<td>4-30</td>
<td>All</td>
</tr>
<tr>
<td>QADD, QADD, QSUB, QSUB</td>
<td>Saturating Arithmetic</td>
<td>4-94</td>
<td>5E, x7M</td>
</tr>
<tr>
<td>QADD8, QADD16, QASX, QSUB8, QSUB16, QSAX</td>
<td>Parallel signed Saturating Arithmetic</td>
<td>4-99</td>
<td>6, x7M</td>
</tr>
<tr>
<td>RBIT</td>
<td>Reverse Bits</td>
<td>4-65</td>
<td>T2</td>
</tr>
<tr>
<td>REV, REV16, REVSH</td>
<td>Reverse byte order</td>
<td>4-65</td>
<td>6</td>
</tr>
<tr>
<td>RFE</td>
<td>Return From Exception</td>
<td>4-32</td>
<td>T2, x7M</td>
</tr>
<tr>
<td>ROR</td>
<td>Rotate Right Register</td>
<td>4-67</td>
<td>All</td>
</tr>
<tr>
<td>RRX</td>
<td>Rotate Right with Extend</td>
<td>4-67</td>
<td>x6M</td>
</tr>
<tr>
<td>RSB</td>
<td>Reverse Subtract</td>
<td>4-45</td>
<td>All</td>
</tr>
<tr>
<td>RSC</td>
<td>Reverse Subtract with Carry</td>
<td>4-45</td>
<td>x7M</td>
</tr>
<tr>
<td>SADD8, SADD16, SASX</td>
<td>Parallel signed arithmetic</td>
<td>4-99</td>
<td>6, x7M</td>
</tr>
<tr>
<td>SBC</td>
<td>Subtract with Carry</td>
<td>4-45</td>
<td>All</td>
</tr>
<tr>
<td>SBFX, UBFX</td>
<td>Signed, Unsigned Bit Field eXtract</td>
<td>4-108</td>
<td>T2</td>
</tr>
<tr>
<td>SDIV</td>
<td>Signed divide</td>
<td>4-70</td>
<td>7M, 7R</td>
</tr>
<tr>
<td>SEL</td>
<td>Select bytes according to APSR GE flags</td>
<td>4-63</td>
<td>6, x7M</td>
</tr>
<tr>
<td>SETEND</td>
<td>Set Endianness for memory accesses</td>
<td>4-141</td>
<td>6, x7M</td>
</tr>
<tr>
<td>SEV</td>
<td>Set Event</td>
<td>4-142</td>
<td>K, 6M</td>
</tr>
<tr>
<td>SHADD8, SHAADD16, SHASX, SHSUB8, SHSUB16, SHSAX</td>
<td>Parallel signed Halving arithmetic</td>
<td>4-99</td>
<td>6, x7M</td>
</tr>
</tbody>
</table>
### ARM and Thumb Instructions

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

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Brief description</th>
<th>Page</th>
<th>Arch.</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMC</td>
<td>Secure Monitor Call</td>
<td>page 4-140</td>
<td>Z</td>
</tr>
<tr>
<td>SMLAxy</td>
<td>Signed Multiply with Accumulate (32 &lt;= 16 x 16 + 32)</td>
<td>page 4-76</td>
<td>5E, x7M</td>
</tr>
<tr>
<td>SMLAD</td>
<td>Dual Signed Multiply Accumulate</td>
<td>page 4-86</td>
<td>6, x7M</td>
</tr>
<tr>
<td>SMLAL</td>
<td>Signed Multiply Accumulate (64 &lt;= 64 + 32 x 32)</td>
<td>page 4-74</td>
<td>x6M</td>
</tr>
<tr>
<td>SMLALxy</td>
<td>Signed Multiply Accumulate (64 &lt;= 64 + 16 x 16)</td>
<td>page 4-80</td>
<td>5E, x7M</td>
</tr>
<tr>
<td>SMLALD</td>
<td>Dual Signed Multiply Accumulate Long</td>
<td>page 4-88</td>
<td>6, x7M</td>
</tr>
<tr>
<td>SMLAwy</td>
<td>Signed Multiply with Accumulate (32 &lt;= 32 x 16 + 32)</td>
<td>page 4-78</td>
<td>5E, x7M</td>
</tr>
<tr>
<td>SMLSD</td>
<td>Dual Signed Multiply Subtract Accumulate</td>
<td>page 4-86</td>
<td>6, x7M</td>
</tr>
<tr>
<td>SMLSLD</td>
<td>Dual Signed Multiply Subtract Accumulate Long</td>
<td>page 4-88</td>
<td>6, x7M</td>
</tr>
<tr>
<td>SMMLA</td>
<td>Signed top word Multiply with Accumulate (32 &lt;= TopWord(32 x 32 + 32))</td>
<td>page 4-84</td>
<td>6, x7M</td>
</tr>
<tr>
<td>SMMLS</td>
<td>Signed top word Multiply with Subtract (32 &lt;= TopWord(32 - 32 x 32))</td>
<td>page 4-84</td>
<td>6, x7M</td>
</tr>
<tr>
<td>SMMUL</td>
<td>Signed top word Multiply (32 &lt;= TopWord(32 x 32))</td>
<td>page 4-84</td>
<td>6, x7M</td>
</tr>
<tr>
<td>SMUAD, SMUSD</td>
<td>Dual Signed Multiply, and Add or Subtract products</td>
<td>page 4-82</td>
<td>6, x7M</td>
</tr>
<tr>
<td>SMULxy</td>
<td>Signed Multiply (32 &lt;= 16 x 16)</td>
<td>page 4-76</td>
<td>5E, x7M</td>
</tr>
<tr>
<td>SMULL</td>
<td>Signed Multiply (64 &lt;= 32 x 32)</td>
<td>page 4-74</td>
<td>x6M</td>
</tr>
<tr>
<td>SMULwy</td>
<td>Signed Multiply (32 &lt;= 32 x 16)</td>
<td>page 4-78</td>
<td>5E, x7M</td>
</tr>
<tr>
<td>SRS</td>
<td>Store Return State</td>
<td>page 4-34</td>
<td>T2, x7M</td>
</tr>
<tr>
<td>SSAT</td>
<td>Signed Saturate</td>
<td>page 4-96</td>
<td>6, x6M</td>
</tr>
<tr>
<td>SSAT16</td>
<td>Signed Saturate, parallel halfwords</td>
<td>page 4-104</td>
<td>6, x7M</td>
</tr>
<tr>
<td>SSUB8, SSUB16, SSAX</td>
<td>Parallel signed arithmetic</td>
<td>page 4-99</td>
<td>6, x7M</td>
</tr>
<tr>
<td>Mnemonic</td>
<td>Brief description</td>
<td>Page</td>
<td>Arch.</td>
</tr>
<tr>
<td>-----------</td>
<td>------------------------------------------</td>
<td>--------</td>
<td>--------</td>
</tr>
<tr>
<td>STC</td>
<td>Store Coprocessor</td>
<td>4-129</td>
<td>x6M</td>
</tr>
<tr>
<td>STC2</td>
<td>Store Coprocessor</td>
<td>4-129</td>
<td>5, x6M</td>
</tr>
<tr>
<td>STM</td>
<td>Store Multiple registers</td>
<td>4-27</td>
<td>All</td>
</tr>
<tr>
<td>STR</td>
<td>Store Register with word</td>
<td>4-10</td>
<td>All</td>
</tr>
<tr>
<td>STR8</td>
<td>Store Register with byte</td>
<td>4-10</td>
<td>All</td>
</tr>
<tr>
<td>STR8T</td>
<td>Store Register with byte, user mode</td>
<td>4-10</td>
<td>x6M</td>
</tr>
<tr>
<td>STRD</td>
<td>Store Registers with two words</td>
<td>4-10</td>
<td>5E, x6M</td>
</tr>
<tr>
<td>STREX</td>
<td>Store Register Exclusive</td>
<td>4-36</td>
<td>6, x6M</td>
</tr>
<tr>
<td>STREXB, STREXH</td>
<td>Store Register Exclusive Byte, Halfword</td>
<td>4-36</td>
<td>K, x6M</td>
</tr>
<tr>
<td>STREXD</td>
<td>Store Register Exclusive Doubleword</td>
<td>4-36</td>
<td>K, x7M</td>
</tr>
<tr>
<td>STRH</td>
<td>Store Register with halfword</td>
<td>4-10</td>
<td>All</td>
</tr>
<tr>
<td>STRHT</td>
<td>Store Register with halfword, user mode</td>
<td>4-10</td>
<td>T2</td>
</tr>
<tr>
<td>STRT</td>
<td>Store Register with word, user mode</td>
<td>4-10</td>
<td>x6M</td>
</tr>
<tr>
<td>SUB</td>
<td>Subtract</td>
<td>4-45</td>
<td>All</td>
</tr>
<tr>
<td>SUBS pc, lr</td>
<td>Exception return, no stack</td>
<td>4-49</td>
<td>T2, x7M</td>
</tr>
<tr>
<td>SVC (formerly SWI)</td>
<td>SuperVisor Call</td>
<td>4-133</td>
<td>All</td>
</tr>
<tr>
<td>SWP, SWPB</td>
<td>Swap registers and memory (ARM only)</td>
<td>4-40</td>
<td>All, x7M</td>
</tr>
<tr>
<td>SXTAB, SXTAB16, SXTAH</td>
<td>Signed extend, with Addition</td>
<td>4-109</td>
<td>6, x7M</td>
</tr>
<tr>
<td>SXTB, SXTH</td>
<td>Signed extend</td>
<td>4-109</td>
<td>6</td>
</tr>
<tr>
<td>SXTB16</td>
<td>Signed extend</td>
<td>4-109</td>
<td>6, x7M</td>
</tr>
<tr>
<td>TBB, TBH</td>
<td>Table Branch Byte, Halfword</td>
<td>4-122</td>
<td>T2</td>
</tr>
<tr>
<td>TEQ</td>
<td>Test Equivalence</td>
<td>4-61</td>
<td>x6M</td>
</tr>
<tr>
<td>TST</td>
<td>Test</td>
<td>4-61</td>
<td>All</td>
</tr>
<tr>
<td>UADD8, UADD16, UASX</td>
<td>Parallel Unsigned Arithmetic</td>
<td>4-99</td>
<td>6, x7M</td>
</tr>
<tr>
<td>UDIV</td>
<td>Unsigned divide</td>
<td>4-70</td>
<td>7M, 7R</td>
</tr>
</tbody>
</table>
### ARM and Thumb Instructions

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

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Brief description</th>
<th>Page</th>
<th>Arch.</th>
</tr>
</thead>
<tbody>
<tr>
<td>UHADD8, UHADD16, UHASX, UHSUB8, UHSUB16, UHSAX</td>
<td>Parallel Unsigned Halving Arithmetic</td>
<td>page 4-99</td>
<td>6, x7M</td>
</tr>
<tr>
<td>UMAAL</td>
<td>Unsigned Multiply Accumulate Accumulate Long</td>
<td>page 4-90</td>
<td>6, x7M</td>
</tr>
<tr>
<td>UMLAL, UMULL</td>
<td>Unsigned Multiply Accumulate, Multiply</td>
<td>page 4-74</td>
<td>x6M</td>
</tr>
<tr>
<td>(64 &lt;= 32 x 32 x 32)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UQADD8, UQADD16, UQASX, UQSUB8, UQSUB16, UQAX</td>
<td>Parallel Unsigned Saturating Arithmetic</td>
<td>page 4-99</td>
<td>6, x7M</td>
</tr>
<tr>
<td>USAD8</td>
<td>Unsigned Sum of Absolute Differences</td>
<td>page 4-102</td>
<td>6, x7M</td>
</tr>
<tr>
<td>USA8</td>
<td>Accumulate Unsigned Sum of Absolute Differences</td>
<td>page 4-102</td>
<td>6, x7M</td>
</tr>
<tr>
<td>USAT</td>
<td>Unsigned Saturate</td>
<td>page 4-96</td>
<td>6, x6M</td>
</tr>
<tr>
<td>USAT16</td>
<td>Unsigned Saturate, parallel halfwords</td>
<td>page 4-104</td>
<td>6, x7M</td>
</tr>
<tr>
<td>USUB8, USUB16, USAX</td>
<td>Parallel unsigned arithmetic</td>
<td>page 4-99</td>
<td>6, x7M</td>
</tr>
<tr>
<td>UXTAB, UXTAB16, UXTAH</td>
<td>Unsigned extend with Addition</td>
<td>page 4-109</td>
<td>6, x7M</td>
</tr>
<tr>
<td>UXTB, UXTH</td>
<td>Unsigned extend</td>
<td>page 4-109</td>
<td>6</td>
</tr>
<tr>
<td>UXTB16</td>
<td>Unsigned extend</td>
<td>page 4-109</td>
<td>6, x7M</td>
</tr>
<tr>
<td>V*</td>
<td>See Chapter 5 NEON and VFP Programming</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WFE, WIFI, YIELD</td>
<td>Wait For Event, Wait For Interrupt, Yield</td>
<td>page 4-142</td>
<td>T2, 6M</td>
</tr>
</tbody>
</table>
a. Entries in the Architecture column have the following meanings:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>All</td>
<td>These instructions are available in all versions of the ARM architecture.</td>
</tr>
<tr>
<td>5</td>
<td>These instructions are available in the ARMv5T*, ARMv6*, and ARMv7 architectures.</td>
</tr>
<tr>
<td>5E</td>
<td>These instructions are available in the ARMv5TE, ARMv6*, and ARMv7 architectures.</td>
</tr>
<tr>
<td>6</td>
<td>These instructions are available in the ARMv6* and ARMv7 architectures.</td>
</tr>
<tr>
<td>6M</td>
<td>These instructions are available in the ARMv6-M and ARMv7 architectures.</td>
</tr>
<tr>
<td>x6M</td>
<td>These instructions are not available in the ARMv6-M profile.</td>
</tr>
<tr>
<td>7</td>
<td>These instructions are available in the ARMv7 architectures.</td>
</tr>
<tr>
<td>7M</td>
<td>These instructions are available in the ARMv7-M profile.</td>
</tr>
<tr>
<td>x7M</td>
<td>These instructions are not available in the ARMv6-M or ARMv7-M profile.</td>
</tr>
<tr>
<td>7R</td>
<td>These instructions are available in the ARMv7-R profile.</td>
</tr>
<tr>
<td>7MP</td>
<td>These instructions are available in the ARMv7 architectures that implement the Multiprocessing Extensions.</td>
</tr>
<tr>
<td>EE</td>
<td>These instructions are available in ThumbEE variants of the ARM architecture.</td>
</tr>
<tr>
<td>J</td>
<td>This instruction is available in the ARMv5TEJ, ARMv6*, and ARMv7 architectures.</td>
</tr>
<tr>
<td>K</td>
<td>These instructions are available in the ARMv6K, and ARMv7 architectures.</td>
</tr>
<tr>
<td>T</td>
<td>These instructions are available in ARMv4T, ARMv5T*, ARMv6*, and ARMv7 architectures.</td>
</tr>
<tr>
<td>T2</td>
<td>These instructions are available in the ARMv6T2 and above architectures.</td>
</tr>
<tr>
<td>XScale</td>
<td>These instructions are available in XScale versions of the ARM architecture.</td>
</tr>
<tr>
<td>Z</td>
<td>This instruction is available if Security Extensions are implemented.</td>
</tr>
</tbody>
</table>
4.2 Memory access instructions

This section contains the following subsections:

- **Address alignment** on page 4-11
  Alignment considerations that apply to all memory access instructions.

- **LDR and STR (immediate offset)** on page 4-12
  Load and Store with immediate offset, pre-indexed immediate offset, or post-indexed immediate offset.

- **LDR and STR (register offset)** on page 4-15
  Load and Store with register offset, pre-indexed register offset, or post-indexed register offset.

- **LDR and STR (User mode)** on page 4-18
  Load and Store, with User mode privilege.

- **LDR (pc-relative)** on page 4-20
  Load register. The address is an offset from the pc.

- **ADR** on page 4-23
  Load a program-relative or register-relative address.

- **PLD, PLDW, and PLI** 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 4-159). 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 ARMv6T2 and above unaligned access is permitted. In ARMv7 and above unaligned access is available (and is the default).

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 all your accesses are aligned, you can use the **--no_unaligned_access** command line option, to avoid linking in any library functions that might have an unaligned option.

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 (immediate offset)

Load and Store with immediate offset, pre-indexed immediate offset, or post-indexed immediate offset.

Syntax

\[
\begin{align*}
\text{op}\{\text{type}\}\{\text{cond}\} & \ R_t, \ [R_n \{, \ #\text{offset}\}] \quad \text{; immediate offset} \\
\text{op}\{\text{type}\}\{\text{cond}\} & \ R_t, \ [R_n, \ #\text{offset}]! \quad \text{; pre-indexed} \\
\text{op}\{\text{type}\}\{\text{cond}\} & \ R_t, \ [R_n], \ #\text{offset} \quad \text{; post-indexed} \\
\text{op}\{\text{cond}\} & \ R_t, \ R_{t2}, \ [R_n \{, \ #\text{offset}\}] \quad \text{; immediate offset, doubleword} \\
\text{op}\{\text{cond}\} & \ R_t, \ R_{t2}, \ [R_n, \ #\text{offset}]! \quad \text{; pre-indexed, doubleword} \\
\text{op}\{\text{cond}\} & \ R_t, \ R_{t2}, \ [R_n], \ #\text{offset} \quad \text{; post-indexed, doubleword}
\end{align*}
\]

where:

\[
\begin{align*}
\text{op} & \quad \text{can be either:} \\
\text{LDR} & \quad \text{Load Register} \\
\text{STR} & \quad \text{Store Register.}
\end{align*}
\]

\[
\begin{align*}
\text{type} & \quad \text{can be any one of:} \\
\text{B} & \quad \text{unsigned Byte (Zero extend to 32 bits on loads.)} \\
\text{SB} & \quad \text{signed Byte (LDR only. Sign extend to 32 bits.)} \\
\text{H} & \quad \text{unsigned Halfword (Zero extend to 32 bits on loads.)} \\
\text{SH} & \quad \text{signed Halfword (LDR only. Sign extend to 32 bits.)} \\
- & \quad \text{omitted, for Word.}
\end{align*}
\]

\[
\begin{align*}
\text{cond} & \quad \text{is an optional condition code (see Conditional execution on page 2-18).}
\end{align*}
\]

\[
\begin{align*}
R_t & \quad \text{is the register to load or store.}
\end{align*}
\]

\[
\begin{align*}
R_n & \quad \text{is the register on which the memory address is based.}
\end{align*}
\]

\[
\begin{align*}
\text{offset} & \quad \text{is an offset. If offset is omitted, the address is the contents of } R_n.
\end{align*}
\]

\[
\begin{align*}
R_{t2} & \quad \text{is the additional register to load or store for doubleword operations.}
\end{align*}
\]

Not all options are available in every instruction set and architecture. See Offset ranges and architectures on page 4-13 for details.
Offset ranges and architectures

Table 4-2 shows the ranges of offsets and availability of these instructions.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Immediate offset</th>
<th>Pre-indexed</th>
<th>Post-indexed</th>
<th>Arch.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARM, word or byte a</td>
<td>–4095 to 4095</td>
<td>–4095 to 4095</td>
<td>–4095 to 4095</td>
<td>All</td>
</tr>
<tr>
<td>ARM, signed byte, halfword, or signed halfword</td>
<td>–255 to 255</td>
<td>–255 to 255</td>
<td>–255 to 255</td>
<td>All</td>
</tr>
<tr>
<td>ARM, doubleword</td>
<td>–255 to 255</td>
<td>–255 to 255</td>
<td>–255 to 255</td>
<td>v5TE +</td>
</tr>
<tr>
<td>32-bit Thumb, word, halfword, signed halfword, byte, or signed byte a</td>
<td>–255 to 4095</td>
<td>–255 to 255</td>
<td>–255 to 255</td>
<td>v6T2, v7</td>
</tr>
<tr>
<td>32-bit Thumb, doubleword</td>
<td>–1020 to 1020 c</td>
<td>–1020 to 1020 c</td>
<td>–1020 to 1020 c</td>
<td>v6T2, v7</td>
</tr>
<tr>
<td>16-bit Thumb, word b</td>
<td>0 to 124 c</td>
<td>Not available</td>
<td>Not available</td>
<td>All T</td>
</tr>
<tr>
<td>16-bit Thumb, unsigned halfword b</td>
<td>0 to 62 d</td>
<td>Not available</td>
<td>Not available</td>
<td>All T</td>
</tr>
<tr>
<td>16-bit Thumb, unsigned byte b</td>
<td>0 to 31</td>
<td>Not available</td>
<td>Not available</td>
<td>All T</td>
</tr>
<tr>
<td>16-bit Thumb, word, Rn is r13 c</td>
<td>0 to 1020 c</td>
<td>Not available</td>
<td>Not available</td>
<td>All T</td>
</tr>
<tr>
<td>16-bit ThumbEE, word b</td>
<td>–28 to 124 c</td>
<td>Not available</td>
<td>Not available</td>
<td>T-2EE</td>
</tr>
<tr>
<td>16-bit ThumbEE, word, Rn is r9 e</td>
<td>0 to 252 c</td>
<td>Not available</td>
<td>Not available</td>
<td>T-2EE</td>
</tr>
<tr>
<td>16-bit ThumbEE, word, Rn is r10 e</td>
<td>0 to 124 c</td>
<td>Not available</td>
<td>Not available</td>
<td>T-2EE</td>
</tr>
</tbody>
</table>

a. For word loads, Rt can be the pc. A load to the pc causes a branch to the address loaded. In ARMv4, bits[1:0] of the address loaded must be 0b00. In ARMv5T and above, bits[1:0] must not be 0b10, and if bit[0] is 1, execution continues in Thumb state, otherwise execution continues in ARM state.
b. Rt and Rn must be in the range r0-r7.
c. Must be divisible by 4.
d. Must be divisible by 2.
e. Rt must be in the range r0-r7.

Register restrictions

Rn must be different from Rt in the pre-index and post-index forms.
Doubleword register restrictions

Rn must be different from Rt2 in the pre-index and post-index forms.

For Thumb-2 instructions, you must not specify sp or pc for either Rt or Rt2.

For ARM instructions:
- Rt must be an even-numbered register
- Rt must not be lr
- it is strongly recommended that you do not use r12 for Rt
- Rt2 must be R(t + 1).

Use of PC

In ARM instructions:
- You can use PC for Rt in LDR word instructions and PC for Rn in LDR instructions.
- You can use PC for Rt in STR word instructions and PC for Rn in STR instructions with immediate offset syntax (that is the forms that do not writeback to the Rn).
- However, these are deprecated.

Other uses of PC are not allowed in these ARM instructions.

In Thumb instructions you can use PC for Rt in LDR word instructions and PC for Rn in LDR instructions. Other uses of PC in these Thumb instructions are not allowed.

Use of SP

You can use SP for Rn.

In ARM, you can use SP for Rt in word instructions. Uses of SP for Rt in non-word instructions are deprecated in ARM code.

In Thumb, you can use SP for Rt in word instructions only. All other use of SP for Rt in these instructions are unpredictable in Thumb code.

Examples

LDR     r8,[r10]            ; loads r8 from the address in r10.
LDRNE   r2,[r5,#960]!       ; (conditionally) loads r2 from a word
                        ; 960 bytes above the address in r5, and
                        ; increments r5 by 960.
STR     r2,[r9,#consta-struc] ; consta-struc is an expression evaluating
                        ; to a constant in the range 0-4095.
4.2.3 LDR and STR (register offset)

Load and Store with register offset, pre-indexed register offset, or post-indexed register offset.

Syntax

\[
\text{op}\{\text{cond}\} \ Rt, [\ Rn, +/-Rm \{, \ shift\}] \ ; \ \text{register offset}
\]

\[
\text{op}\{\text{cond}\} \ Rt, [\ Rn, +/-Rm \{, \ shift\}]! \ ; \ \text{pre-indexed}
\]

\[
\text{op}\{\text{cond}\} \ Rt, [\ Rn], +/-Rm \{, \ shift\} \ ; \ \text{post-indexed}
\]

\[
\text{opD}{\text{cond}} \ Rt, Rt2, [\ Rn, +/-Rm \{, \ shift\}] \ ; \ \text{register offset, doubleword}
\]

\[
\text{opD}{\text{cond}} \ Rt, Rt2, [\ Rn, +/-Rm \{, \ shift\}]! \ ; \ \text{pre-indexed, doubleword}
\]

\[
\text{opD}{\text{cond}} \ Rt, Rt2, [\ Rn], +/-Rm \{, \ shift\} \ ; \ \text{post-indexed, doubleword}
\]

where:

- \text{op} can be either:
  - \text{LDR} Load Register
  - \text{STR} Store Register.

- \text{type} can be any one of:
  - \text{B} unsigned Byte (Zero extend to 32 bits on loads.)
  - \text{SB} signed Byte (LDR only. Sign extend to 32 bits.)
  - \text{H} unsigned Halfword (Zero extend to 32 bits on loads.)
  - \text{SH} signed Halfword (LDR only. Sign extend to 32 bits.)
  - \text{-} omitted, for Word.

- \text{cond} is an optional condition code (see \textit{Conditional execution} on page 2-18).

- \text{Rt} is the register to load or store.

- \text{Rn} is the register on which the memory address is based.

- \text{Rm} is a register containing a value to be used as the offset. \text{Rm} must not be r15.

- \text{–Rm} is not allowed in Thumb code.

- \text{shift} is an optional shift.

- \text{Rt2} is the additional register to load or store for doubleword operations.

Not all options are available in every instruction set and architecture. See Offset register and shift options on page 4-16 for details.
Offset register and shift options

Table 4-3 shows the ranges of offsets and availability of these instructions.

Table 4-3 Options and architectures, LDR/STR (register offsets)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>+/-Rm</th>
<th>shift</th>
<th>Arch.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARM, word or byte b</td>
<td>+/-Rm</td>
<td>LSL #0-31</td>
<td>All</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LSR #1-32</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ASR #1-32</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ROR #1-31</td>
<td>RXX</td>
</tr>
<tr>
<td>ARM, signed byte, halfword, or signed halfword</td>
<td>+/-Rm</td>
<td>Not available</td>
<td>All</td>
</tr>
<tr>
<td>ARM, double word</td>
<td>+/-Rm</td>
<td>Not available</td>
<td>v5TE+</td>
</tr>
<tr>
<td>32-bit Thumb, word, halfword, signed halfword,</td>
<td>+/-Rm</td>
<td>LSL #0-3</td>
<td>v6T2, v7</td>
</tr>
<tr>
<td>byte, or signed byte b</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32-bit Thumb, doubleword</td>
<td>+/-Rm</td>
<td>Not available</td>
<td>v6T2, v7</td>
</tr>
<tr>
<td>16-bit Thumb, all except doubleword c</td>
<td>+/-Rm</td>
<td>Not available</td>
<td>All T</td>
</tr>
<tr>
<td>16-bit ThumbEE, word b</td>
<td>+/-Rm</td>
<td>LSL #2 (required)</td>
<td>T-2EE</td>
</tr>
<tr>
<td>16-bit ThumbEE, halfword, signed halfword b</td>
<td>+/-Rm</td>
<td>LSL #1 (required)</td>
<td>T-2EE</td>
</tr>
<tr>
<td>16-bit ThumbEE, byte, signed byte b</td>
<td>+/-Rm</td>
<td>Not available</td>
<td>T-2EE</td>
</tr>
</tbody>
</table>

a. Where +/-Rm is shown, you can use –Rm, +Rm, or Rm. Where +Rm is shown, you cannot use –Rm.
b. For word loads, Rt can be the pc. A load to the pc causes a branch to the address loaded. In ARMv4, bits[1:0] of the address loaded must be 0b00. In ARMv5T and above, bits[1:0] must not be 0b10, and if bit[0] is 1, execution continues in Thumb state, otherwise execution continues in ARM state.
c. Rt, Rn, and Rm must all be in the range r0-r7.

Register restrictions

In the pre-index and post-index forms:
- Rn must be different from Rt
- Rn must be different from Rm in architectures before ARMv6.

Doubleword register restrictions

For Thumb-2 instructions, you must not specify sp or pc for either Rt or Rt2.

For ARM instructions:
- Rt must be an even-numbered register
- Rt must not be lr
it is strongly recommended that you do not use r12 for \( R_t \)
- \( R_t 2 \) must be \( R(t + 1) \)
- \( R_n \) must be different from \( R_t \) and \( R_t 2 \) in LDRD instructions
- \( R_n \) must be different from \( R_t 2 \) in the pre-index and post-index forms.

**Use of PC**

In ARM instructions:
- You can use PC for \( R_t \) in LDR word instructions and PC for \( R_n \) in LDR instructions with register offset syntax (that is the forms that do not writeback to the \( R_n \)).
- You can use PC for \( R_t \) in STR word instructions and PC for \( R_n \) in STR instructions with register offset syntax (that is the forms that do not writeback to the \( R_n \)). However, these are deprecated.

Other uses of PC are not allowed in ARM instructions.

In Thumb instructions you can use PC for \( R_t \) in LDR word instructions. Other uses of PC in these Thumb instructions are not allowed.

**Use of SP**

You can use SP for \( R_n \).

In ARM, you can use SP for \( R_t \) in word instructions. Uses of SP for \( R_t \) in non-word instructions are deprecated in ARM code.

In Thumb, you can use SP for \( R_t \) in word instructions only. All other use of SP for \( R_t \) in these instructions are unpredictable in Thumb code.

Use of SP for \( R_n \) is unpredictable in Thumb state and deprecated in ARM state.
4.2.4 LDR and STR (User mode)

Load and Store, byte, halfword, or word, with User mode privilege.

When these instructions are executed in a privileged mode, they access memory with the same restrictions as they would have if they were executed in User mode.

In User mode, these instructions behave in exactly the same way as normal memory accesses.

Syntax

\[
\begin{align*}
\text{op}(\text{type})T(\text{cond}) \ Rt, [\ Rn \ {, \ #\ offset}] & \quad \text{; immediate offset (Thumb-2 only)} \\
\text{op}(\text{type})T(\text{cond}) \ Rt, [\ Rn \ {, \ #\ offset}] & \quad \text{; post-indexed (ARM only)} \\
\text{op}(\text{type})T(\text{cond}) \ Rt, [\ Rn], +/-\ Rm \ {, \ shift} & \quad \text{; post-indexed (register) (ARM only)}
\end{align*}
\]

where:

- \( \text{op} \) can be either:
  - LDR Load Register
  - STR Store Register.

- \( \text{type} \) can be any one of:
  - \( B \) unsigned Byte (Zero extend to 32 bits on loads.)
  - \( SB \) signed Byte (LDR only. Sign extend to 32 bits.)
  - \( H \) unsigned Halfword (Zero extend to 32 bits on loads.)
  - \( SH \) signed Halfword (LDR only. Sign extend to 32 bits.)
  - \( - \) omitted, for Word.

- \( \text{cond} \) is an optional condition code (see Conditional execution on page 2-18).

- \( \text{Rt} \) is the register to load or store.

- \( \text{Rn} \) is the register on which the memory address is based.

- \( \text{offset} \) is an offset. If offset is omitted, the address is the value in \( \text{Rn} \).

- \( \text{Rm} \) is a register containing a value to be used as the offset. \( \text{Rm} \) must not be \( \text{r15} \).

- \( \text{shift} \) is an optional shift.
## Offset ranges and architectures

Table 4-2 on page 4-13 shows the ranges of offsets and availability of these instructions.

### Table 4-4 Offsets and architectures, LDR/STR (User mode)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Immediate offset</th>
<th>Post-indexed</th>
<th>+/– Rm a</th>
<th>shift</th>
<th>Arch.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARM, word or byte</td>
<td>Not available</td>
<td>–4095 to 4095</td>
<td>+/- Rm</td>
<td>LSL #0–31</td>
<td>All</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>LSR #1–32</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ASR #1–32</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ROR #1–31</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>RRX</td>
<td></td>
</tr>
<tr>
<td>ARM, signed byte, halfword, or signed halfword</td>
<td>Not available</td>
<td>–255 to 255</td>
<td>+/- Rm</td>
<td>Not available</td>
<td>v6T2, v7</td>
</tr>
<tr>
<td>32-bit Thumb, word, halfword, signed halfword,</td>
<td>0 to 255</td>
<td>Not available</td>
<td>Not available</td>
<td>v6T2, v7</td>
<td></td>
</tr>
<tr>
<td>byte, or signed byte</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

a. You can use –Rm, +Rm, or Rm.
4.2.5  LDR (pc-relative)

Load register. The address is an offset from the pc.

--- Note ---
See also *Pseudo-instructions* on page 4-154.

--- Syntax ---

LDR{type}{cond}{.W} Rt, label

LDRD{cond} Rt, Rt2, label ; Doubleword

where:

type can be any one of:
- B  unsigned Byte (Zero extend to 32 bits on loads.)
- SB signed Byte (LDR only. Sign extend to 32 bits.)
- H  unsigned Halfword (Zero extend to 32 bits on loads.)
- SH signed Halfword (LDR only. Sign extend to 32 bits.)
- omitted, for Word.

cond is an optional condition code (see *Conditional execution* on page 2-18).

.W is an optional instruction width specifier. See *LDR (pc-relative) in Thumb-2* on page 4-21 for details.

Rt is the register to load or store.

Rt2 is the second register to load or store.

label is a program-relative expression. See *Register-relative and program-relative expressions* on page 3-37 for more information.

label must be within a limited distance of the current instruction. See *Offset range and architectures* on page 4-21 for details.

--- Note ---
Equivalent syntaxes are available for the STR instruction in ARM code but they are deprecated.
Offset range and architectures

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

Table 4-5 shows the possible offsets between label and the current instruction.

### Table 4-5 pc-relative offsets

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Offset range</th>
<th>Architectures</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARM LDR, LDRB, LDRSB, LDRH, LDRSH a</td>
<td>+/- 4095</td>
<td>All</td>
</tr>
<tr>
<td>ARM LDRD</td>
<td>+/- 255</td>
<td>v5TE +</td>
</tr>
<tr>
<td>32-bit Thumb LDR, LDRB, LDRSB, LDRH, LDRSH a</td>
<td>+/- 4095</td>
<td>v6T2, v7</td>
</tr>
<tr>
<td>32-bit Thumb LDRD</td>
<td>+/- 1020 b</td>
<td>v6T2, v7</td>
</tr>
<tr>
<td>16-bit Thumb LDR c</td>
<td>0-1020 b</td>
<td>All T</td>
</tr>
</tbody>
</table>

a. For word loads, Rt can be the pc. A load to the pc causes a branch to the address loaded. In ARMv4, bits[1:0] of the address loaded must be 0b00. In ARMv5T and above, bits[1:0] must not be 0b10, and if bit[0] is 1, execution continues in Thumb state, otherwise execution continues in ARM state.
b. Must be a multiple of 4.
c. Rt must be in the range r0-r7. There are no byte, halfword, or doubleword 16-bit instructions.

---

**Note**

In ARMv7-M, LDRD (PC-relative) instructions must be on a word-aligned address.

---

**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 sp or pc for either Rt or Rt2.
For ARM instructions:
- \( R_t \) must be an even-numbered register
- \( R_t \) must not be \( lr \)
- it is strongly recommended that you do not use \( r12 \) for \( R_t \)
- \( R_t2 \) must be \( R(t + 1) \).

**Use of SP**

In ARM, you can use SP for \( R_t \) in LDR word instructions. Uses of SP for \( R_t \) in LDR non-word instructions are deprecated in ARM code.

In Thumb, you can use SP for \( R_t \) in LDR word instructions only. All other uses of SP in these instructions are unpredictable in Thumb code.
4.2.6 ADR

ADR adds an immediate value to the pc value, and writes the result to the destination register.

**Syntax**

ADR{cond}{.W} Rd, label

where:

- **cond** is an optional condition code (see Conditional execution on page 2-18).
- **.W** is an optional instruction width specifier. See ADR in Thumb-2 on page 4-24 for details.
- **Rd** is the register to load.
- **label** is a program-relative expression. See Register-relative and program-relative expressions on page 3-37 for more information. *label* must be within a limited distance of the current instruction. See Offset range and architectures on page 4-24 for details.

**Usage**

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 (see ADRL pseudo-instruction on page 4-155).

If *label* is program-relative, it must evaluate to an address in the same assembler area as the ADR instruction, see AREA on page 7-70.

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

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

Table 4-5 on page 4-21 shows the possible offsets between label and the current instruction.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Offset range</th>
<th>Architectures</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARM ADR</td>
<td>See Constants in Operand2 on page 4-43</td>
<td>All</td>
</tr>
<tr>
<td>32-bit Thumb ADR</td>
<td>+/- 4095</td>
<td>v6T2, v7</td>
</tr>
<tr>
<td>16-bit Thumb ADR a</td>
<td>0-1020 b</td>
<td>All T</td>
</tr>
</tbody>
</table>

a. Rd must be in the range r0-r7.

b. Must be a multiple of 4.

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 with .W always generates a 32-bit instruction, even if the address can be generated in a 16-bit instruction.

For forward references, ADR without .W always generates a 16-bit instruction in Thumb code, even if that results in failure for an address that could be generated in a 32-bit Thumb-2 ADR instruction.
4.2.7 PLD, PLDW, and PLI

Preload Data and Preload Instruction. The processor can signal the memory system that a data or instruction load from an address is likely in the near future.

Syntax

\[ \text{PLtype}\{\text{cond}\} \ [Rn \ [\text{#offset}] \] \]

\[ \text{PLtype}\{\text{cond}\} \ [Rn, +/-Rm \ [\text{shift}] \] \]

\[ \text{PLtype}\{\text{cond}\} \ label \]

where:

type can be one of:

D Data address

DW Data address with intention to write

I Instruction address.

type cannot be DW if the syntax specifies label.

cond is an optional condition code (see Conditional execution on page 2-18).

\[ \text{Note} \]

cond is allowed only in Thumb-2 code, using a preceding IT instruction. This is an unconditional instruction in ARM and you must not use cond.

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

offset is an immediate offset. If offset is omitted, the address is the value in Rn.

Rm is a register containing a value to be used as the offset. Rm must not be r15. For Thumb instructions Rm must also not be r13.

shift is an optional shift.

label is a program-relative expression. See Register-relative and program-relative expressions on page 3-37 for more information.

Range of 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, when Rn is not r15.
• −4095 to +4095 for Thumb-2 instructions, when $Rn$ is r15.

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

**Register or shifted register offset**

In ARM, 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

**Address alignment for preloads**

No alignment checking is performed for preload instructions.

**Architectures**

ARM PLD is available in ARMv5TE and above.

32-bit Thumb PLD is available in ARMv6T2 and above.

PLDW is available only in ARMv7 and above that implement the Multiprocessing Extensions.

PLI is available only in ARMv7 and above.

There are no 16-bit Thumb PLD, PLDW, or PLI instructions.

These are hint instructions, and their implementation is optional. If they are not implemented, they execute as NOPs.
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

\[
\text{op}\{\text{addr\_mode}\}\{\text{cond}\}\ Rn\{!\}, \ \text{reglist}\{^\wedge\}
\]

where:

- **op** can be either:
  - LDM Load Multiple registers
  - STM Store Multiple registers.

- **addr\_mode** is any one of the following:
  - IA Increment address After each transfer. This is the default, and can be omitted.
  - IB Increment address Before each transfer (ARM only).
  - DA Decrement address After each transfer (ARM only).
  - DB Decrement address Before each transfer.

See Table 2-9 on page 2-41 for stack oriented addressing mode suffixes.

- **cond** is an optional condition code (see Conditional execution on page 2-18).

- **Rn** is the base register, the ARM register holding the initial address for the transfer. \( Rn \) must not be r15.

- **!** is an optional suffix. If ! is present, the final address is written back into \( Rn \).

- **reglist** is a list of one or more registers to be loaded 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 Examples on page 4-29).

See Restrictions on reglist in 32-bit Thumb-2 instructions 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:
  - If the instruction is \( \text{LDM} \) (with any addressing mode) and \( \text{reglist} \) contains the pc (r15), in addition to the normal multiple register transfer, the SPSR is copied into the CPSR. This is for returning from exception handlers. Use this only from exception modes.
Otherwise, data is transferred into or out of the User mode registers instead of the current mode registers.

Restrictions on reglist in 32-bit Thumb-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
- there must be two or more registers in the list.

If you write an `STM` or `LDM` instruction with only one register in reglist, the assembler automatically substitutes the equivalent `STR` or `LDR` instruction. Be aware of this when comparing disassembly listings with source code.

You can use the `--diag_warning 1645` assembler command-line option to check when an instruction substitution occurs.

Restrictions on reglist in ARM instructions

ARM store instructions can have `SP` and `PC` in the `reglist` but these instructions that include `SP` or `PC` in the `reglist` are deprecated.

ARM load instructions can have `SP` and `PC` in the `reglist` but these instructions that include `SP` in the `reglist` or both `PC` and `LR` in the `reglist` are deprecated.

16-bit instructions

16-bit versions of a subset of these instructions are available in Thumb-2 code, and in Thumb code on pre-Thumb-2 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 for `STM` instructions
- writeback must be specified for `LDM` instructions where `Rn` is not in the `reglist`.

--- Note ---
16-bit Thumb `STM` instructions with writeback that specify `Rn` as the lowest register in the `reglist` are deprecated.
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.

Note

These 16-bit instructions are not available in Thumb-2EE.

Loading to the pc

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

In ARMv4, bits[1:0] of the address loaded must be 0b00.

In ARMv5T and above:
- bits[1:0] must not be 0b10
- if bit[0] is 1, execution continues in Thumb state
- if bit[0] is 0, execution continues in ARM state.

Loading or storing the base register, with writeback

In ARM code or pre-Thumb-2 Thumb code, 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 cannot be relied upon.

In Thumb-2 code, if Rn is in reglist, and writeback is specified with the ! suffix:
- all 32-bit instructions are unpredictable
- 16-bit instructions behave in the same way as in pre-Thumb-2 Thumb code, but the use of these instructions is deprecated.

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, and pop registers off a full descending stack.

Syntax

PUSH{cond} reglist

POP{cond} reglist

where:

cond is an optional condition code (see Conditional execution on page 2-18).

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

Usage

PUSH and POP are synonyms for STMDB and LDM (or LDMIA), with the base register sp (r13), 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 the pc. This is usually a return from a subroutine, where the lr was pushed onto the stack at the start of the subroutine.

In ARMv5T and above:
- bits[1:0] must not be 0b10
- if bit[0] is 1, execution continues in Thumb state
- if bit[0] is 0, execution continues in ARM state.

In ARMv4, bits[1:0] of the address loaded must be 0b00. POP cannot be used to change state.
Thumb instructions

A subset of these instructions are available in the Thumb instruction set.

The following restrictions apply to the 16-bit instructions:
- For `PUSH`, `reglist` can only include the Lo registers and the `lr`
- For `POP`, `reglist` can only include the Lo registers and the `pc`.

The following restrictions apply to the 32-bit instructions:
- `reglist` must not include the `sp`
- For `PUSH`, `reglist` must not include the `pc`
- For `POP`, `reglist` can include either the `lr` or the `pc`, but not both.

Restrictions on reglist in ARM instructions

ARM `PUSH` instructions can have `SP` and `PC` in the `reglist` but these instructions that include `SP` or `PC` in the `reglist` are deprecated.

ARM `POP` instructions can have `SP` and `PC` in the `reglist` but these instructions that include `SP` in the `reglist` or both `PC` and `LR` in the `reglist` are deprecated.

Examples

```
PUSH    {r0,r4-r7}
PUSH    {r2,lr}
POP     {r0,r10,pc} ; no 16-bit version available
```
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.

If *addr_mode* is omitted, it defaults to Increment After.

*cond* is an optional condition code (see *Conditional execution* on page 2-18).

---

**Note**

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

---

*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). *Rn* is usually the *sp* where the return state information was saved.

**Operation**

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

RFE writes an address to the pc. The alignment of this address must be correct for the instruction set in use after the exception return:

- For a return to ARM, the address written to the pc must be word-aligned.
- For a return to Thumb-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. 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.

Do not use RFE in unprivileged modes, see Processor mode on page 2-5.

Do not use RFE in Thumb2-EE.

Architectures

This ARM instruction is available in ARMv6 and above.

This 32-bit Thumb instruction is available in ARMv6T2 and above, except the ARMv7-M profile.

There is no 16-bit version of this instruction.

Example

RFE sp!
4.2.11 SRS

Store Return State onto a stack.

Syntax

SRS{addr_mode}{cond} sp{!}, #modenum
SRS{addr_mode}{cond} #modenum[!] ; This is a pre-UAL syntax

where:

addr_mode

is any one of the following:

IA  Increment address After each transfer
IB  Increment address Before each transfer (ARM only)
DA  Decrement address After each transfer (ARM only)
DB  Decrement address Before each transfer (Full Descending stack).

If addr_mode is omitted, it defaults to Increment After. See Table 2-9 on page 2-41 for stack oriented addressing mode suffixes.

cond

is an optional condition code (see Conditional execution on page 2-18).

---------- Note ----------

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

----------

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

modenum

specifies the number of the mode whose banked sp is used as the base register, see Processor mode on page 2-5. You must use only the defined mode numbers.

Operation

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

---------- Note ----------

For full descending stack, you must use SRSFD or SRSDB.
Usage

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

Notes

Where addresses are not word-aligned, SRS ignores the least significant two bits of the specified address.

The time order of the accesses to individual words of memory generated by SRS is not architecturally defined. Do not use this instruction on memory-mapped I/O locations where access order matters.

Do not use SRS in User and System modes because these modes do not have a SPSR.

Do not use SRS in Thumb-2EE.

SRS is unpredictable in a non-secure state if modenum specifies monitor mode, see Processor mode on page 2-5.

Architectures

This ARM instruction is available in ARMv6 and above.

This 32-bit Thumb instruction is available in ARMv6T2 and above, except the ARMv7-M profile.

There is no 16-bit version of this instruction.

Example

R13_usr EQU 16
SRSFD sp,#R13_usr
4.2.12 LDREX and STREX

Load and Store Register Exclusive.

Syntax

LDREX{cond} Rt, [Rn {, #offset}]
STREX{cond} Rd, Rt, [Rn {, #offset}]
LDREXB{cond} Rt, [Rn]
STREXB{cond} Rd, Rt, [Rn]
LDREXH{cond} Rt, [Rn]
STREXH{cond} Rd, Rt, [Rn]
LDREXD{cond} Rt, Rt2, [Rn]
STREXD{cond} Rd, Rt, Rt2, [Rn]

where:

cond is an optional condition code (see Conditional execution on page 2-18).
Rd is the destination register for the returned status.
Rt is the register to load or store.
Rt2 is the second register for doubleword loads or stores.
Rn is the register on which the memory address is based.
offset is an optional offset applied to the value in Rn. offset is allowed only in Thumb-2 instructions. If offset 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, the tag is cleared, and the value 0 is returned in Rd.
- If the physical address does not have the Shared TLB attribute, and the executing processor does not have an outstanding tagged physical address, the store does not take place, and the value 1 is returned in Rd.
- If the physical address has the Shared TLB attribute, and the physical address is tagged as exclusive access for the executing processor, the store takes place, the tag is cleared, and the value 0 is returned in Rd.
- If the physical address has the Shared TLB attribute, and the physical address is not tagged as exclusive access for the executing processor, the store does not take place, and the value 1 is returned in Rd.

Restrictions

r15 must not be used for any of Rd, Rt, Rt2, or Rn.

For STREX, Rd must not be the same register as Rt, Rt2, or Rn.

For ARM instructions:
- For LDREXD and STREXD, Rt must be an even numbered register, and not r14
- Rt2 must be R(t+1)
- offset is not allowed.

For Thumb instructions:
- r13 must not be used for any of Rd, Rt, or Rt2
- for LDREXD, Rt and Rt2 must not be the same register
- the value of offset can be any multiple of four in the range 0-1020.
Usage

Use LDREX and STREX to implement interprocess communication in multiple-processor and shared-memory systems.

For reasons of performance, keep the number of instructions between corresponding LDREX and STREX 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.

Architectures

ARM LDREX and STREX are available in ARMv6 and above.

ARM LDREXB, LDREXH, LDREXD, STREXB, STREXD, and STREXH are available in ARMv6K and above.

All these 32-bit Thumb instructions are available in ARMv6T2 and above, except that LDREXD and STREXD are not available in the ARMv7-M profile.

There are no 16-bit versions of these instructions.

Examples

```
; load the 'lock taken' value
MOV r1, #0x1

; load the lock value
try
  LDREX r0, [LockAddr]

; is the lock free?
  CMP r0, #0

; try and claim the lock
  STREXEQ r0, r1, [LockAddr]

; did this succeed?
  CMP EQ r0, #0

; no - try again
  BNE try

; 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 2-18).

Note

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

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 ARMv6K and above.

This 32-bit Thumb-2 instruction is available in ARMv7 and above.

There is no 16-bit Thumb CLREX instruction.
4.2.14 SWP and SWPB

Swap data between registers and memory.

**Syntax**

\[ \text{SWP}\{B}\{\text{cond}\} \ Rt, \ Rt2, [Rn] \]

where:

- \( \text{cond} \) is an optional condition code (see *Conditional execution* on page 2-18).
- \( \text{B} \) is an optional suffix. If \( \text{B} \) is present, a byte is swapped. Otherwise, a 32-bit word is swapped.
- \( \text{Rt} \) is the destination register.
- \( \text{Rt2} \) is the source register. \( \text{Rt2} \) can be the same register as \( \text{Rt} \).
- \( \text{Rn} \) contains the address in memory. \( \text{Rn} \) must be a different register from both \( \text{Rt} \) and \( \text{Rt2} \).

**Usage**

You can use \text{SWP} and \text{SWPB} to implement semaphores:

- Data from memory is loaded into \( \text{Rt} \).
- The contents of \( \text{Rt2} \) is saved to memory.
- If \( \text{Rt2} \) is the same register as \( \text{Rt} \), the contents of the register is swapped with the contents of the memory location.

**Note**

The use of \text{SWP} and \text{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.

**Architectures**

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

There are no Thumb \text{SWP} or \text{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 on page 4-49
  Return from exception without popping the stack.
- AND, ORR, EOR, BIC, and ORN on page 4-51
  Logical AND, OR, Exclusive OR, OR NOT, and Bit Clear.
- 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 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 APSR 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.
- SDIV and UDIV on page 4-70
  Signed Divide and Unsigned Divide.
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:

1. 
   `
   \#constant
   `

2. 
   `
   Rm\{, \text{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, \textit{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, \textit{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, \textit{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).
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).

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.

You can use the --diag_warning 1645 assembler command-line option to check when an instruction substitution occurs.
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 4-99.

Syntax

\[ \text{op}\{S\}\{\text{cond}\} \{\text{Rd}\}, \text{Rn}, \text{Operand2} \]

\[ \text{op}\{\text{cond}\} \{\text{Rd}\}, \text{Rn}, \#\text{imm12} \quad ; \text{Thumb-2 ADD and SUB only} \]

where:

- \( \text{op} \) 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).

- \( S \) 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 2-18).

- \( \text{cond} \) is an optional condition code (see Conditional execution on page 2-18).

- \( \text{Rd} \) is the destination register.

- \( \text{Rn} \) is the register holding the first operand.

- \( \text{Operand2} \) is a flexible second operand. See Flexible second operand on page 4-42 for details of the option.

- \( \#\text{imm12} \) is any value in the range 0-4095.

Usage

The ADD instruction adds the values in \( \text{Rn} \) and \( \text{Operand2} \).

The SUB instruction subtracts the value of \( \text{Operand2} \) from the value in \( \text{Rn} \).

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

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 \( \text{Operand2} \), together with the carry flag.

The SBC (Subtract with Carry) instruction subtracts the value of \( \text{Operand2} \) 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 \( \text{Operand2} \). 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.

Use of pc in Thumb-2 instructions

In most of these instructions, you cannot use pc (r15) for \( Rd \), or any operand.

The exception is that you can use pc for \( Rn \) in ADD and SUB instructions, with a constant \( \text{Operand2} \) 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 on page 4-49.

See also ADR on page 4-23.

Use of pc in ARM instructions

If you use pc (r15) as \( Rn \), the value used is the address of the instruction plus 8.

If you use pc as \( Rd \):

- Execution branches to the address corresponding to the result.
- If you use the S suffix, the SPSR of the current mode is copied to the CPSR. You can use this to return from exceptions (see Chapter 6 Handling Processor Exceptions in the Developer Guide).

See also ADR on page 4-23.

--- Caution ---

Do not use the S suffix when using pc as \( Rd \) in User mode or System mode. The effect of such an instruction is unpredictable, but the assembler cannot warn you at assembly time.

You cannot use pc 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 pre-Thumb-2 Thumb code, and are 16-bit instructions when used in Thumb-2 code:

- `ADDS Rd, Rn, #imm`  
  `imm` range 0-7. RD and Rn must both be Lo registers.

- `ADDS Rd, Rn, Rm`  
  Rd, Rn and Rm must all be Lo registers.

- `ADD Rd, Rd, Rn`  
  ARMv6 and earlier: either Rd or Rn, or both, must be a Hi register. ARMv6T2 and above: this restriction does not apply.

- `ADDS Rd, Rd, #imm`  
  `imm` range 0-255. Rd must be a Lo register.

- `ADCS Rd, Rd, Rm`  
  Rd, Rn and Rm must all be Lo registers.

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

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

- `SUBS Rd, Rn, Rm`  
  Rd, Rn and Rm must all be Lo registers.

- `SUBS Rd, Rn, #imm`  
  `imm` range 0-7. Rd and Rn both Lo registers.

- `SUBS Rd, Rd, #imm`  
  `imm` range 0-255. Rd must be a Lo register.

- `SBCS Rd, Rd, Rm`  
  Rd, Rn and Rm must all be Lo registers.

- `SUB SP, SP, #imm`  
  `imm` range 0-508, word aligned.

- `RSBS Rd, Rn, #0`  
  Rd and Rn both Lo registers.
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
RSCSLE  r0,r5,r0,LSL r4 ; conditional, flags set

Incorrect example

RSCSLE  r0,pc,r0,LSL r4 ; pc 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

Exception return, without stack popping.

--- Note ---

This is a special case instruction in Thumb-2. The same instruction is available in ARM code as a normal form of the SUB instruction described in ADD, SUB, RSB, ADC, SBC, and RSC on page 4-45.

Syntax

SUBS{cond} pc, lr, #imm

where:

imm is an immediate constant. In Thumb-2 code, it is limited to the range 0-255. In ARM code, it is a flexible second operand. See Flexible second operand on page 4-42 for details.

cond is an optional condition code (see Conditional execution on page 2-18).

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

This 32-bit Thumb instruction is available in ARMv6T2 and above, except the ARMv7-M profile.

There is no 16-bit Thumb version of this instruction.
4.3.4 AND, ORR, EOR, BIC, and ORN

Logical AND, OR, Exclusive OR, Bit Clear, and OR NOT.

Syntax

\( \text{op}\{S\}\{\text{cond}\} \ Rd, \ Rn, \ \text{Operand2} \)

where:

- \( \text{op} \) is one of:
  - AND logical AND.
  - ORR logical OR.
  - EOR logical Exclusive OR.
  - BIC logical AND NOT.
  - ORN logical OR NOT (Thumb-2 only).

- \( S \) 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 2-18).

- \( \text{cond} \) is an optional condition code (see Conditional execution on page 2-18).

- \( \text{Rd} \) is the destination register.

- \( \text{Rn} \) is the register holding the first operand.

- \( \text{Operand2} \) is a flexible second operand. See Flexible second operand 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 \( \text{Operand2} \).

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 \( \text{Operand2} \).

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 \( \text{Operand2} \).

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 pc in Thumb-2 instructions

You cannot use pc (r15) for \( rd \) or any operand in any of these instructions.

Use of pc in ARM instructions

Note

All these uses of pc in these ARM instructions are deprecated.

If you use pc as \( Rn \), the value used is the address of the instruction plus 8.

If you use pc as \( Rd \):

- Execution branches to the address corresponding to the result.
- If you use the \( S \) suffix, the SPSR of the current mode is copied to the CPSR. You can use this to return from exceptions (see Chapter 6 Handling Processor Exceptions in the Developer Guide).

Caution

Do not use the \( S \) suffix when using pc as \( Rd \) in User mode or System mode. The effect of such an instruction is unpredictable, but the assembler cannot warn you at assembly time.

You cannot use pc for 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 pre-Thumb-2 Thumb code, and are 16-bit instructions when used in Thumb-2 code:

\[
\begin{align*}
\text{ANDS } & \text{ Rd, Rd, Rm } \quad \text{Rd and Rm must both be Lo registers.} \\
\text{EORS } & \text{ Rd, Rd, Rm } \quad \text{Rd and Rm must both be Lo registers.} \\
\text{ORRS } & \text{ Rd, Rd, Rm } \quad \text{Rd and Rm must both be Lo registers.} \\
\text{BICS } & \text{ Rd, Rd, Rm } \quad \text{Rd and Rm must both be Lo registers.}
\end{align*}
\]

In the first three cases, it does not matter if you specify \(\text{OPS } \text{Rd, Rm, Rd.} \) The instruction is the same.

Examples

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>r9, r2, #0xFF00</td>
</tr>
<tr>
<td>ORREQ</td>
<td>r2, r0, r5</td>
</tr>
<tr>
<td>EORS</td>
<td>r0, r0, r3, ROR r6</td>
</tr>
<tr>
<td>ANDS</td>
<td>r9, r8, #0x19</td>
</tr>
<tr>
<td>EORS</td>
<td>r7, r11, #0x18181818</td>
</tr>
<tr>
<td>BIC</td>
<td>r0, r1, #0xab</td>
</tr>
<tr>
<td>ORN</td>
<td>r7, r11, r14, ROR #4</td>
</tr>
<tr>
<td>ORNS</td>
<td>r7, r11, r14, ASR #32</td>
</tr>
</tbody>
</table>

Incorrect example

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EORS</td>
<td>r0, pc, r3, ROR r6 ; pc not permitted with register</td>
</tr>
<tr>
<td></td>
<td>; controlled shift</td>
</tr>
</tbody>
</table>

ARM and Thumb Instructions
4.3.5 CLZ

Count Leading Zeros.

Syntax

\[ \text{CLZ}\{\text{cond}\} \ Rd, \ Rm \]

where:
\( \text{cond} \) is an optional condition code (see Conditional execution on page 2-18).
\( \text{Rd} \) is the destination register. \( \text{Rd} \) must not be r15.
\( \text{Rm} \) is the operand register. \( \text{Rm} \) must not be r15.

Usage

The CLZ instruction counts the number of leading zeros in the value in \( \text{Rm} \) and returns the result in \( \text{Rd} \). 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 ARMv5T and above.

This 32-bit Thumb instruction is available in ARMv6T2 and above.

There is no 16-bit Thumb version of this instruction.

Examples

\[
\begin{align*}
\text{CLZ} & \quad r4, r9 \\
\text{CLZNE} & \quad r2, r3 \\
\end{align*}
\]

Use the CLZ Thumb-2 instruction followed by a left shift of \( \text{Rm} \) by the resulting \( \text{Rd} \) value to normalize the value of register \( \text{Rm} \). Use MOV5, rather than MOV, to flag the case where \( \text{Rm} \) is zero:

\[
\begin{align*}
\text{CLZ} & \quad r5, r9 \\
\text{MOV5} & \quad r9, r9, \text{LSL} \ r5 \\
\end{align*}
\]
4.3.6 CMP and CMN

Compare and Compare Negative.

Syntax

\[
\text{CMP} \{\text{cond}\} \ Rn, \ \text{Operand2} \\
\text{CMN} \{\text{cond}\} \ Rn, \ \text{Operand2}
\]

where:

- \textit{cond} is an optional condition code (see \textit{Conditional execution} on page 2-18).
- \textit{Rn} is the ARM register holding the first operand.
- \textit{Operand2} is a flexible second operand. See \textit{Flexible second operand} on page 4-42 for details of the options.

Usage

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

The \textit{CMP} instruction subtracts the value of \textit{Operand2} from the value in \textit{Rn}. This is the same as a \textit{SUBS} instruction, except that the result is discarded.

The \textit{CMN} instruction adds the value of \textit{Operand2} to the value in \textit{Rn}. This is the same as an \textit{ADDS} instruction, except that the result is discarded.

In certain circumstances, the assembler can substitute \textit{CMN} for \textit{CMP}, or \textit{CMP} for \textit{CMN}. Be aware of this when reading disassembly listings. See \textit{Instruction substitution} on page 4-44 for details.

Use of pc in ARM instructions

\textbf{Note}

Use of pc (r15) in these ARM instructions is deprecated.

If you use pc as \textit{Rn}, the value used is the address of the instruction plus 8.

You cannot use pc for any operand in any data processing instruction that has a register-controlled shift (see \textit{Flexible second operand} on page 4-42).
Use of pc in Thumb-2 instructions

You cannot use pc (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 pre-Thumb-2 Thumb code, and are 16-bit instructions when used in Thumb-2 code:

- **CMP** \( Rn, Rm \)
- **CMN** \( Rn, Rm \)  \( Rn \) and \( Rm \) must both be Lo registers.
- **CMP** \( Rn, \#imm \)  \( Rn \) must be a Lo register. \( imm \) range 0-255.

Examples

- **CMP**     \( r2, r9 \)
- **CMN**     \( r0, \#6400 \)
- **CMPT**    \( r13, r7, LSL \#2 \)

Incorrect example

- **CMP**     \( r2, pc, ASR r0 \)  ; pc not permitted with register controlled shift
4.3.7  MOV and MVN

Move and Move Not.

Syntax

\[
\begin{align*}
\text{MOV}\{S\}\{\text{cond}\} & \quad R_d, \text{Operand2} \\
\text{MOV}\{\text{cond}\} & \quad R_d, \#\text{imm16} \\
\text{MVN}\{S\}\{\text{cond}\} & \quad R_d, \text{Operand2}
\end{align*}
\]

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 2-18).
- \(\text{cond}\) is an optional condition code (see Conditional execution on page 2-18).
- \(R_d\) is the destination register.
- \(\text{Operand2}\) is a flexible second operand. See Flexible second operand on page 4-42 for details of the options.
- \(\text{imm16}\) is any value in the range 0-65535.

Usage

The MOV instruction copies the value of \(\text{Operand2}\) into \(R_d\).

The MVN instruction takes the value of \(\text{Operand2}\), performs a bitwise logical NOT operation on the value, and places the result into \(R_d\).

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.

Use of pc in Thumb-2 MOV and MVN

You cannot use pc (r15) for \(R_d\), or in \(\text{Operand2}\), in the Thumb-2 MOV or MVN instructions.

Use of pc in ARM MOV and MVN

--- Note ---

MOV \(R_d, R_n\) syntax is allowed with \(R_d\) or \(R_n = \text{pc}\), but not both. All other cases are deprecated.
If you use pc as Rd, the value used is the address of the instruction plus 8.

If you use pc as Rd:

- Execution branches to the address corresponding to the result.
- If you use the S suffix, the SPSR of the current mode is copied to the CPSR. You can use this to return from exceptions (see Chapter 6 Handling Processor Exceptions in the Developer Guide).

--- Caution ---

Do not use the S suffix when using pc as Rd in User mode or System mode. The effect of such an instruction is unpredictable, but the assembler cannot warn you at assembly time.

---

You cannot use pc 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 pre-Thumb-2 Thumb code, and are 16-bit instructions when used in Thumb-2 code:

- **MOV S Rd, #imm**
  - Rd must be a Lo register.
  - imm range 0-255.

- **MOV S Rd, Rm**
  - Rd and Rm must both be Lo registers.

- **MOV Rd, Rm**
  - In architectures before ARMv6, either Rd or Rm, or both, must be a Hi register. In ARMv6 and above, this restriction does not apply.
Architectures

The \texttt{#imm16} form of the ARM instruction is available in ARMv6T2 and above. The other forms of the ARM instruction are available in all versions of the ARM architecture.

These 32-bit Thumb instructions are available in ARMv6T2 and above.

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

Example

\begin{verbatim}
MVNNE r11, #0xF000000B ; ARM only. This constant is not available in T2.
\end{verbatim}

Incorrect example

\begin{verbatim}
MVN pc, r3, ASR r0 ; pc not permitted with register controlled shift
\end{verbatim}
4.3.8 MOVT

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 2-18).
Rd is the destination register. Rd cannot be the pc.
immed_16 is a 16-bit immediate constant.

Usage

MOVT 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, MOVT instruction pair.
See also MOV32 pseudo-instruction on page 4-157.

Condition flags

This instruction does not change the flags.

Architectures

This ARM instruction is available in ARMv6T2 and above.
This 32-bit Thumb instruction is available in ARMv6T2 and above.
There is no 16-bit Thumb version of this instruction.
4.3.9 TST and TEQ

Test bits and Test Equivalence.

Syntax

\[
\text{TST}\{\text{cond}\} \ Rn, \ \text{Operand2} \\
\text{TEQ}\{\text{cond}\} \ Rn, \ \text{Operand2}
\]

where:

\[
\text{cond} \quad \text{is an optional condition code (see Conditional execution on page 2-18).}\\
\text{Rn} \quad \text{is the ARM register holding the first operand.}\\
\text{Operand2} \quad \text{is a flexible second operand. See Flexible second operand on page 4-42 for details of the options.}
\]

Usage

These instructions test the value in a register against \text{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 \text{Rn} and the value of \text{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 \text{Rn} and the value of \text{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 pc

For ARM instructions:

- if you use pc (r15) as \text{Rn}, the value used is the address of the instruction plus 8
- you cannot use pc for any operand in any data processing instruction that has a register-controlled shift (see Flexible second operand on page 4-42).

For Thumb-2 instructions, you cannot use pc for \text{Rn} or in \text{Operand2}.
**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 pre-Thumb-2 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 architectures with ARM.

The `TST` Thumb instruction is available in all architectures with Thumb.

The `TEQ` Thumb instruction is available in ARMv6T2 and above.

**Examples**

- `TST` \( r0, \#0x3F8 \)
- `TEQ` \( r10, r9 \)
- `TSTNE` \( r1, r5, ASR r1 \)

**Incorrect example**

- `TEQ` \( pc, r1, ROR r0 \) ; pc not permitted with register
  \( ; \) controlled shift
4.3.10 SEL

Select bytes from each operand according to the state of the APSR GE flags.

**Syntax**

SEL{cond} {Rd}, Rn, Rm

where:
- **cond** is an optional condition code (see *Conditional execution* on page 2-18).
- **Rd** is the destination register.
- **Rn** is the register holding the first operand.
- **Rm** is the register holding the second operand.

**Operation**

The SEL instruction selects bytes from Rn or Rm according to the APSR GE flags:

- if GE[0] is set, Rd[7:0] come from Rn[7:0], otherwise from Rm[7:0]
- if GE[1] is set, Rd[15:8] come from Rn[15:8], otherwise from Rm[15:8]
- if GE[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 4-99. 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.

These 32-bit Thumb instructions are available in ARMv6T2 and above, except the ARMv7-M profile.

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
4.3.11 REV, REV16, REVSH, and RBIT

Reverse bytes or bits within words or halfwords.

**Syntax**

\[ op\{cond\} \ Rd, \ Rn \]

where:

*op* 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.

*cond* is an optional condition code (see *Conditional execution* on page 2-18).

*Rd* is the destination register. *Rd* must not be r15.

*Rn* is the register holding the operand. *Rn* 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:
  - 16-bit signed big-endian data into 32-bit signed little-endian data
  - 16-bit signed little-endian data into 32-bit signed big-endian data.

**Condition flags**

These instructions do not change the flags.
16-bit instructions

The following forms of these instructions are available in pre-Thumb-2 Thumb code, and are 16-bit instructions when used in Thumb-2 code:

\[ \text{REV } Rd, Rm \quad \text{Rd and Rm must both be Lo registers.} \]
\[ \text{REV16 } Rd, Rm \quad \text{Rd and Rm must both be Lo registers.} \]
\[ \text{REVSH } Rd, Rm \quad \text{Rd and Rm must both be Lo registers.} \]

Architectures

Other than RBIT, these ARM instructions are available in ARMv6 and above.

The RBIT ARM instruction is available in ARMv6T2 and above.

These 32-bit Thumb instructions are available in ARMv6T2 and above.

These 16-bit Thumb instructions are available in all T variants of ARMv6 and above.

Examples

\[ \text{REV } r3, r7 \]
\[ \text{REV16 } r0, r0 \]
\[ \text{REVSH } r0, r5 \quad ; \text{Reverse Signed Halfword} \]
\[ \text{REVHS } r3, r7 \quad ; \text{Reverse with Higher or Same condition} \]
\[ \text{RBIT } r7, r8 \]
4.3.12 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 2-18).
- \( Rd \) 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.

**Restrictions**

ARM instructions that use Rs must not use r15. Thumb instructions must not use r15 or r13.

**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 pre-Thumb-2 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.

- `ASRS Rd, Rd, Rs`  
  Rd and Rs must both be Lo registers.

- `LSLS Rd, Rm, #sh`  
  Rd and Rm must both be Lo registers.

- `LSLS Rd, Rd, Rs`  
  Rd and Rs must both be Lo registers.

- `LSRS Rd, Rm, #sh`  
  Rd and Rm must both be Lo registers.

- `LSRS Rd, Rd, Rs`  
  Rd and Rs must both be Lo registers.

- `RORS Rd, Rd, Rs`  
  Rd and Rs must both be Lo registers.

**Architectures**

The ASR, LSL, LSR, and ROR ARM instructions are available in all architectures.

The RRX ARM instruction is available in all architectures with ARM.

The ASR, LSL, LSR, and ROR Thumb instructions are available in all architectures with Thumb. These instructions must use the flag setting suffix S in architectures before ARMv6T2.
The **RRX** Thumb instruction is available in ARMv6T2 and above.

**Examples**

- **ASR**  r7, r8, r9
- **LSLS** r1, r2, r3
- **LSR** r4, r5, r6
- **ROR** r4, r5, r6
4.3.13 SDIV and UDIV

Signed and Unsigned Divide.

Syntax

SDIV{cond} {Rd}, Rn, Rm
UDIV{cond} {Rd}, Rn, Rm

where:

cond is an optional condition code (see Conditional execution on page 2-18).
Rd is the destination register.
Rn is the register holding the value to be divided.
Rm is a register holding the divisor.

Register restrictions

pc or sp cannot be used for Rd, Rn or Rm.

Architectures

These 32-bit Thumb instructions are available in ARMv7-R and ARMv7-M only.

There are no ARM or 16-bit Thumb SDIV and UDIV instructions.
4.4 Multiply instructions

This section contains the following subsections:

- **MUL, MLA, and MLS** on page 4-72
  Multiply, Multiply Accumulate, and Multiply Subtract (32-bit by 32-bit, bottom 32-bit result).

- **UMULL, UMLAL, SMULL, and SMLAL** on page 4-74
  Unsigned and signed Long Multiply and Multiply Accumulate (32-bit by 32-bit, 64-bit result or 64-bit accumulator).

- **SMUL\text{xy} and SMLA\text{xy}** on page 4-76
  Signed Multiply and Signed Multiply Accumulate (16-bit by 16-bit, 32-bit result).

- **SMUL\text{Wy} and SMLAW\text{Wy}** on page 4-78
  Signed Multiply and Signed Multiply Accumulate (32-bit by 16-bit, top 32-bit result).

- **SMLAL\text{xy}** on page 4-80
  Signed Multiply Accumulate (16-bit by 16-bit, 64-bit accumulate).

- **SMUAD\{X\} and SMUSD\{X\}** on page 4-82
  Dual 16-bit Signed Multiply with Addition or Subtraction of products.

- **SMMUL, SMMLA, and SMMLS** on page 4-84
  Multiply, Multiply Accumulate, and Multiply Subtract (32-bit by 32-bit, top 32-bit result).

- **SMLAD and SMLSD** on page 4-86
  Dual 16-bit Signed Multiply, 32-bit Accumulation of Sum or Difference of 32-bit products.

- **SMLALD and SMLSLD** on page 4-88
  Dual 16-bit Signed Multiply, 64-bit Accumulation of Sum or Difference of 32-bit products.

- **UMAAL** on page 4-90
  Unsigned Multiply Accumulate Accumulate Long.

- **MIA, MIAPH, and MIA\text{xy}** on page 4-91
  Multiplies with Internal Accumulate (XScale coprocessor 0 instructions).
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}, Rn, Rm
MLA{S}{cond} Rd, Rn, Rm, Ra
MLS{cond} Rd, Rn, Rm, Ra

where:

cond is an optional condition code (see Conditional execution on page 2-18).
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 2-18).
Rd is the destination register.
Rn, Rm are registers holding the values to be multiplied.
Ra is a register holding the value to be added or subtracted from.

Usage

The MUL instruction multiplies the values from Rn and Rm, and places the least significant 32 bits of the result in Rd.

The MLA instruction multiplies the values from Rn and Rm, adds the value from Ra, and places the least significant 32 bits of the result in Rd.

The MLS instruction multiplies the values from Rn and Rm, subtracts the result from the value from Ra, and places the least significant 32 bits of the final result in Rd.

Do not use r15 for Rd, Rn, Rm, or Ra.
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 ARMv5T and above.

Thumb instructions

The following form of the MUL instruction is available in pre-Thumb-2 Thumb code, and is a 16-bit instruction when used in Thumb-2 code:

MULS Rd, Rn, Rd

Rd and Rn must both be Lo registers.

There are no other Thumb multiply instructions that can update the condition code flags.

Architectures

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

The MLS ARM instruction is available in ARMv6T2 and above.

These 32-bit Thumb instructions are available in ARMv6T2 and above.

The MULS 16-bit 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
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

\[
\text{Op}\{S\}\{\text{cond}\} \ RdLo, \ RdHi, \ Rn, \ Rm
\]

where:

- \text{Op} is one of UMULL, UMLAL, SMULL, or SMLAL.
- \text{S} is an optional suffix available in ARM state only. If \text{S} is specified, the condition code flags are updated on the result of the operation (see Conditional execution on page 2-18).
- \text{cond} is an optional condition code (see Conditional execution on page 2-18).
- \text{RdLo}, \ \text{RdHi} are the destination registers. For UMLAL and SMLAL they also hold the accumulating value. \text{RdLo} and \text{RdHi} must be different registers.
- \text{Rn}, \ \text{Rm} are ARM registers holding the operands.

Do not use \text{r15} for \text{RdHi}, \text{RdLo}, \text{Rn}, or \text{Rm}.

Usage

The UMULL instruction interprets the values from \text{Rn} and \text{Rm} as unsigned integers. It multiplies these integers and places the least significant 32 bits of the result in \text{RdLo}, and the most significant 32 bits of the result in \text{RdHi}.

The UMLAL instruction interprets the values from \text{Rn} and \text{Rm} as unsigned integers. It multiplies these integers, and adds the 64-bit result to the 64-bit unsigned integer contained in \text{RdHi} and \text{RdLo}.

The SMULL instruction interprets the values from \text{Rn} and \text{Rm} as two’s complement signed integers. It multiplies these integers and places the least significant 32 bits of the result in \text{RdLo}, and the most significant 32 bits of the result in \text{RdHi}.

The SMLAL instruction interprets the values from \text{Rn} and \text{Rm} as two’s complement signed integers. It multiplies these integers, and adds the 64-bit result to the 64-bit signed integer contained in \text{RdHi} and \text{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 versions of the ARM architecture.

These 32-bit Thumb instructions are available in ARMv6T2 and above.

There are no 16-bit Thumb versions of these instructions.

Examples

  UMULL       r0, r4, r5, r6
  UMLALS      r4, r5, r3, r8
4.4.3 SMUL<xy> and SMLA<xy>

Signed Multiply and Multiply Accumulate, with 16-bit operands and a 32-bit result and accumulator.

Syntax

SMUL<xy>{cond} {Rd}, Rn, Rm
SMLA<xy>{cond} Rd, Rn, Rm, Ra

where:

<xy> is either B or T. B means use the bottom half (bits [15:0]) of Rn, T means use the top half (bits [31:16]) of Rn.

<xy> is either B or T. B means use the bottom half (bits [15:0]) of Rm, T means use the top half (bits [31:16]) of Rm.

cond is an optional condition code (see Conditional execution on page 2-18).

Rd is the destination register.

Rn, Rm are the registers holding the values to be multiplied.

Ra is the register holding the value to be added.

Usage

Do not use r15 for Rd, Rn, Rm, or Ra.

SMUL<xy> multiplies the 16-bit signed integers from the selected halves of Rn and Rm, and places the 32-bit result in Rd.

SMLA<xy> multiplies the 16-bit signed integers from the selected halves of Rn and Rm, adds the 32-bit result to the 32-bit value in Ra, and places the result in Rd.

Condition flags

These instructions do not affect the N, Z, C, or V flags.

If overflow occurs in the accumulation, SMLA<xy> sets the Q flag. To read the state of the Q flag, use an MRS instruction (see MRS on page 4-134).

Note

SMLA<xy> never clears the Q flag. To clear the Q flag, use an MSR instruction (see MSR on page 4-136).
Architectures

These ARM instructions are available in ARMv6 and above, and E variants of ARMv5T.

These 32-bit Thumb instructions are available in ARMv6T2 and above, except the ARMv7-M profile.

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
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

\[
\text{SMULW}<y>\{\text{cond}\} \{Rd\}, \ Rn, \ Rm \\
\text{SMLAW}<y>\{\text{cond}\} \ Rd, \ Rn, \ Rm, \ Ra
\]

where:

\(<y>\) is either B or T. B means use the bottom half (bits [15:0]) of \(Rm\), T means use the top half (bits [31:16]) of \(Rm\).

\(\text{cond}\) is an optional condition code (see Conditional execution on page 2-18).

\(Rd\) is the destination register.

\(Rn, \ Rm\) are the registers holding the values to be multiplied.

\(Ra\) is the register holding the value to be added.

Usage

Do not use r15 for \(Rd, \ Rn, \ Rm, \) or \(Ra\).

SMULWy multiplies the signed integer from the selected half of \(Rm\) by the signed integer from \(Rn\), and places the upper 32-bits of the 48-bit result in \(Rd\).

SMLAWy multiplies the signed integer from the selected half of \(Rm\) by the signed integer from \(Rn\), adds the 32-bit result to the 32-bit value in \(Ra\), 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, SMLAWy sets the Q flag (see MRS on page 4-134).

Architectures

These ARM instructions are available in ARMv6 and above, and E variants of ARMv5T.

These 32-bit Thumb instructions are available in ARMv6T2 and above, except the ARMv7-M profile.
There are no 16-bit Thumb versions of these instructions.
4.4.5 SMLAL<xy>

Signed Multiply-Accumulate with 16-bit operands and a 64-bit accumulator.

**Syntax**

\[
\text{SMLAL}<x><y><\text{cond} \ \ RdLo, \ RdHi, \ Rn, \ Rm}
\]

where:

\(<x>\) is either B or T. B means use the bottom half (bits [15:0]) of \(Rn\), T means use the top half (bits [31:16]) of \(Rn\).

\(<y>\) is either B or T. B means use the bottom half (bits [15:0]) of \(Rm\), T means use the top half (bits [31:16]) of \(Rm\).

\(\text{cond}\) is an optional condition code (see Conditional execution on page 2-18).

\(RdHi, \ RdLo\) are the destination registers. They also hold the accumulate value. \(RdHi\) and \(RdLo\) must be different registers.

\(Rn, \ Rm\) are the registers holding the values to be multiplied.

Do not use r15 for \(RdHi, RdLo, Rn, \) or \(Rm\).

**Usage**

SMLAL<xy> multiplies the signed integer from the selected half of \(Rm\) by the signed integer from the selected half of \(Rn\), and adds the 32-bit result to the 64-bit value in \(RdHi\) and \(RdLo\).

**Condition flags**

This instruction does not change the flags.

---

**Note**

SMLAL<xy> cannot raise an exception. If overflow occurs on this instruction, the result wraps round without any warning.
Architectures

This ARM instruction is available in ARMv6 and above, and E variants of ARMv5T. These 32-bit Thumb instructions are available in ARMv6T2 and above, except the ARMv7-M profile.

There is no 16-bit Thumb version of this instruction.

Examples

\[
\begin{align*}
\text{SMLALTB} & \quad r2, r3, r7, r1 \\
\text{SMLALBTVS} & \quad r0, r1, r9, r2
\end{align*}
\]
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**

\[
op\{\text{cond}\} \{Rd\}, \ Rn, \ Rm
\]

where:

- \(\text{op}\) 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.

- \(\text{cond}\) is an optional condition code (see *Conditional execution* on page 2-18).

- \(Rd\) is the destination register.

- \(Rn, \ Rm\) are the registers holding the operands.

Do not use r15 for any of \(Rd, \ Rn, \) or \(Rm\).

**Usage**

SMUAD multiplies the bottom halfword of \(Rn\) with the bottom halfword of \(Rm\), and the top halfword of \(Rn\) with the top halfword of \(Rm\). It then adds the products and stores the sum to \(Rd\).

SMUSD multiplies the bottom halfword of \(Rn\) with the bottom halfword of \(Rm\), and the top halfword of \(Rn\) with the top halfword of \(Rm\). It then subtracts the second product from the first, and stores the difference to \(Rd\).

**Condition flags**

The SMUAD instruction sets the Q flag if the addition overflows.
Architectures

These ARM instructions are available in ARMv6 and above.

These 32-bit Thumb instructions are available in ARMv6T2 and above, except the ARMv7-M profile.

There are no 16-bit Thumb versions of these instructions.

Examples

\[
\begin{align*}
\text{SMUAD} & \quad r2, r3, r2 \\
\text{SMUSDXNE} & \quad r0, r1, r2 
\end{align*}
\]
4.4.7 SMMUL, SMMLA, and SMMLS

Signed Most significant word Multiply, Signed Most significant word Multiply with Accumulation, and Signed Most significant word Multiply with Subtraction. These instructions have 32-bit operands and produce only the most significant 32-bits of the result.

Syntax

- \text{SMMUL\{R\}\{\text{cond}\}\{Rd\},\ Rn,\ Rm}
- \text{SMMLA\{R\}\{\text{cond}\}\ \ Rd,\ Rn,\ Rm,\ Ra}
- \text{SMMLS\{R\}\{\text{cond}\}\ \ Rd,\ Rn,\ Rm,\ Ra}

where:

- \text{R} is an optional parameter. If \text{R} is present, the result is rounded, otherwise it is truncated.
- \text{cond} is an optional condition code (see \text{Conditional execution} on page 2-18).
- \text{Rd} is the destination register.
- \text{Rn}, \text{Rm} are the registers holding the operands.
- \text{Ra} is a register holding the value to be added or subtracted from.

Do not use \text{r}15 for any of \text{Rd}, \text{Rn}, \text{Rm}, or \text{Ra}.

Operation

- \text{SMMUL} multiplies the values from \text{Rn} and \text{Rm}, and stores the most significant 32 bits of the 64-bit result to \text{Rd}.
- \text{SMMLA} multiplies the values from \text{Rn} and \text{Rm}, adds the value in \text{Ra} to the most significant 32 bits of the product, and stores the result in \text{Rd}.
- \text{SMMLS} multiplies the values from \text{Rn} and \text{Rm}, subtracts the product from the value in \text{Ra} shifted left by 32 bits, and stores the most significant 32 bits of the result in \text{Rd}.

If the optional \text{R} parameter is specified, 0x80000000 is added before extracting the most significant 32 bits. This has the effect of rounding the result.

Condition flags

These instructions do not change the flags.
Architectures

These ARM instructions are available in ARMv6 and above.

These 32-bit Thumb instructions are available in ARMv6T2 and above, except the ARMv7-M profile.

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

\[ \text{op}\{\text{X}\}\{\text{cond}\} \ R_d, \ R_n, \ R_m, \ R_a \]

where:

\( \text{op} \) is one of:

- SMLAD  Dual multiply, accumulate sum of products.
- SMLSD  Dual multiply, accumulate difference of products.

\( \text{cond} \) is an optional condition code (see Conditional execution on page 2-18).

\( \text{X} \) is an optional parameter. If \( \text{X} \) is present, the most and least significant halfwords of the second operand are exchanged before the multiplications occur.

\( \text{R}_d \) is the destination register.

\( \text{R}_n, \text{R}_m \) are the registers holding the operands.

\( \text{R}_a \) is the register holding the accumulate operand.

Do not use r15 for any of \( \text{R}_d, \text{R}_n, \text{R}_m \), or \( \text{R}_a \).

Operation

SMLAD multiplies the bottom halfword of \( \text{R}_n \) with the bottom halfword of \( \text{R}_m \), and the top halfword of \( \text{R}_n \) with the top halfword of \( \text{R}_m \). It then adds both products to the value in \( \text{R}_a \) and stores the sum to \( \text{R}_d \).

SMLSD multiplies the bottom halfword of \( \text{R}_n \) with the bottom halfword of \( \text{R}_m \), and the top halfword of \( \text{R}_n \) with the top halfword of \( \text{R}_m \). It then subtracts the second product from the first, adds the difference to the value in \( \text{R}_a \), and stores the result to \( \text{R}_d \).

Condition flags

These instructions do not change the flags.
Architectures

These ARM instructions are available in ARMv6 and above.

These 32-bit Thumb instructions are available in ARMv6T2 and above, except the ARMv7-M profile.

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

\[ \text{op}[\{X\}{}\text{cond}] \ RdLo, RdHi, Rn, Rm \]

where:

- \text{op} is one of:
  - SMLALD  Dual multiply, accumulate sum of products.
  - SMLSLD  Dual multiply, accumulate difference of products.

- \text{X} is an optional parameter. If \text{X} is present, the most and least significant halfwords of the second operand are exchanged before the multiplications occur.

- \text{cond} is an optional condition code (see Conditional execution on page 2-18).

- \text{RdLo}, \text{RdHi} are the destination registers for the 64-bit result. They also hold the 64-bit accumulate operand. \text{RdHi} and \text{RdLo} must be different registers.

- \text{Rn}, \text{Rm} are the registers holding the operands.

Do not use r15 for any of \text{RdLo}, \text{RdHi}, \text{Rn}, or \text{Rm}.

Operation

SMLALD multiplies the bottom halfword of \text{Rn} with the bottom halfword of \text{Rm}, and the top halfword of \text{Rn} with the top halfword of \text{Rm}. It then adds both products to the value in \text{RdLo}, \text{RdHi} and stores the sum to \text{RdLo}, \text{RdHi}.

SMLSLD multiplies the bottom halfword of \text{Rn} with the bottom halfword of \text{Rm}, and the top halfword of \text{Rn} with the top halfword of \text{Rm}. It then subtracts the second product from the first, adds the difference to the value in \text{RdLo}, \text{RdHi}, and stores the result to \text{RdLo}, \text{RdHi}.

Condition flags

These instructions do not change the flags.
Architectures

These ARM instructions are available in ARMv6 and above.

These 32-bit Thumb instructions are available in ARMv6T2 and above, except the ARMv7-M profile.

There are no 16-bit Thumb versions of these instructions.

Examples

- **SMLALD**  \( r10, r11, r5, rl \)
- **SMLS LD**  \( r3, r0, r5, rl \)
4.4.10 UMAAL

Unsigned Multiply Accumulate Accumulate Long.

**Syntax**

UMAAL{cond} RdLo, RdHi, Rn, Rm

where:

- `cond` is an optional condition code (see *Conditional execution* on page 2-18).
- `RdLo`, `RdHi` are the destination registers for the 64-bit result. They also hold the two 32-bit accumulate operands. `RdLo` and `RdHi` must be distinct registers.
- `Rn`, `Rm` are the registers holding the multiply operands.

Do not use r15 for any of `RdLo`, `RdHi`, `Rn`, or `Rm`.

**Operation**

The UMAAL instruction multiplies the 32-bit values in `Rn` and `Rm`, adds the two 32-bit values in `RdHi` and `RdLo`, and stores the 64-bit result to `RdLo`, `RdHi`.

**Condition flags**

This instruction does not change the flags.

**Architectures**

This ARM instruction is available in ARMv6 and above.

These 32-bit Thumb instructions are available in ARMv6T2 and above, except the ARMv7-M profile.

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

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, Rn, Rm
MIAPH{cond} Acc, Rn, Rm
MIA<x><y>{cond} Acc, Rn, Rm

where:

\( \text{cond} \) is an optional condition code (see Conditional execution on page 2-18).

\( \text{Acc} \) 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 in current processors.

\( \text{Rn, Rm} \) are the ARM registers holding the values to be multiplied.

Do not use r15 for either \( \text{Rn} \) or \( \text{Rm} \).

\( \langle x \rangle \langle y \rangle \) is one of: BB, BT, TB, TT.

Usage

The MIA instruction multiplies the signed integers from \( \text{Rn} \) and \( \text{Rm} \), and adds the result to the 40-bit value in \( \text{Acc} \).

The MIAPH instruction multiplies the signed integers from the bottom halves of \( \text{Rn} \) and \( \text{Rm} \), multiplies the signed integers from the upper halves of \( \text{Rn} \) and \( \text{Rm} \), and adds the two 32-bit results to the 40-bit value in \( \text{Acc} \).

The MIAxy instruction multiplies the signed integer from the selected half of \( \text{Rs} \) by the signed integer from the selected half of \( \text{Rm} \), and adds the 32-bit result to the 40-bit value in \( \text{Acc} \). \( \langle x \rangle \Rightarrow \text{B} \) means use the bottom half (bits [15:0]) of \( \text{Rn} \), \( \langle x \rangle \Rightarrow \text{T} \) means use the top half (bits [31:16]) of \( \text{Rn} \). \( \langle y \rangle \Rightarrow \text{B} \) means use the bottom half (bits [15:0]) of \( \text{Rm} \), \( \langle y \rangle \Rightarrow \text{T} \) means use the top half (bits [31:16]) of \( \text{Rm} \).
**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 coprocessor 0 instructions are only available in XScale processors.

There are no Thumb 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:

- Saturating arithmetic
- QADD, QSUB, QDADD, and QDSUB on page 4-94
- SSAT and USAT on page 4-96.

Some of the parallel instructions are also saturating, see Parallel instructions on page 4-98.

4.5.1 Saturating arithmetic

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 4-136).

The Q flag can also be set by two other instructions (see SMULxy and SMLAxy on page 4-76 and SMULWy and SMLAWy on page 4-78), 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 4-99.

Syntax

$\text{op\{cond\}} \{\text{Rd}\}, \text{Rm}, \text{Rn}$

where:

- $\text{op}$ is one of QADD, QSUB, QDADD, or QDSUB.
- $\text{cond}$ is an optional condition code (see Conditional execution on page 2-18).
- $\text{Rd}$ is the destination register.
- $\text{Rm}, \text{Rn}$ are the registers holding the operands.

Do not use r15 for $\text{Rd}, \text{Rm}, \text{or Rn}$.

Usage

The QADD instruction adds the values in $\text{Rm}$ and $\text{Rn}$.

The QSUB instruction subtracts the value in $\text{Rn}$ from the value in $\text{Rm}$.

The QDADD instruction calculates $\text{SAT}(\text{Rm} + \text{SAT}(\text{Rn} \times 2))$. 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 $\text{SAT}(\text{Rm} - \text{SAT}(\text{Rn} \times 2))$. 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 4-99 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).

Architectures

These ARM instructions are available in ARMv6 and above, and E variants of ARMv5T.

These 32-bit Thumb instructions are available in ARMv6T2 and above, except the ARMv7-M profile.

There are no 16-bit Thumb versions of these instructions.

Examples

QADD    r0, r1, r9
QDSUBLT r9, r0, r1
4.5.3 SSAT and USAT

Signed Saturate and Unsigned Saturate to any bit position, with optional shift before saturating.

SSAT saturates a signed value to a signed range.

USAT saturates a signed value to an unsigned range.

See also SSAT16 and USAT16 on page 4-104.

Syntax

\[
\text{op}\{\text{cond}\} \; \text{Rd}, \; \text{#sat}, \; \text{Rm}\{, \; \text{shift}\}
\]

where:

- \(\text{op}\) is either SSAT or USAT.
- \(\text{cond}\) is an optional condition code (see Conditional execution on page 2-18).
- \(\text{Rd}\) is the destination register. \(\text{Rd}\) must not be r15.
- \(\text{sat}\) specifies the bit position to saturate to, in the range 1 to 32 for SSAT, and 0 to 31 for USAT.
- \(\text{Rm}\) is the register containing the operand. \(\text{Rm}\) must not be r15.
- \(\text{shift}\) is an optional shift. It must be one of the following:
  - ASR \#n where \(n\) is in the range 1-32 (ARM) or 1-31 (Thumb-2)
  - 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.
These 32-bit Thumb instructions are available in ARMv6T2 and above.
There are no 16-bit Thumb versions of these instructions.

Examples

SSAT    r7, #16, r7, LSL #4
USATNE  r0, #7, r5
4.6 Parallel instructions

This section contains the following subsections:

- Parallel add and subtract on page 4-99
  Various byte-wise and halfword-wise additions and subtractions.

- USAD8 and USADA8 on page 4-102
  Unsigned sum of absolute differences, and accumulate unsigned sum of absolute differences.

- SSAT16 and USAT16 on page 4-104
  Parallel halfword saturating instructions.

There are also some parallel unpacking instructions, see SXT, SXTA, UXT, and UXTA on page 4-109.
4.6.1 Parallel add and subtract

Various byte-wise and halfword-wise additions and subtractions.

Syntax

\[
\langle \text{prefix} \rangle \text{op}\{\text{cond}\} \{\text{Rd}\}, \text{Rn}, \text{Rm}
\]

where:

- \(\langle \text{prefix} \rangle\) is one of:
  - S: Signed arithmetic modulo 2^8 or 2^16. Sets APSR GE flags.
  - Q: Signed saturating arithmetic.
  - SH: Signed arithmetic, halving the results.
  - U: Unsigned arithmetic modulo 2^8 or 2^16. Sets APSR GE flags.
  - UQ: Unsigned saturating arithmetic.
  - UH: Unsigned arithmetic, halving the results.

- \text{op} is one of:
  - ADD8: Byte-wise Addition
  - ADD16: Halfword-wise Addition.
  - SUB8: Byte-wise Subtraction.
  - SUB16: Halfword-wise Subtraction.
  - ASX: Exchange halfwords of \text{Rm}, then Add top halfwords and Subtract bottom halfwords.
  - SAX: Exchange halfwords of \text{Rm}, then Subtract top halfwords and Add bottom halfwords.

- \text{cond} is an optional condition code (see Conditional execution on page 2-18).

- \text{Rd} is the destination register. Do not use r15 for \text{Rd}.

- \text{Rm}, \text{Rn} are the ARM registers holding the operands. Do not use r15 for \text{Rm} or \text{Rn}.

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 APSR 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 $\$u$ and $\s$ prefix variants of these instructions set the GE flags in the APSR 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

- 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

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 instructions are available in ARMv6T2 and above, except the ARMv7-M profile.

There are no 16-bit Thumb versions of these instructions.

Examples

- SHADD8 \( r_4, r_3, r_9 \)
- USAXNE \( r_0, r_0, r_2 \)

Incorrect examples

- QHADD \( r_2, r_9, r_3 \); No such instruction, should be QHADD8 or QHADD16
- SAX \( r_{10}, r_8, r_5 \); 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\}, Rn, Rm
USADA8\{cond\} Rd, Rn, Rm, Ra

where:

- \textit{cond} is an optional condition code (see \textit{Conditional execution} on page 2-18).
- \textit{Rd} is the destination register.
- \textit{Rn} is the register holding the first operand.
- \textit{Rm} is the register holding the second operand.
- \textit{Ra} is the register holding the accumulate operand.

Do not use r15 for \textit{Rd}, \textit{Rn}, \textit{Rn}, or \textit{Ra}.

Operation

The USAD8 instruction finds the four differences between the unsigned values in corresponding bytes of \textit{Rn} and \textit{Rm}. It adds the absolute values of the four differences, and saves the result to \textit{Rd}.

The USADA8 instruction adds the absolute values of the four differences to the value in \textit{Ra}, and saves the result to \textit{Rd}.

Condition flags

These instructions do not alter any flags.

Architectures

These ARM instructions are available in ARMv6 and above.

These 32-bit Thumb instructions are available in ARMv6T2 and above, except the ARMv7-M profile.

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
USADA16     r0, r4, r0, r1  ; no such instruction
4.6.3 SSAT16 and USAT16

Parallel halfword Saturating instructions.
SSAT16 saturates a signed value to a signed range.
USAT16 saturates a signed value to an unsigned range.

Syntax

\[ \text{op}(\text{cond}) \ Rd, \ \#\text{sat}, \ Rn \]

where:

- \( \text{op} \) is one of:
  - SSAT16: Signed saturation.
  - USAT16: Unsigned saturation.
- \( \text{cond} \) is an optional condition code (see Conditional execution on page 2-18).
- \( \text{Rd} \) is the destination register.
- \( \text{sat} \) specifies the bit position to saturate to, and is in the range 1 to 16 for SSAT16, or 0 to 15 for USAT16.
- \( \text{Rn} \) is the register holding the operand.

Do not use r15 for \( \text{Rd} \) or \( \text{Rn} \).

Operation

Halfword-wise signed and unsigned saturation to any bit position.

The SSAT16 instruction saturates each signed halfword to the signed range
\[-2^\text{sat} - 1 \leq x \leq 2^\text{sat} - 1 - 1.\]

The USAT16 instruction saturates each signed 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 instructions are available in ARMv6T2 and above, except the ARMv7-M profile.

There are no 16-bit Thumb versions of these instructions.

Examples

\[
\begin{align*}
\text{SSAT16} & \ r7, \ #12, \ r7 \\
\text{USAT16} & \ r0, \ #7, \ r5 \\
\end{align*}
\]

Incorrect examples

\[
\begin{align*}
\text{SSAT16} & \ r1, \ #16, \ r2, \ LSL \ #4 \ ; \ \text{shifts not permitted with halfword saturations}
\end{align*}
\]
4.7 Packing and unpacking instructions

This section contains the following subsections:

- **BFC and BFI** on page 4-107
  Bit Field Clear and Bit Field Insert.

- **SBFX and UBFX** on page 4-108
  Signed or Unsigned Bit Field extract.

- **SXT, SXTA, UXT, and UXTA** on page 4-109
  Sign Extend or Zero Extend instructions, with optional Add.

- **PKHBT and PKHTB** on page 4-112
  Halfword Packing instructions.
### 4.7.1 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**
  
  \[
  \text{BFC}\{\text{cond}\} \text{ Rd, #lsb, #width}\]
  
- **BFI**
  
  \[
  \text{BFI}\{\text{cond}\} \text{ Rd, Rn, #lsb, #width}\]

Where:

- \textit{cond} is an optional condition code (see \textit{Conditional execution} on page 2-18).
- \textit{Rd} is the destination register. \textit{Rd} must not be r15.
- \textit{Rn} is the source register. \textit{Rn} must not be r15.
- \textit{lsb} is the least significant bit that is to be cleared or copied.
- \textit{width} is the number of bits to be cleared or copied. \textit{width} must not be 0, and \((\text{width} + \text{lsb})\) must be less than 32.

#### BFC

\textit{width} bits in \textit{Rd} are cleared, starting at \textit{lsb}. Other bits in \textit{Rd} are unchanged.

#### BFI

\textit{width} bits in \textit{Rd}, starting at \textit{lsb}, are replaced by \textit{width} bits from \textit{Rn}, starting at bit[0]. Other bits in \textit{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 instructions are available in ARMv6T2 and above.

There are no 16-bit Thumb versions of these instructions.
4.7.2 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

\[ \text{op\{cond\} Rd, Rn, } \#\text{lsb}, \#width \]

where:

- \text{op} is either SBFX or UBFX.
- \text{cond} is an optional condition code (see Conditional execution on page 2-18).
- \text{Rd} is the destination register.
- \text{Rn} is the source register.
- \text{lsb} is the bit number of least significant bit in the bitfield, in the range 0 to 31.
- \text{width} is the width of the bitfield, in the range 1 to \((32 - \text{lsb})\).

Do not use r15 for \text{Rd} or \text{Rn}.

Condition flags

These instructions do not alter any flags.

Architectures

These ARM instructions are available in ARMv6T2 and above.

These 32-bit Thumb instructions are available in ARMv6T2 and above.

There are no 16-bit Thumb versions of these instructions.
4.7.3 SXT, SXTA, UXT, and UXTA

Sign extend, Sign extend with Add, Zero extend, and Zero extend with Add.

Syntax

\[
\begin{align*}
\text{SXT} & \langle \text{extend} \rangle \{ \text{cond} \} \{ \text{Rd} \}, \text{Rm} \{, \text{rotation} \} \\
\text{SXTA} & \langle \text{extend} \rangle \{ \text{cond} \} \{ \text{Rd} \}, \text{Rn}, \text{Rm} \{, \text{rotation} \} \\
\text{UXT} & \langle \text{extend} \rangle \{ \text{cond} \} \{ \text{Rd} \}, \text{Rm} \{, \text{rotation} \} \\
\text{UXTA} & \langle \text{extend} \rangle \{ \text{cond} \} \{ \text{Rd} \}, \text{Rn}, \text{Rm} \{, \text{rotation} \}
\end{align*}
\]

where:

\[\langle \text{extend} \rangle\] 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.

\[\text{cond}\] is an optional condition code (see Conditional execution on page 2-18).

\[\text{Rd}\] is the destination register.

\[\text{Rn}\] is the register holding the number to add (SXTA and UXTA only).

\[\text{Rm}\] is the register holding the value to extend.

\[\text{rotation}\] is one of:
- ROR #8 Value from \text{Rm} is rotated right 8 bits.
- ROR #16 Value from \text{Rm} is rotated right 16 bits.
- ROR #24 Value from \text{Rm} is rotated right 24 bits.

If \text{rotation} is omitted, no rotation is performed.

You must not use r15 for \text{Rd}, \text{Rn}, or \text{Rm}. 

**Operation**

These instructions do the following:

1. Rotate the value from $Rm$ right by 0, 8, 16 or 24 bits.

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**

Only the following forms are 16-bit instructions when used in Thumb code, and only these forms are available in pre-Thumb-2 Thumb code:

- **SXTB** $Rd$, $Rm$  \(Rd\) and \(Rm\) must both be Lo registers.
- **SXTH** $Rd$, $Rm$  \(Rd\) and \(Rm\) must both be Lo registers.
- **UXTB** $Rd$, $Rm$  \(Rd\) and \(Rm\) must both be Lo registers.
- **UXTH** $Rd$, $Rm$  \(Rd\) and \(Rm\) must both be Lo registers.

**Architectures**

These ARM instructions are available in ARMv6 and above.

- **SXTA** and **UXTA** Thumb instructions are available in ARMv6T2 and above, except the ARMv7-M profile.
- **SXTB16** and **UXTB16** Thumb instructions are available in ARMv6T2 and above, except the ARMv7-M profile.
- **SXTB**, **SXTH**, **UXTB**, and **UXTH** Thumb instructions are available in ARMv6 and above.
Examples

SXTH     r3, r9, r4
UXTAB16EQ r0, r0, r4, ROR #16

Incorrect examples

SXTH     r9, r3, r2, ROR #12 ; rotation must be by 0, 8, 16, or 24.
4.7.4 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

PKHBT{cond} {Rd}, Rn, Rm[, LSL #leftshift]
PKHTB{cond} {Rd}, Rn, Rm[, ASR #rightshift]

where:

PKHBT Combines bits[15:0] of Rn with bits[31:16] of the shifted value from Rm.
PKHTB Combines bits[31:16] of Rn with bits[15:0] of the shifted value from Rm.

cond is an optional condition code (see Conditional execution on page 2-18).
Rd is the destination register.
Rn is the register holding the first operand.
Rm is the register holding the first operand.
leftshift is in the range 0 to 31.
rightshift is in the range 1 to 32.

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 instructions are available in ARMv6T2 and above, except the ARMv7-M profile.

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

PKHBTEQ r4, r5, r1, ASR #8 ; ASR not permitted with PKHBT
4.8 Branch and control instructions

This section contains the following subsections:

- **B, BL, BX, BLX, and BXJ** on page 4-115
  Branch, Branch with Link, Branch and exchange instruction set, Branch with Link and exchange instruction set, Branch and change instruction set to Jazelle.

- **IT** on page 4-118
  If-Then. IT makes up to four following instructions conditional, with either the same condition, or some with one condition and others with the inverse condition. IT is available only in Thumb-2.

- **CBZ and CBNZ** on page 4-121
  Compare against zero and branch. These instructions are available only in Thumb-2.

- **TBB and TBH** on page 4-122
  Table Branch Byte or Halfword. These instructions are available only in Thumb-2.
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

\[ op1\{cond\}\{.W\} \text{ label} \]
\[ op2\{cond\} \text{ Rm} \]

where:

- \( op1 \) is one of:
  - B Branch.
  - BL Branch with link.
  - BLX Branch with link, and exchange instruction set.

- \( op2 \) is one of:
  - 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 2-18). \( cond \) is not available on all forms of this instruction, see *Instruction availability and branch ranges* on page 4-116.

- \( .W \) 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 4-117 for details.

- \( \text{label} \) is a program-relative expression. See *Register-relative and program-relative expressions* on page 3-37 for more information.

- \( \text{Rm} \) is a register containing an address to branch to.

Operation

All these instructions cause a branch to \( \text{label} \), or to the address contained in \( \text{Rm} \). In addition:

- The BL and BLX instructions copy the address of the next instruction into lr (r14, the link register).

- The BX and BLX instructions can change the processor state from ARM to Thumb, or from Thumb to ARM.
  - BLX \( \text{label} \) 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 4-7 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>
<thead>
<tr>
<th>Instruction</th>
<th>ARM</th>
<th>16-bit Thumb</th>
<th>32-bit Thumb</th>
</tr>
</thead>
<tbody>
<tr>
<td>B label</td>
<td>±32MB (All)</td>
<td>±2KB (All T)</td>
<td>±16MBa (All T2)</td>
</tr>
<tr>
<td>B{cond} label</td>
<td>±32MB (All)</td>
<td>-252 to +258 (All T)</td>
<td>±1MBa (All T2)</td>
</tr>
<tr>
<td>BL label</td>
<td>±32MB (All)</td>
<td>±4MB b (All T)</td>
<td>±16MB (All T2)</td>
</tr>
<tr>
<td>BL{cond} label</td>
<td>±32MB (All)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>BX ( Rm ) c</td>
<td>Available (4T, 5)</td>
<td>Available (All T)</td>
<td>Use 16-bit (All T2)</td>
</tr>
<tr>
<td>BX{cond} ( Rm ) c</td>
<td>Available (4T, 5)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>BLX label</td>
<td>±32MB (5)</td>
<td>±4MB b (5T)</td>
<td>±16MB (All T2 except ARMv7-M)</td>
</tr>
<tr>
<td>BLX ( Rm )</td>
<td>Available (5)</td>
<td>Available (5T)</td>
<td>Use 16-bit (All T2)</td>
</tr>
<tr>
<td>BLX{cond} ( Rm )</td>
<td>Available (5)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>BXJ ( Rm )</td>
<td>Available (5J, 6)</td>
<td>-</td>
<td>Available (All T2 except ARMv7-M)</td>
</tr>
<tr>
<td>BXJ{cond} ( Rm )</td>
<td>Available (5J, 6)</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

- Use \( .W \) to instruct the assembler to use this 32-bit instruction.
- a. This is an instruction pair.
- c. The assembler accepts BX{cond} \( Rm \) for code assembled for ARMv4 and converts it to MOV{cond} PC, \( Rm \) at link time, unless objects targeted for ARMv4T are present.
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 Chapter 3 Using the Basic Linker Functionality in the Linker User 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 instruction.

BX, BLX, and BXJ in Thumb-2EE

These instructions can be used as branches in Thumb-2EE code, but cannot be used to change state. You cannot use the op{cond} label form of these instructions in Thumb-2EE. In the register form, bit[0] of Rm must be 1, and execution continues at the target address in ThumbEE state.

Condition flags

These instructions do not change the flags.

Architectures

See Instruction availability and branch ranges on page 4-116 for details of availability of these instructions in each architecture.

Examples

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>loopA</td>
</tr>
<tr>
<td>BLE</td>
<td>ng+8</td>
</tr>
<tr>
<td>BL</td>
<td>subC</td>
</tr>
<tr>
<td>BLLT</td>
<td>rtX</td>
</tr>
<tr>
<td>BEQ</td>
<td>{pc}+4 ; #0x8004</td>
</tr>
</tbody>
</table>
4.8.2 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.

The instructions (including branches) in the IT block must also specify the condition in the \{cond\} part of their syntax.

You do not need to write IT instructions in your code, because the assembler generates them for you automatically according to the conditions specified on the following instructions. However, if you do write IT instructions, the assembler validates the conditions specified in the IT instructions against the conditions specified in the following instructions.

Writing the IT instructions ensures that you consider the placing of conditional instructions, and the choice of conditions, in the design of your code.

When assembling to ARM code, the assembler performs the same checks, but does not generate any IT instructions.

**Syntax**

\texttt{IT\{x\{y\{z\}\}\} \{cond\}}

where:

\begin{align*}
x & \quad \text{specifies the condition switch for the second instruction in the IT block.} \\
y & \quad \text{specifies the condition switch for the third instruction in the IT block.} \\
z & \quad \text{specifies the condition switch for the fourth instruction in the IT block.} \\
\text{cond} & \quad \text{specifies the condition for the first instruction in the IT block.}
\end{align*}

The condition switch for the second, third and fourth instruction in the IT block can be either:

\begin{align*}
\text{T} & \quad \text{Then. Applies the condition} \text{\ cond} \text{ to the instruction.} \\
\text{E} & \quad \text{Else. Applies the inverse condition of} \ \text{cond} \text{ to the instruction.}
\end{align*}
Usage

With the exception of CMP, CMN, and TST, the 16-bit instructions that normally affect the condition code flags, do not affect them when used inside an IT block.

A BKPT instruction in an IT block is always executed, even if its condition fails.

Note

You can use an IT block for unconditional instructions by using the AL condition.

Conditional branches inside an IT block have a longer branch range than those outside the IT block.

Restrictions

The following instructions are not permitted in an IT block:

- IT
- CBZ and CBNZ
- TBB and TBH
- CPS, CPSID and CPSIE
- SETEND.

Other restrictions when using an IT block are:

- A branch or any instruction that modifies the pc is only allowed in an IT block if it is the last instruction in the block.
- You cannot branch to any instruction in an IT block, unless when returning from an exception handler.
- You cannot use any assembler directives in an IT block.

Condition flags

This instruction does not change the flags.
Exceptions

Exceptions can occur between an IT instruction and the corresponding IT block, or within an IT block. This exception results in entry to the appropriate exception handler, with suitable return information in lr and SPSR.

Instructions designed for use 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 instruction is available in ARMv6T2 and above.

Example

ITTE NE ; IT can be omitted
ANDNE r0,r0,r1 ; 16-bit AND, not ANDS
ADDSNE r2,r2,#1 ; 32-bit ADDS (16-bit ADDS does not set flags in IT block)
MOVEQ r2,r3 ; 16-bit MOV
ITT AL ; emit 2 non-flag setting 16-bit instructions
ADDAL r0,r0,r1 ; 16-bit ADD, not ADDS
SUBAL r2,r2,#1 ; 16-bit SUB, not SUB
ADD r0,r0,r1 ; expands into 32-bit ADD
IT NE
ADD r0,r0,r1 ; syntax error: no condition code used in IT block
ITT EQ
MOVEQ r0,r1
BEQ dloop

Notes

Unpredictable instructions in an IT block

The assembler warns about unpredictable instructions in an IT block, for example, B, BL, and CPS. It also warns about instructions that change the pc, for example, BX, CBZ, and RFE.
4.8.3 CBZ and CBNZ

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

Syntax

CBZ \textit{Rn}, \textit{label} \\
CBNZ \textit{Rn}, \textit{label}

where:
\begin{itemize}
  \item \textit{Rn} is the register holding the operand.
  \item \textit{label} is the branch destination.
\end{itemize}

Usage

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

Except that it does not change the condition code flags, CBZ \textit{Rn}, \textit{label} is equivalent to:
\begin{verbatim}
    CMP \textit{Rn}, #0
    BEQ \textit{label}
\end{verbatim}

Except that it does not change the condition code flags, CBNZ \textit{Rn}, \textit{label} is equivalent to:
\begin{verbatim}
    CMP \textit{Rn}, #0
    BNE \textit{label}
\end{verbatim}

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 instructions are available in ARMv6T2 and above.

There are no ARM or 32-bit Thumb versions of these instructions.
4.8.4 TBB and TBH

Table Branch Byte and Table Branch Halfword.

Syntax

TBB [Rn, Rm]
TBH [Rn, Rm, LSL #1]

where:
Rn is the base register. This contains the address of the table of branch lengths. Rn must not be r13.
If r15 is specified for Rn, the value used is the address of the instruction plus 4.
Rm is the index register. This contains an index into the table.
Rm must not be r15 or r13.

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

In Thumb-2EE, if the value in the base register is zero, execution branches to the NullCheck handler at HandlerBase - 4.

Architectures

These 32-bit Thumb instructions are available in ARMv6T2 and above.

There are no ARM, or 16-bit Thumb, versions of these instructions.
4.9 Coprocessor instructions

This section does not describe VFP (see Chapter 5 NEON and VFP Programming) or Wireless MMX™ Technology instructions (see Chapter 6 Wireless MMX Technology Instructions). XScale-specific instructions are described later in this chapter (see Miscellaneous instructions on page 4-131).

It contains the following sections:

- **CDP and CDP2 on page 4-124**
  Coprocessor Data oPerations.

- **MCR, MCR2, MCRR, and MCRR2 on page 4-125**
  Move to Coprocessor from ARM Register or Registers, possibly with coprocessor operations.

- **MRC, MRC2, MRRC and MRRC2 on page 4-127**
  Move to ARM Register or Registers from Coprocessor, possibly with coprocessor operations.

- **LDC, LDC2, STC, and STC2 on page 4-129**
  Transfer data between memory and Coprocessor.

--- Note ---

A coprocessor instruction causes an Undefined Instruction exception if the specified coprocessor is not present, or if it is not enabled.
4.9.1 CDP and CDP2

Coprocessor data operations.

Syntax

\[ \text{op}\{\text{cond}\} \text{coproc}, \#\text{opcode1}, \text{CRd}, \text{CRn}, \text{CRm}\{, \#\text{opcode2}\} \]

where:

- \( \text{op} \) is either CDP or CDP2.
- \( \text{cond} \) is an optional condition code (see Conditional execution on page 2-18). In ARM code, \( \text{cond} \) is not allowed for CDP2.
- \( \text{coproc} \) is the name of the coprocessor the instruction is for. The standard name is \( p_n \), where \( n \) is an integer in the range 0 to 15.
- \( \text{opcode1} \) is a 4-bit coprocessor-specific opcode.
- \( \text{opcode2} \) is an optional 3-bit coprocessor-specific opcode.
- \( \text{CRd}, \text{CRn}, \text{CRm} \) are coprocessor registers.

Usage

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

Architectures

The CDP ARM instruction is available in all versions of the ARM architecture.

The CDP2 ARM instruction is available in ARMv5T and above.

These 32-bit Thumb instructions are available in ARMv6T2 and above.

There are no 16-bit 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

\[ \text{op1}\{\text{cond}\} \text{ coproc, } \#\text{opcode1}, \text{Rt, CRn, CRm}\{, \#\text{opcode2}\} \]

\[ \text{op2}\{\text{cond}\} \text{ coproc, } \#\text{opcode3}, \text{Rt, Rt2, CRm} \]

where:

- \text{op1} is either \text{MCR} or \text{MCR2}.
- \text{op2} is either \text{MCRR} or \text{MCRR2}.
- \text{cond} is an optional condition code (see \textit{Conditional execution} on page 2-18). In ARM code, \text{cond} is not allowed for \text{MCR2} or \text{MCRR2}.
- \text{coproc} is the name of the coprocessor the instruction is for. The standard name is p\text{n}, where \text{n} is an integer in the range 0 to 15.
- \text{opcode1} is a 3-bit coprocessor-specific opcode.
- \text{opcode2} is an optional 3-bit coprocessor-specific opcode.
- \text{opcode3} is a 4-bit coprocessor-specific opcode.
- \text{Rt, Rt2} are ARM source registers. \text{MCRR} or \text{MCRR2} must not use r15.
- \text{CRn, CRm} are coprocessor registers.

Usage

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

The MCR ARM instruction is available in all versions of the ARM architecture.
The MCR2 ARM instruction is available in ARMv5T and above.
The MCRR ARM instruction is available in ARMv6 and above, and E variants of ARMv5T.
The MCRR2 ARM instruction is available in ARMv6 and above.
These 32-bit Thumb instructions are available in ARMv6T2 and above.
There are no 16-bit 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**

\[
\text{op1}\{\text{cond}\} \coproc, \#\text{opcode1}, \text{Rt}, \text{CRn}, \text{CRm}\{, \#\text{opcode2}\} \\
\text{op2}\{\text{cond}\} \coproc, \#\text{opcode3}, \text{Rt}, \text{Rt2}, \text{CRm}
\]

where:

- \text{op1} is either MRC or MRC2.
- \text{op2} is either MRRC or MRRC2.
- \text{cond} is an optional condition code (see Conditional execution on page 2-18). In ARM code, \text{cond} is not allowed for MRC2 or MRRC2.
- \coproc is the name of the coprocessor the instruction is for. The standard name is \text{pn}, where \text{n} is an integer in the range 0 to 15.
- \text{opcode1} is a 3-bit coprocessor-specific opcode.
- \text{opcode2} is an optional 3-bit coprocessor-specific opcode.
- \text{opcode3} is a 4-bit coprocessor-specific opcode.
- \text{Rt}, \text{Rt2} are ARM source registers. Do not use r15. In MRC and MRC2, \text{Rt} can be APSR\_nzcv.
- \text{CRn}, \text{CRm} are coprocessor registers.

**Usage**

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

The MRC ARM instruction is available in all versions of the ARM architecture.

The MRC2 ARM instruction is available in ARMv5T and above.

The MRRC ARM instruction is available in ARMv6 and above, and E variants of ARMv5T.

The MRRC2 ARM instruction is available in ARMv6 and above.

These 32-bit Thumb instructions are available in ARMv6T2 and above.

There are no 16-bit Thumb versions of these instructions.
4.9.4 LDC, LDC2, STC, and STC2

Transfer Data between memory and Coprocessor.

**Syntax**

\[
\text{op}\{L}\{\text{cond}\} \coproc, \text{CRd}, \{\text{Rn}\}
\]

\[
\text{op}\{L\}\{\text{cond}\} \coproc, \text{CRd}, \{\text{Rn}, \#\{-\}\text{offset}\}\!
\]

\[
\text{op}\{L\}\{\text{cond}\} \coproc, \text{CRd}, \{\text{Rn}, \#\{-\}\text{offset}\}
\]

\[
\text{op}\{L\}\{\text{cond}\} \coproc, \text{CRd}, \{\text{label}\}
\]

where:

- \(\text{op}\) is one of LDC, LDC2, STC, or STC2.
- \(\text{cond}\) is an optional condition code (see Conditional execution on page 2-18). In ARM code, \(\text{cond}\) is not allowed for LDC2 or STC2.
- \(\text{L}\) is an optional suffix specifying a long transfer.
- \(\coproc\) is the name of the coprocessor the instruction is for. The standard name is \(p_n\), where \(n\) is an integer in the range 0 to 15.
- \(\text{CRd}\) is the coprocessor register to load or store.
- \(\text{Rn}\) 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 \(\text{Rn}\). Otherwise, the offset is added to \(\text{Rn}\).
- \(\text{offset}\) 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 \(\text{Rn}\).
- \(\text{label}\) is a word-aligned program-relative expression. See Register-relative and program-relative expressions on page 3-37 for more information. \(\text{label}\) must be within 1020 bytes of the current instruction.
Usage

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

In Thumb-2EE, if the value in the base register is zero, execution branches to the NullCheck handler at HandlerBase - 4.

Architectures

LDC and STC are available in all versions of the ARM architecture.

LDC2 and STC2 are available in ARMv5T and above.

These 32-bit Thumb instructions are available in ARMv6T2 and above.

There are no 16-bit Thumb versions of these instructions.

Notes

Use of program relative addressing in the STC and STC2 instructions is deprecated.
4.10 Miscellaneous instructions

This section contains the following subsections:

- **BKPT** on page 4-132
  Breakpoint.

- **SVC** on page 4-133
  Supervisor Call (formerly SWI).

- **MRS** on page 4-134
  Move the contents of the CPSR or SPSR to a general-purpose register.

- **MSR** on page 4-136
  Load specified fields of the CPSR or SPSR with an immediate constant, or from the contents of a general-purpose register.

- **CPS** on page 4-138
  Change Processor State.

- **SMC** on page 4-140
  Secure Monitor Call (formerly SMI).

- **SETEND** on page 4-141
  Set the Endianness bit in the CPSR.

- **NOP, SEV, WFE, WFI, and YIELD** on page 4-142
  No Operation, Set Event, Wait For Event, Wait for Interrupt, and Yield hint instructions.

- **DBG, DMB, DSB, and ISB** on page 4-144
  Debug, Data Memory Barrier, Data Synchronization Barrier, and Instruction Synchronization Barrier hint instructions.

- **MAR and MRA** on page 4-147
  Transfer between two general-purpose registers and a 40-bit internal accumulator (XScale coprocessor 0 instructions).
4.10.1 BKPT

Breakpoint.

Syntax

BKPT #immed

where:

immed is an expression evaluating to an integer in the range:

- 0-65535 (a 16-bit value) in an ARM instruction
- 0-255 (an 8-bit value) in a 16-bit Thumb instruction.

Usage

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

In both ARM state and Thumb state, 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 ARMv5T and above.

This 16-bit Thumb instruction is available in ARMv5T and above.

There is no 32-bit Thumb version of this instruction.
4.10.2 SVC

SuperVisor Call.

Syntax

\[
\text{SVC}\{\text{cond}\} \ #\text{immed}
\]

where:

\text{cond} \quad \text{is an optional condition code (see Conditional execution on page 2-18).}

\text{immed} \quad \text{is an expression evaluating to an integer in the range:}

\begin{itemize}
  \item 0 to \(2^{24}-1\) (a 24-bit value) in an ARM instruction
  \item 0-255 (an 8-bit value) in a 16-bit Thumb instruction.
\end{itemize}

Usage

The SVC instruction causes an exception. This means that the processor mode changes to Supervisor, the CPSR is saved to the Supervisor mode SPSR, and execution branches to the SVC vector (see Chapter 6 Handling Processor Exceptions in the Developer Guide).

\text{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. In this release of RVCT, SWI instructions disassemble to SVC, with a comment to say that this was formerly SWI.

Condition flags

This instruction does not change the flags.

Architectures

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

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

Move the contents of a PSR to a general-purpose register.

Syntax

MRS{cond} Rd, psr

where:

cond is an optional condition code (see Conditional execution on page 2-18).
Rd is the destination register. Rd must not be r15.
psr is one of:

APSR on any processor, in any mode.
CPSR deprecated synonym for APSR and for use in Debug state, on any processor except ARMv7-M and ARMv6-M.
SPSR on any processor except ARMv7-M and ARMv6-M, in privileged modes only.
Mpsr on ARMv7-M and ARMv6-M processors only.

Mpsr can be any of: IPSR, EPSR, IEPSR, IAPSR, EAPSR, MSP, PSP, XPSR, PRIMASK, BASEPRI, BASEPRI_MAX, FAULTMASK, or CONTROL.

Usage

Use MRS in combination with MSR as part of a read-modify-write sequence for updating a PSR, for example to change processor mode, or to clear the Q flag.

In process swap code, the programmers’ model state of the process being swapped out must be saved, including relevant PSR contents. Similarly, the state of the process being swapped in must also be restored. These operations make use of MRS/store and load/MSR instruction sequences.

SPSR

You must not attempt to access the SPSR when the processor is in User or System mode. This is your responsibility. The assembler cannot warn you about this, because it has no information about the processor mode at execution time.

If you attempt to access the SPSR when the processor is in User or System mode, the result is unpredictable.
CPSR

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.

The condition flags can be read in any mode on any processor. Use APSR instead of CPSR.

Condition flags

This instruction does not change the flags.

Architectures

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

These 32-bit Thumb instructions are available in ARMv6T2 and above.

There is no 16-bit Thumb version of this instruction.
4.10.4 MSR

Load an immediate constant, or the contents of a general-purpose register, into specified fields of a *Program Status Register* (PSR).

**Syntax (except ARMv7-M and ARMv6-M)**

`MSR{cond} APSR_flags, #constant`

`MSR{cond} APSR_flags, Rm`

`MSR{cond} psr_fields, #constant`

`MSR{cond} psr_fields, Rm`

where:

- **cond** is an optional condition code (see *Conditional execution* on page 2-18).
- **flags** specifies the APSR flags to be moved. *Flags* can be one or more of:
  - `nzcvq` ALU flags field mask, PSR[31:27] (User mode)
  - `g` SIMD GE flags field mask, PSR[19:16] (User mode).
- **constant** is an expression evaluating to a numeric constant. The constant must correspond to an 8-bit pattern rotated by an even number of bits within a 32-bit word. Not available in Thumb.
- **Rm** is the source register.
- **psr** is one of:
  - **CPSR** for use in Debug state, also deprecated synonym for APSR
  - **SPSR** on any processor, in privileged modes only.
- **fields** specifies the SPSR or CPSR fields to be moved. *Fields* can be one or more of:
  - `c` control field mask byte, PSR[7:0] (privileged modes)
  - `x` extension field mask byte, PSR[15:8] (privileged modes)
  - `s` status field mask byte, PSR[23:16] (privileged modes)
  - `f` flags field mask byte, PSR[31:24] (privileged modes).
Syntax (ARMv7-M and ARMv6-M only)

MSR\{cond\} psr, \textit{Rm}

where:

\textit{cond} is an optional condition code (see \textit{Conditional execution} on page 2-18).
\textit{Rm} is the source register.
\textit{psr} can be any of: APSR, IPSR, EPSR, IEPSR, IAPSR, EAPSR, XPSR, MSP, PSP, PRIMASK, BASEPRI, BASEPRI\_MAX, FAULTMASK, or CONTROL.

Usage

See \textit{MRS} on page 4-134.

In User mode:

- Use APSR to access condition flags, Q, or GE bits.
- Writes to unallocated, privileged or execution state bits in the CPSR are ignored. This ensures that User mode programs cannot change to a privileged mode.

If you access the SPSR when in User or System mode, the result is unpredictable.

Condition flags

This instruction updates the flags explicitly if the APSR\_nzcvq or CPSR\_f field is specified.

Architectures

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

This 32-bit Thumb instruction is available in ARMv6T2 and above.

There is no 16-bit 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**

\[ \text{CPS}\text{effect iflags}\{, \#mode} \]

\[ \text{CPS \#mode} \]

where:

- \text{effect} is one of:
  - \text{IE} Interrupt or abort enable.
  - \text{ID} Interrupt or abort disable.

- \text{iflags} is a sequence of one or more of:
  - \text{a} Enables or disables imprecise aborts.
  - \text{i} Enables or disables IRQ interrupts.
  - \text{f} Enables or disables FIQ interrupts.

- \text{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 pre-Thumb-2 Thumb code, and are 16-bit instructions when used in Thumb-2 code:

- \text{CPSIE iflags}
- \text{CPSID iflags}

You cannot specify a mode change in a 16-bit Thumb instruction.
Architectures

This ARM instruction is available in ARMv6 and above.

This 32-bit Thumb instruction are available in ARMv6T2 and above.

This 16-bit Thumb instruction is available in T variants of ARMv6 and above.

Examples

CPSIE if ; enable 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
4.10.6 SMC

Secure Monitor Call.

For details see the ARM Architecture Reference Manual.

Syntax

SMC{cond} #immed_4

where:

cond is an optional condition code (see Conditional execution on page 2-18).

immed_4 is a 4-bit immediate value. This is ignored by the ARM processor, but can be used by the SMC exception handler to determine what service is being requested.

Note

As part of the development of the ARM assembly language, the SMI instruction has been renamed to SMC. In this release of RVCT, SMI instructions disassemble to SMC, with a comment to say that this was formerly SMI.

Architectures

This ARM instruction is available in implementations of ARMv6 and above, if they have the Security Extensions.

This 32-bit Thumb instruction is available in implementations of ARMv6T2 and above, if they have the Security Extensions.

There is no 16-bit 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:

specifier is one of:

BE       Big-endian.
LE       Little-endian.

Usage

Use SETEND to access data of different endianness, for example, to access several big-endian DMA-formatted data fields from an otherwise little-endian application.

Architectures

This ARM instruction is available in ARMv6 and above.

This 16-bit Thumb instruction is available in T variants of ARMv6 and above, except the ARMv6-M and ARMv7-M profiles.

There is no 32-bit Thumb 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**

\[
\begin{align*}
\text{NOP}\{\text{cond}\} \\
\text{SEV}\{\text{cond}\} \\
\text{WFE}\{\text{cond}\} \\
\text{WFI}\{\text{cond}\} \\
\text{YIELD}\{\text{cond}\}
\end{align*}
\]

where:

\[\text{cond}\] is an optional condition code (see *Conditional execution* on page 2-18).

**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 treats it as a pseudo-instruction and generates an alternative instruction that does nothing, such as `MOV r0, r0` (ARM) or `MOV r8, r8` (Thumb).

`NOP` is not necessarily a time-consuming `NOP`. The processor might remove it from the pipeline before it reaches the execution stage.

You can use `NOP` for padding, for example to place the following instruction on a 64-bit boundary.

**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 ARMv6K and above.

These 32-bit Thumb instructions are available in ARMv6T2 and above.

These 16-bit Thumb instructions are available in ARMv6T2 and above.

NOP is available on all other ARM and Thumb architectures as a pseudo-instruction. SEV, WFE, WFI, and YIELD execute as NOP instructions in ARMv6T2.
4.10.9 DBG, DMB, DSB, and ISB

Debug, Data Memory Barrier, Data Synchronization Barrier, and Instruction Synchronization Barrier.

Syntax

DBG{cond} {#option}
DMB{cond} {#option}
DSB{cond} {#option}
ISB{cond} {#option}

where:

cond is an optional condition code (see Conditional execution on page 2-18).
option is an optional limitation on the operation of the hint.

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.

DBG

Debug hint provides a hint to debug and related systems. See their documentation for what use (if any) they make of this instruction.

DMB

Data Memory Barrier acts as a memory barrier. It ensures that all explicit memory accesses that appear in program order before the DMB instruction are observed before any explicit memory accesses that appear in program order after the DMB instruction. It does not affect the ordering of any other instructions executing on the processor.

Allowed values of option are:

SY Full system DMB operation. This is the default and can be omitted.
ST DMB operation that waits only for stores to complete.
ISH DMB operation only to the inner shareable domain.
ISHST DMB operation that waits only for stores to complete, and only to the inner shareable domain.
NSH  DMB operation only out to the point of unification.
NSHST DMB operation that waits only for stores to complete and only out to the point of unification.
OSH  DMB operation only to the outer shareable domain.
OSHST DMB operation that waits only for stores to complete, and only to the outer shareable domain.

**DSB**

Data Synchronization Barrier acts as a special kind of memory barrier. No instruction in program order after this instruction executes until this instruction completes. This instruction completes when:

- All explicit memory accesses before this instruction complete.
- All Cache, Branch predictor and TLB maintenance operations before this instruction complete.

Allowed values of *option* are:

SY  Full system DSB operation. This is the default and can be omitted.
ST  DSB operation that waits only for stores to complete.
ISH DSB operation only to the inner shareable domain.
ISHST DSB operation that waits only for stores to complete, and only to the inner shareable domain.
NSH DSB operation only out to the point of unification.
NSHST DSB operation that waits only for stores to complete and only out to the point of unification.
OSH DSB operation only to the outer shareable domain.
OSHST DSB operation that waits only for stores to complete, and only to the outer shareable domain.
**ISB**

Instruction Synchronization Barrier flushes the pipeline in the processor, so that all instructions following the ISB are fetched from cache or memory, after the instruction has been completed. It ensures that the effects of context altering operations, such as changing the ASID, or completed TLB maintenance operations, or branch predictor maintenance operations, as well as all changes to the CP15 registers, executed before the ISB instruction are visible to the instructions fetched after the ISB.

In addition, the ISB instruction ensures that any branches that appear in program order after it are always written into the branch prediction logic with the context that is visible after the ISB instruction. This is required to ensure correct execution of the instruction stream.

Allowed values of *option* are:

- **SY** Full system ISB operation. This is the default, and can be omitted.

**Alias**

The following alternative values of *option* are supported for DMB and DSB, but ARM recommends that you do not use them:

- **SH** is an alias for **ISH**
- **SHST** is an alias for **ISHST**
- **UN** is an alias for **NSH**
- **UNST** is an alias for **NSHST**

**Architectures**

These ARM and 32-bit Thumb instructions are available in ARMv7.

There are no 16-bit Thumb versions of these instructions.
4.10.10 MAR and MRA

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 2-18).
- **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. RdLo and RdHi must not be the pc, and for MRA they must be different 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 coprocessor 0 instructions are only available in XScale processors.

There are no Thumb versions of these instructions.

**Examples**

- MAR acc0, r0, r1
- MRA r4, r5, acc0
- MARNE acc0, r9, r2
- MRACT r4, r8, acc0
### 4.11 Instruction width selection in Thumb

If you are writing Thumb code for ARMv6T2 or later processors, some instructions can have either a 16-bit encoding or a 32-bit encoding. The assembler normally generates the 16-bit encoding where both are available. (See *Different behavior for some instructions* for exceptions to this behavior.)

#### 4.11.1 Instruction width specifiers, .W and .N

If you want to over-ride this behavior, you can use the .W width specifier. This forces the assembler to generate a 32-bit encoding, even if a 16-bit encoding is available.

You can use the .W specifier in code that might be assembled to either ARM or Thumb (ARMv6T2 or later) code. The .W specifier has no effect when assembling to ARM code.

If you want to be sure that an instruction is encoded in 16 bits, you can use the .N width specifier. In this case, if the instruction cannot be encoded in 16 bits or you are assembling to ARM code, the assembler generates an error.

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

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

#### 4.11.2 Different behavior for some instructions

For forward references, LDR, ADR, and B without .W always generate a 16-bit instruction, even if that results in failure for a target that could be reached using a 32-bit instruction.

For external references, LDR and B without .W always generate a 32-bit instruction.
4.11.3 Diagnostic warning

You can use a diagnostic warning to detect when a branch instruction could have been encoded in 16 bits, but has been encoded in 32 bits because you specified `.W`:

```
--diag_warning 1607
```

This warning is off by default.

--- Note ---

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

Apart from ENTERX and LEAVEX, these ThumbEE instructions are only accepted when the assembler has been switched into the ThumbEE state using the --thumbx command line option or the THUMBX directive.

This section contains the following subsections:

- **ENTERX and LEAVEX** on page 4-151
  Switch between Thumb state and ThumbEE state.

- **CHKA** on page 4-152
  Check array.

- **HB, HBL, HBLP, and HBP** on page 4-153
  Handler Branch, branches to a specified handler.
4.12.1 ENTERX and LEAVEX

Switch between Thumb state and ThumbEE state.

Syntax

ENTERX
LEAVEX

Usage

ENTERX causes a change from Thumb state to ThumbEE state, or has no effect in ThumbEE state.

LEAVEX causes a change from ThumbEE state to Thumb state, or has no effect in Thumb state.

Do not use ENTERX or LEAVEX in an IT block.

Architectures

These instructions are not available in the ARM instruction set.

These 32-bit Thumb and Thumb-2EE instructions are available in ARMv7, with Thumb-2EE support.

There are no 16-bit Thumb-2 versions of these instructions.

For details see the ARM Architecture Reference Manual Thumb-2 Execution Environment Supplement.
4.12.2 CHKA

CHKA (Check Array) compares the unsigned values in two registers.

If the value in the first register is lower than, or the same as, the second, it copies the pc to the lr, and causes a branch to the IndexCheck handler.

**Syntax**

CHKA \( Rn, Rm \)

where:

- \( Rn \) contains the array size. Do not use r15.
- \( Rm \) contains the array index. Do not use r13 or r15.

**Architectures**

This instruction is not available in ARM state.

This 16-bit ThumbEE instruction is only available in ARMv7, with Thumb-2EE support.
4.12.3 HB, HBL, HBLP, and HBP

Handler Branch, branches to a specified handler.

This instruction can optionally store a return address to the lr, pass a parameter to the handler, or both.

Syntax

HB{L} #HandlerID

HB{L}P #immed, #HandlerID

where:

L is an optional suffix. If L is present, the instruction saves a return address in the lr.

P is an optional suffix. If P is present, the instruction passes the value of immed to the handler in r8.

immed is an immediate value. If L is present, immed must be in the range 0-31, otherwise immed must be in the range 0-7.

HandlerID is the index number of the handler to be called. If P is present, HandlerID must be in the range 0-31, otherwise HandlerID must be in the range 0-255.

Architectures

These instructions are not available in ARM state.

These 16-bit ThumbEE instructions are only available in ThumbEE state, in ARMv7 with Thumb-2EE support.
4.13 Pseudo-instructions

The ARM assembler supports a number of pseudo-instructions that are translated into the appropriate combination of ARM, Thumb-2, or pre-Thumb-2 Thumb instructions at assembly time.

The pseudo-instructions are described in the following sections:

- **ADRL pseudo-instruction** on page 4-155
  Load a program-relative or register-relative address into a register (medium range, position independent)

- **MOV32 pseudo-instruction** on page 4-157
  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 4-159
  Load a register with a 32-bit constant value or an address (unlimited range, but not position independent). Available for all ARM architectures.

- **UND pseudo-instruction** on page 4-162
  Generate an architecturally undefined instruction. Available for all ARM architectures.
4.13.1 ADRL pseudo-instruction

Load a program-relative or register-relative address into a register. It is similar to the ADR 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 Thumb instructions for pre-Thumb-2 processors.

---

Syntax

ADRL\{cond\} Rd, label

where:

\textit{cond} is an optional condition code (see \textit{Conditional execution} on page 2-18).

\textit{Rd} is the register to load.

\textit{label} is a program-relative or register-relative expression. See \textit{Register-relative and program-relative expressions} on page 3-37 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 \textit{LDR pseudo-instruction} on page 4-159 for information on loading a wider range of addresses (see also \textit{Loading constants into registers} on page 2-25).

ADRL produces position-independent code, because the address is program-relative or register-relative.

If \textit{label} is program-relative, it must evaluate to an address in the same assembler area as the ADRL pseudo-instruction, see \textit{AREA} on page 7-70.

If you use ADRL to generate a target for a BX or BLX instruction, it is your responsibility to set the Thumb bit (bit 0) of the address if the target contains Thumb instructions.
Architectures and range

The available range depends on the instruction set in use:

**ARM**
- ±64KB to a byte or halfword-aligned address.
- ±256KB bytes to a word-aligned address.

**32-bit Thumb**
- ±1MB bytes to a byte, halfword, or word-aligned address.

**16-bit Thumb**
- `ADRL` is not available.

The given range is relative to a point four bytes (in Thumb code) or two words (in ARM code) after the address of the current instruction. More distant addresses can be in range if the alignment is 16-bytes or more relative to this point.
4.13.2 MOV32 pseudo-instruction

Load a register with either:
- a 32-bit constant value
- any address.

MOV32 always generates two 32-bit instructions, a MOV, MOVT pair. This allows you to load any 32-bit constant, or to access the whole 32-bit address space.

Syntax

\[
\text{MOV32}\{\text{cond}\}\ Rd, \ expr
\]

where:

- \text{cond} is an optional condition code (see Conditional execution on page 2-18).
- \text{Rd} is the register to be loaded. \text{Rd} must not be sp or pc.
- \text{expr} can be any one of the following:
  - \text{symbol} A label in this or another program area.
  - \#\text{constant} Any 32-bit constant.
  - \text{symbol} + \text{constant} A label plus a 32-bit constant.

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.

---

MOV32 sets the Thumb bit (bit 0) of the address if the label referenced is in Thumb code.

Architectures

This pseudo-instruction is available in ARMv6T2 and above in both ARM and Thumb.
Examples

MOV32 r3, #0xABCDEF12 ; loads 0xABCDEF12 into r3
MOV32 r1, Trigger+12 ; loads the address that is 12 bytes higher than
                      ; the address Trigger into r1
4.13.3 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 4-10 for information on the LDR instruction.

Syntax

\[
\text{LDR}\{\text{cond}\}\{.W\} \ Rt, =expr
\]

\[
\text{LDR}\{\text{cond}\}\{.W\} \ Rt, =\text{label\_expr}
\]

where:

\(\text{cond}\) is an optional condition code (see Conditional execution on page 2-18).

\(\text{.W}\) is an optional instruction width specifier.

\(\text{Rt}\) is the register to be loaded.

\(\text{expr}\) evaluates to a numeric constant (see Numeric constants on page 3-28):

• The assembler generates a \texttt{MOV} or \texttt{MVN} instruction, if the value of \(\text{expr}\) is within range.

• If the value of \(\text{expr}\) is not within range of a \texttt{MOV} or \texttt{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.

See Loading with LDR \(Rd, =\text{const}\) on page 2-30, for information on loading constants.

\(\text{label\_expr}\) is a program-relative or external expression of an address in the form of a label plus or minus a numeric constant (see Register-relative and program-relative expressions on page 3-37 for more information). The assembler places the value of \(\text{label\_expr}\) in a literal pool and generates a program-relative LDR instruction that loads the value from the literal pool.

If \(\text{label\_expr}\) is an external expression, or is not contained in the current section, the assembler places a linker relocation directive in the object file. The linker generates the address at link time.
If `label_expr` is a local label (see Local labels on page 3-31), the assembler places a linker relocation directive in the object file and generates a symbol for that local label. The address is generated at link time. If the local label references Thumb code, the Thumb bit (bit 0) of the address is set.

--- Note ---
In RVCT v2.2, the Thumb bit of the address was not set. If you have code that relies on this behavior, use the command line option --untyped_local_labels to force the assembler not to set the Thumb bit when referencing labels in Thumb code.

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 ±4KB (ARM, 32-bit Thumb-2) or in the range 0 to +1KB (16-bit Thumb-2, pre-Thumb2 Thumb). You are responsible for ensuring that there is a literal pool within range. See LTORG on page 7-17 for more information.

If the label referenced is in Thumb code, the LDR pseudo-instruction sets the Thumb bit (bit 0) of `label_expr`.

See Loading constants into registers on page 2-25 for a more detailed explanation of how to use LDR, and for more information on MOV and MVN.

LDR in Thumb code

You can use the `.W` width specifier to force LDR to generate a 32-bit instruction in Thumb code on ARMv6T2 and later processors. 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.
If the value of the constant is not known in the first pass of the assembler, LDR without .W generates a 16-bit instruction in Thumb code, even if that results in a 16-bit pc-relative load for a constant that could be generated in a 32-bit MOV or MVN instruction. However, if the constant is known in the first pass, and it can be generated using a 32-bit MOV or MVN instruction, the MOV or MVN instruction is used.

The LDR pseudo-instruction never generates a 16-bit flag-setting MOV instruction. Use the --diag_warning 1727 assembler command-line option to check when a 16-bit instruction could have been used.

See MOV32 pseudo-instruction on page 4-157 for generating constants or addresses without loading from a literal pool.

**Examples**

```
LDR   r3,=0xff0    ; loads 0xff0 into r3
    ; =>  MOV.W r3,#0xff0
LDR   r1,=0xfff    ; loads 0xfff into r1
    ; =>  LDR r1,[pc,offset_to_litpool]
    ;     ...  
    ;     litpool DCD 0xfff
LDR   r2,=place    ; loads the address of place into r2
    ; =>  LDR r2,[pc,offset_to_litpool]
    ;     ...  
    ;     litpool DCD place
```
4.13.4 UND pseudo-instruction

Generate an architecturally undefined instruction. An attempt to execute an undefined instruction causes the Undefined instruction exception. Architecturally undefined instructions are expected to remain undefined.

Syntax

UND{cond}{.W} {#expr}

where:

cond is an optional condition code (see Conditional execution on page 2-18). cond is not allowed on this pseudo-instruction in pre-Thumb-2 Thumb code.

.W is an optional instruction width specifier.

expr evaluates to a numeric constant. Table 4-8 shows the range and encoding of expr in the instruction, where Y shows the locations of the bits that encode for expr and V is the 4 bits that encode for the condition code. If expr is omitted, the value 0 is used.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Encoding</th>
<th>Number of bits for expr</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARM</td>
<td>0xV7FYYYYFY</td>
<td>16</td>
<td>0-65535</td>
</tr>
<tr>
<td>32-bit Thumb</td>
<td>0xF7FYAYFY</td>
<td>12</td>
<td>0-4095</td>
</tr>
<tr>
<td>16-bit Thumb</td>
<td>0xDEYY</td>
<td>8</td>
<td>0-255</td>
</tr>
</tbody>
</table>

UND in Thumb code

You can use the .W width specifier to force UND to generate a 32-bit instruction in Thumb code on ARMv6T2 and later processors. UND.W always generates a 32-bit instruction, even if expr is in the range 0-255.

Disassembly

The encodings that this pseudo-instruction produces disassemble to DCI.
Chapter 5
NEON and VFP Programming

This chapter describes the assembly programming of NEON™ and the VFP coprocessor:

- Instruction summary on page 5-2
- Architecture support for NEON and VFP on page 5-8
- The extension register bank on page 5-9
- Condition codes on page 5-12
- General information on page 5-14
- Instructions shared by NEON and VFP on page 5-20
- NEON logical and compare operations on page 5-28
- NEON general data processing instructions on page 5-36
- NEON shift instructions on page 5-48
- NEON general arithmetic instructions on page 5-54
- NEON multiply instructions on page 5-67
- NEON load / store element and structure instructions on page 5-72
- NEON and VFP pseudo-instructions on page 5-80
- NEON and VFP system registers on page 5-87
- Flush-to-zero mode on page 5-92
- VFP instructions on page 5-94
- VFP vector mode on page 5-104.
5.1 Instruction summary

This section provides a summary of the NEON and VFP instructions. Use it to locate individual instructions and pseudo-instructions described in the rest of this chapter. It contains:

- **NEON instructions**
- **Shared NEON and VFP instructions** on page 5-6
- **VFP instructions** on page 5-7.

5.1.1 NEON instructions

Table 5-1 shows a summary of NEON instructions. These instructions are not available in VFP.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Brief description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>VABA, VABD</td>
<td>Absolute difference, Absolute difference and Accumulate</td>
<td>page 5-55</td>
</tr>
<tr>
<td>VABS</td>
<td>Absolute value</td>
<td>page 5-56</td>
</tr>
<tr>
<td>VACGE, VACGT</td>
<td>Absolute Compare Greater than or Equal, Greater Than</td>
<td>page 5-33</td>
</tr>
<tr>
<td>VACLE, VACLT</td>
<td>Absolute Compare Less than or Equal, Less than (pseudo-instructions)</td>
<td>page 5-85</td>
</tr>
<tr>
<td>VADD</td>
<td>Add</td>
<td>page 5-57</td>
</tr>
<tr>
<td>VADDPH</td>
<td>Add, select High half</td>
<td>page 5-58</td>
</tr>
<tr>
<td>VAND</td>
<td>Bitwise AND</td>
<td>page 5-29</td>
</tr>
<tr>
<td>VAND</td>
<td>Bitwise AND (pseudo-instruction)</td>
<td>page 5-84</td>
</tr>
<tr>
<td>VBIC</td>
<td>Bitwise Bit Clear (register)</td>
<td>page 5-29</td>
</tr>
<tr>
<td>VBIC</td>
<td>Bitwise Bit Clear (immediate)</td>
<td>page 5-30</td>
</tr>
<tr>
<td>VBIF, VBIT, VBSL</td>
<td>Bitwise Insert if False, Insert if True, Select</td>
<td>page 5-31</td>
</tr>
<tr>
<td>VCCEQ, VCLE, VCLT</td>
<td>Compare Equal, Less than or Equal, Compare Less Than</td>
<td>page 5-34</td>
</tr>
<tr>
<td>VCGE, VCGT</td>
<td>Compare Greater than or Equal, Greater Than</td>
<td>page 5-34</td>
</tr>
<tr>
<td>VCLE, VCLT</td>
<td>Compare Less than or Equal, Compare Less Than (pseudo-instruction)</td>
<td>page 5-86</td>
</tr>
<tr>
<td>VCLS, VCLZ, VCNT</td>
<td>Count Leading Sign bits, Count Leading Zeros, and Count set bits</td>
<td>page 5-63</td>
</tr>
<tr>
<td>VCVT</td>
<td>Convert fixed-point or integer to floating point, floating-point to integer or fixed-point</td>
<td>page 5-37</td>
</tr>
<tr>
<td>Mnemonic</td>
<td>Brief description</td>
<td>Page</td>
</tr>
<tr>
<td>----------</td>
<td>-----------------------------------------------------------------------------------</td>
<td>--------</td>
</tr>
<tr>
<td>VCVT</td>
<td>Convert between half-precision and single-precision floating-point numbers</td>
<td>5-38</td>
</tr>
<tr>
<td>VDUP</td>
<td>Duplicate scalar to all lanes of vector</td>
<td>5-39</td>
</tr>
<tr>
<td>VEOR</td>
<td>Bitwise Exclusive OR</td>
<td>5-29</td>
</tr>
<tr>
<td>VEXT</td>
<td>Extract</td>
<td>5-40</td>
</tr>
<tr>
<td>VHADD, VHSUB</td>
<td>Halving Add, Halving Subtract</td>
<td>5-59</td>
</tr>
<tr>
<td>VLD</td>
<td>Vector Load</td>
<td>5-72</td>
</tr>
<tr>
<td>VMAX, VMIN</td>
<td>Maximum, Minimum</td>
<td>5-62</td>
</tr>
<tr>
<td>VMLA, VMLS</td>
<td>Multiply Accumulate, Multiply Subtract (vector)</td>
<td>5-68</td>
</tr>
<tr>
<td>VMLA, VMLS</td>
<td>Multiply Accumulate, Multiply Subtract (by scalar)</td>
<td>5-69</td>
</tr>
<tr>
<td>VMOV</td>
<td>Move (immediate)</td>
<td>5-41</td>
</tr>
<tr>
<td>VMOV</td>
<td>Move (register)</td>
<td>5-32</td>
</tr>
<tr>
<td>VMOVl, VMOV{U}N</td>
<td>Move Long, Move Narrow (register)</td>
<td>5-42</td>
</tr>
<tr>
<td>VMUL</td>
<td>Multiply (vector)</td>
<td>5-68</td>
</tr>
<tr>
<td>VMUL</td>
<td>Multiply (by scalar)</td>
<td>5-69</td>
</tr>
<tr>
<td>VMVN</td>
<td>Move Negative (immediate)</td>
<td>5-41</td>
</tr>
<tr>
<td>VNEG</td>
<td>Negate</td>
<td>5-56</td>
</tr>
<tr>
<td>VORN</td>
<td>Bitwise OR NOT</td>
<td>5-29</td>
</tr>
<tr>
<td>VORN</td>
<td>Bitwise OR NOT (pseudo-instruction)</td>
<td>5-84</td>
</tr>
<tr>
<td>VORR</td>
<td>Bitwise OR (register)</td>
<td>5-29</td>
</tr>
<tr>
<td>VORR</td>
<td>Bitwise OR (immediate)</td>
<td>5-30</td>
</tr>
<tr>
<td>VPADD, VPADAL</td>
<td>Pairwise Add, Pairwise Add and Accumulate</td>
<td>5-60</td>
</tr>
<tr>
<td>VPMAX, VPMIN</td>
<td>Pairwise Maximum, Pairwise Minimum</td>
<td>5-62</td>
</tr>
<tr>
<td>VQABS</td>
<td>Absolute value, saturate</td>
<td>5-56</td>
</tr>
<tr>
<td>VQADD</td>
<td>Add, saturate</td>
<td>5-57</td>
</tr>
<tr>
<td>VQDM{L}, VQD{M}LS</td>
<td>Saturating Doubling Multiply Accumulate, and Multiply Subtract</td>
<td>5-70</td>
</tr>
</tbody>
</table>
Table 5-1 Location of NEON instructions (continued)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Brief description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>VQDMUL</td>
<td>Saturating Doubling Multiply</td>
<td>page 5-70</td>
</tr>
<tr>
<td>VQDMULH</td>
<td>Saturating Doubling Multiply returning High half</td>
<td>page 5-71</td>
</tr>
<tr>
<td>VQMOV(U)N</td>
<td>Saturating Move (register)</td>
<td>page 5-42</td>
</tr>
<tr>
<td>VQNEG</td>
<td>Negate, saturate</td>
<td>page 5-56</td>
</tr>
<tr>
<td>VQRDMDUH</td>
<td>Saturating Doubling Multiply returning High half</td>
<td>page 5-71</td>
</tr>
<tr>
<td>VQRSHL</td>
<td>Shift Left, Round, saturate (by signed variable)</td>
<td>page 5-50</td>
</tr>
<tr>
<td>VQRSHR</td>
<td>Shift Right, Round, saturate (by immediate)</td>
<td>page 5-52</td>
</tr>
<tr>
<td>VQSHL</td>
<td>Shift Left, saturate (by immediate)</td>
<td>page 5-49</td>
</tr>
<tr>
<td>VQSHL</td>
<td>Shift Left, saturate (by signed variable)</td>
<td>page 5-50</td>
</tr>
<tr>
<td>VQSHR</td>
<td>Shift Right, saturate (by immediate)</td>
<td>page 5-52</td>
</tr>
<tr>
<td>VQSUB</td>
<td>Subtract, saturate</td>
<td>page 5-57</td>
</tr>
<tr>
<td>VRADDH</td>
<td>Add, select High half, Round</td>
<td>page 5-58</td>
</tr>
<tr>
<td>VRECPE</td>
<td>Reciprocal Estimate</td>
<td>page 5-64</td>
</tr>
<tr>
<td>VRECPS</td>
<td>Reciprocal Step</td>
<td>page 5-65</td>
</tr>
<tr>
<td>VREV</td>
<td>Reverse elements</td>
<td>page 5-43</td>
</tr>
<tr>
<td>VRHADD</td>
<td>Halving Add, Round</td>
<td>page 5-59</td>
</tr>
<tr>
<td>VRSRHR, VRSRA</td>
<td>Shift Right and Round, Shift Right, Round, and Accumulate (by immediate)</td>
<td>page 5-51</td>
</tr>
<tr>
<td>VRSQRTES</td>
<td>Reciprocal Square Root Estimate</td>
<td>page 5-64</td>
</tr>
<tr>
<td>VRSQRTS</td>
<td>Reciprocal Square Root Step</td>
<td>page 5-65</td>
</tr>
<tr>
<td>VRSUBH</td>
<td>Subtract, select High half, Round</td>
<td>page 5-58</td>
</tr>
<tr>
<td>VSHL</td>
<td>Shift Left (by immediate)</td>
<td>page 5-49</td>
</tr>
<tr>
<td>VSHR</td>
<td>Shift Right (by immediate)</td>
<td>page 5-51</td>
</tr>
<tr>
<td>VSLI</td>
<td>Shift Left and Insert</td>
<td>page 5-53</td>
</tr>
<tr>
<td>VSRA</td>
<td>Shift Right, Accumulate (by immediate)</td>
<td>page 5-51</td>
</tr>
<tr>
<td>VSRI</td>
<td>Shift Right and Insert</td>
<td>page 5-53</td>
</tr>
</tbody>
</table>
### Table 5-1 Location of NEON instructions (continued)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Brief description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>VST</td>
<td>Vector Store</td>
<td>page 5-72</td>
</tr>
<tr>
<td>VSUB</td>
<td>Subtract</td>
<td>page 5-57</td>
</tr>
<tr>
<td>VSUBH</td>
<td>Subtract, select High half</td>
<td>page 5-58</td>
</tr>
<tr>
<td>VSWP</td>
<td>Swap vectors</td>
<td>page 5-44</td>
</tr>
<tr>
<td>VTBL, VTBX</td>
<td>Vector table look-up</td>
<td>page 5-45</td>
</tr>
<tr>
<td>VTRN</td>
<td>Vector transpose</td>
<td>page 5-46</td>
</tr>
<tr>
<td>VTST</td>
<td>Test bits</td>
<td>page 5-35</td>
</tr>
<tr>
<td>VUZP, VZIP</td>
<td>Vector interleave and de-interleave</td>
<td>page 5-47</td>
</tr>
</tbody>
</table>
5.1.2 Shared NEON and VFP instructions

Table 5-2 shows a summary of instructions that are common to NEON and VFP.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>VLDM</td>
<td>Load multiple</td>
<td>page 5-22</td>
<td>-</td>
<td>All</td>
</tr>
<tr>
<td>VLDR</td>
<td>Load (see also VLDR pseudo-instruction on page 5-81)</td>
<td>page 5-21</td>
<td>Scalar</td>
<td>All</td>
</tr>
<tr>
<td></td>
<td>Load (post-increment and pre-decrement)</td>
<td>page 5-82</td>
<td>Scalar</td>
<td>All</td>
</tr>
<tr>
<td>VMOV</td>
<td>Transfer from one ARM* register to half of a doubleword register</td>
<td>page 5-25</td>
<td>Scalar</td>
<td>All</td>
</tr>
<tr>
<td></td>
<td>Transfer from two ARM registers to a doubleword register</td>
<td>page 5-24</td>
<td>Scalar</td>
<td>VFPv2</td>
</tr>
<tr>
<td></td>
<td>Transfer from half of a doubleword register to ARM register</td>
<td>page 5-25</td>
<td>Scalar</td>
<td>All</td>
</tr>
<tr>
<td></td>
<td>Transfer from a doubleword register to two ARM registers</td>
<td>page 5-24</td>
<td>Scalar</td>
<td>VFPv2</td>
</tr>
<tr>
<td></td>
<td>Transfer from single-precision to ARM register</td>
<td>page 5-26</td>
<td>Scalar</td>
<td>All</td>
</tr>
<tr>
<td></td>
<td>Transfer from ARM register to single-precision</td>
<td>page 5-26</td>
<td>Scalar</td>
<td>All</td>
</tr>
<tr>
<td>VMRS</td>
<td>Transfer from NEON and VFP system register to ARM register</td>
<td>page 5-27</td>
<td>-</td>
<td>All</td>
</tr>
<tr>
<td>VMSR</td>
<td>Transfer from ARM register to NEON and VFP system register</td>
<td>page 5-27</td>
<td>-</td>
<td>All</td>
</tr>
<tr>
<td>VPOP</td>
<td>Pop VFP or NEON registers from full-descending stack</td>
<td>page 5-22</td>
<td>-</td>
<td>All</td>
</tr>
<tr>
<td>VPUSH</td>
<td>Push VFP or NEON registers to full-descending stack</td>
<td>page 5-22</td>
<td>-</td>
<td>All</td>
</tr>
<tr>
<td>VSTM</td>
<td>Store multiple</td>
<td>page 5-22</td>
<td>-</td>
<td>All</td>
</tr>
<tr>
<td>VSTR</td>
<td>Store</td>
<td>page 5-21</td>
<td>Scalar</td>
<td>All</td>
</tr>
<tr>
<td></td>
<td>Store (post-increment and pre-decrement)</td>
<td>page 5-82</td>
<td>Scalar</td>
<td>All</td>
</tr>
</tbody>
</table>
### 5.1.3 VFP instructions

Table 5-3 shows a summary of VFP instructions that are not available in NEON.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>VABS</td>
<td>Absolute value</td>
<td>page 5-95</td>
<td>Vector</td>
<td>All</td>
</tr>
<tr>
<td>VADD</td>
<td>Add</td>
<td>page 5-96</td>
<td>Vector</td>
<td>All</td>
</tr>
<tr>
<td>V CMP</td>
<td>Compare</td>
<td>page 5-98</td>
<td>Scalar</td>
<td>All</td>
</tr>
<tr>
<td>VC VT</td>
<td>Convert between single-precision and double-precision</td>
<td>page 5-99</td>
<td>Scalar</td>
<td>All</td>
</tr>
<tr>
<td>VC VT</td>
<td>Convert between floating-point and integer</td>
<td>page 5-100</td>
<td>Scalar</td>
<td>All</td>
</tr>
<tr>
<td>VC VT</td>
<td>Convert between floating-point and fixed-point</td>
<td>page 5-101</td>
<td>Scalar</td>
<td>VFPv3</td>
</tr>
<tr>
<td>VC VT B, VC VT T</td>
<td>Convert between half-precision and single-precision floating-point</td>
<td>page 5-102</td>
<td>Scalar</td>
<td>Half-precision</td>
</tr>
<tr>
<td>V DIV</td>
<td>Divide</td>
<td>page 5-96</td>
<td>Vector</td>
<td>All</td>
</tr>
<tr>
<td>V M LA</td>
<td>Multiply accumulate</td>
<td>page 5-97</td>
<td>Vector</td>
<td>All</td>
</tr>
<tr>
<td>V M LS</td>
<td>Multiply subtract</td>
<td>page 5-97</td>
<td>Vector</td>
<td>All</td>
</tr>
<tr>
<td>V M O V</td>
<td>Insert floating-point constant in single-precision or double-precision register (see also Table 5-2 on page 5-6)</td>
<td>page 5-103</td>
<td>Scalar</td>
<td>VFPv3</td>
</tr>
<tr>
<td>V M U L</td>
<td>Multiply</td>
<td>page 5-97</td>
<td>Vector</td>
<td>All</td>
</tr>
<tr>
<td>V N E G</td>
<td>Negate</td>
<td>page 5-95</td>
<td>Vector</td>
<td>All</td>
</tr>
<tr>
<td>V N M LA</td>
<td>Negated multiply accumulate</td>
<td>page 5-97</td>
<td>Vector</td>
<td>All</td>
</tr>
<tr>
<td>V N M LS</td>
<td>Negated multiply subtract</td>
<td>page 5-97</td>
<td>Vector</td>
<td>All</td>
</tr>
<tr>
<td>V N M U L</td>
<td>Negated multiply</td>
<td>page 5-97</td>
<td>Vector</td>
<td>All</td>
</tr>
<tr>
<td>V S Q R T</td>
<td>Square Root</td>
<td>page 5-95</td>
<td>Vector</td>
<td>All</td>
</tr>
<tr>
<td>V S U B</td>
<td>Subtract</td>
<td>page 5-96</td>
<td>Vector</td>
<td>All</td>
</tr>
</tbody>
</table>
5.2 Architecture support for NEON and VFP

The NEON extension is optionally available only for the ARMv7-A and ARMv7-R architectures. All NEON instructions, with the exception of half-precision instructions, are available on systems that support NEON. Some of these instructions are also available on systems that implement VFP extension without NEON. These are called shared instructions.

The half-precision instructions are only available on NEON or VFPv3 systems that implement the half-precision extension (see Half-precision extension).

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

ARMv7-M does not support VFP. All other ARMv7 architecture profiles support the VFPv3 architecture.

VFPv3 has variants that do not support all VFPv3 registers and floating-point data types. For details of the implemented VFP architecture and variant, you must always refer to the appropriate product documentation.

NEON and VFP instructions, including the half-precision instructions, are treated as Undefined Instructions on systems that do not support the necessary architecture extension. Even on systems that support NEON and VFP, the instructions are undefined if the necessary coprocessors are not enabled in the Coprocessor Access Control Register (CP15 CPACR). For more information, see the Technical Reference Manual for your processor.

5.2.1 Half-precision extension

The Half-precision extension is an optional architecture that extends both the VFPv3 and the NEON architectures. It provides VFP and NEON instructions that perform conversion between single-precision (32-bit) and half-precision (16-bit) floating-point numbers.
5.3 The extension register bank

NEON and VFP use the same extension register bank. This is distinct from the ARM register bank.

The VFP coprocessor has 32 single-precision registers, each of which 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. However, some VFPv3 variants add 16 further double-precision registers to the VFP register set. These do not overlap with any single-precision VFP registers.

The extension register bank can be referenced using three explicitly aliased views, as described in the following sections.

Figure 5-1 on page 5-11 shows the three views of the extension register bank, and the way the word, doubleword, and quadword registers overlap.

--- Note ---
If your processor has both NEON and VFP, all the NEON registers overlap with the VFP registers.

---
You can use:

- some registers for single-precision values at the same time as you are using others for double-precision values, and others as NEON vectors

- the same registers for single-precision values, double-precision values, and NEON vectors, 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.

5.3.1 NEON views of the register bank

In NEON, you can view the extension register bank as:

- Sixteen 128-bit quadword registers, Q0-Q15.

- Thirty-two 64-bit doubleword registers, D0-D31.

- A combination of registers from the above views.

NEON views each register as containing a vector of 1, 2, 4, 8, or 16 elements, all of the same size and type. Individual elements can also be accessed as scalars.
5.3.2 VFP views of the extension register bank

In VFPv3 and VFPv3_fp16, you can view the extension register bank as:

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

In VFPv2, VFPv3-D16, and VFPv3-D16_fp16, you can view the extension register bank as:

- Sixteen 64-bit doubleword registers, D0-D15.
- Thirty-two 32-bit single word registers, S0-S31.
- A combination of registers from the above views.
The mapping between the registers is as follows:

- $S_{2n}$ maps to the least significant half of $D_{n}$
- $S_{2n+1}$ maps to the most significant half of $D_{n}$
- $D_{2n}$ maps to the least significant half of $Q_{n}$
- $D_{2n+1}$ maps to the most significant half of $Q_{n}$.

For example, you can access the least significant half of the elements of a vector in $Q_6$ by referring to $D_{12}$, and the most significant half of the elements by referring to $D_{13}$. 
5.4 Condition codes

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

In ARM state, except for the instructions that are common to both VFP and NEON, you cannot use a condition code to control the execution of NEON instructions.

In Thumb® state on a Thumb-2 processor, you can use an IT instruction to set condition codes on up to four following NEON or VFP instructions. See IT on page 4-118 for details.

The only VFP instruction that can be used to update the status flags is VCMP. It does not update the flags in the APSR directly, but updates a separate set of flags in the FPSCR (see FPSCR, the floating-point status and control register on page 5-87).

Note

To use these flags to control conditional instructions, including conditional VFP instructions, you must first copy them into the APSR using a VMRS instruction (see VMRS and VMSR on page 5-27).

Following an VCMP 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 5-4 on page 5-13.
### Table 5-4 Condition codes

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Meaning after ARM data processing instruction</th>
<th>Meaning after VFP VCMP instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>EQ</td>
<td>Equal</td>
<td>Equal</td>
</tr>
<tr>
<td>NE</td>
<td>Not equal</td>
<td>Not equal, or unordered</td>
</tr>
<tr>
<td>CS or HS</td>
<td>Carry set or Unsigned higher or same</td>
<td>Greater than or equal, or unordered</td>
</tr>
<tr>
<td>CC or LO</td>
<td>Carry clear or Unsigned lower</td>
<td>Less than</td>
</tr>
<tr>
<td>MI</td>
<td>Negative</td>
<td>Less than</td>
</tr>
<tr>
<td>PL</td>
<td>Positive or zero</td>
<td>Greater than or equal, or unordered</td>
</tr>
<tr>
<td>VS</td>
<td>Overflow</td>
<td>Unordered (at least one NaN operand)</td>
</tr>
<tr>
<td>VC</td>
<td>No overflow</td>
<td>Not unordered</td>
</tr>
<tr>
<td>HI</td>
<td>Unsigned higher</td>
<td>Greater than, or unordered</td>
</tr>
<tr>
<td>LS</td>
<td>Unsigned lower or same</td>
<td>Less than or equal</td>
</tr>
<tr>
<td>GE</td>
<td>Signed greater than or equal</td>
<td>Greater than or equal</td>
</tr>
<tr>
<td>LT</td>
<td>Signed less than</td>
<td>Less than, or unordered</td>
</tr>
<tr>
<td>GT</td>
<td>Signed greater than</td>
<td>Greater than</td>
</tr>
<tr>
<td>LE</td>
<td>Signed less than or equal</td>
<td>Less than or equal, or unordered</td>
</tr>
<tr>
<td>AL</td>
<td>Always (normally omitted)</td>
<td>Always (normally omitted)</td>
</tr>
</tbody>
</table>

--- **Note**

The type of the instruction that last updated the flags in the APSR determines the meaning of condition codes.
5.5 General information

Certain information common to many instructions is presented here, to avoid
duplication. This section contains the following subsections:

- Floating-point exceptions
- NEON and VFP data types
- Normal, Long, Wide, Narrow, and saturating instructions in NEON on page 5-15
- NEON Scalars on page 5-17
- Extended notation on page 5-18
- Polynomial arithmetic over \{0,1\} on page 5-18
- The VFP coprocessor on page 5-19.

5.5.1 Floating-point exceptions

In the descriptions of those instructions that can cause floating-point exceptions, there
is a subsection listing the exceptions. If there is no Floating-point exceptions subsection
in an instruction description, that instruction cannot cause any floating-point exception.

5.5.2 NEON and VFP data types

Data type specifiers in NEON and VFP instructions consist of a letter indicating the type
of data, usually followed by a number indicating the width. They are separated from the
instruction mnemonic by a point. Table 5-5 shows the data types available in NEON
instructions. Table 5-6 on page 5-15 shows the data types available in VFP instructions.

<table>
<thead>
<tr>
<th>Table 5-5 NEON data types</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>Unsigned integer</td>
</tr>
<tr>
<td>Signed integer</td>
</tr>
<tr>
<td>Integer of unspecified type</td>
</tr>
<tr>
<td>Floating-point number</td>
</tr>
<tr>
<td>Polynomial over {0,1}</td>
</tr>
</tbody>
</table>
The datatype of the second (or only) operand is specified in the instruction.

--- Note ---
- Most instructions have a restricted range of permitted data types. See the instruction pages for details. However, the data type description is flexible:
  - If the description specifies I, you can also use S or U data types
  - If only the data size is specified, you can specify a type (I, S, U, P or F)
  - If no data type is specified, you can specify a data type.
- The F16 data type is only available on systems that implement the half-precision architecture extension.

## 5.5.3 Normal, Long, Wide, Narrow, and saturating instructions in NEON

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

NEON instructions can operate on:

- **Doubleword vectors consisting of:**
  - eight 8-bit elements
  - four 16-bit elements
  - two 32-bit elements
  - one 64-bit element.

- **Quadword vectors consisting of:**
  - sixteen 8-bit elements
  - eight 16-bit elements
  - four 32-bit elements
  - two 64-bit elements.
**Normal instructions**

Normal instructions can operate on any of these vector types, and produce result vectors the same size, and usually the same type, as the operand vectors.

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

**Long instructions**

Long instructions operate on Doubleword vector operands and produce a Quadword vector result. The elements of the result are usually twice the width of the elements of the operands, and the same type.

Long instructions are specified using an L appended to the instruction mnemonic.

**Wide instructions**

Wide instructions operate on one Doubleword vector operand and one Quadword vector operand. They produce a Quadword vector result. The elements of the result and the first operand are twice the width of the elements of the second operand.

Wide instructions are specified using a W appended to the instruction mnemonic.

**Narrow instructions**

Narrow instructions operate on Quadword vector operands, and produce a Doubleword vector result. The elements of the result are usually half the width of the elements of the operands.

Narrow instructions are specified using an N appended to the instruction mnemonic.
Saturating instructions

For a general description of what saturating instructions do, see Saturating instructions on page 4-93. See Table 5-7 for the ranges that NEON saturating instructions saturate to.

Saturating instructions are specified using a Q prefix between the V and the instruction mnemonic.

Table 5-7 NEON saturation ranges

<table>
<thead>
<tr>
<th>Data type</th>
<th>Saturation range of x</th>
</tr>
</thead>
<tbody>
<tr>
<td>S8</td>
<td>(-2^7 &lt;= x &lt; 2^7)</td>
</tr>
<tr>
<td>S16</td>
<td>(-2^{15} &lt;= x &lt; 2^{15})</td>
</tr>
<tr>
<td>S32</td>
<td>(-2^{31} &lt;= x &lt; 2^{31})</td>
</tr>
<tr>
<td>S64</td>
<td>(-2^{63} &lt;= x &lt; 2^{63})</td>
</tr>
<tr>
<td>U8</td>
<td>0 &lt;= x &lt; 2^8</td>
</tr>
<tr>
<td>U16</td>
<td>0 &lt;= x &lt; 2^{16}</td>
</tr>
<tr>
<td>U32</td>
<td>0 &lt;= x &lt; 2^{32}</td>
</tr>
<tr>
<td>U64</td>
<td>0 &lt;= x &lt; 2^{64}</td>
</tr>
</tbody>
</table>

5.5.4 NEON Scalars

Some NEON instructions act on scalars in combination with vectors. NEON scalars can be 8-bit, 16-bit, 32-bit, or 64-bit. Other than multiply instructions, instructions that access scalars can access any element in the register bank. The instruction syntax refers to the scalars using an index into a doubleword vector, so that \(Dm[x]\) is the \(x\)th element in \(Dm\).

Multiply instructions only allow 16-bit or 32-bit scalars, and can only access the first 32 scalars in the register bank. That is, in multiply instructions:

- 16-bit scalars are restricted to registers D0-D7, with \(x\) in the range 0-3
- 32-bit scalars are restricted to registers D0-D15, with \(x\) either 0 or 1.
5.5.5 Extended notation

The assembler implements an extension to the architectural NEON and VFP assembly syntax, called extended notation. This extension allows you to include datatype information or scalar indexes in register names. If you do this, you do not need to include the datatype or scalar index information in every instruction.

Register names can be any of the following:

** Untyped **

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

** Untyped with scalar index **

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

** Typed **

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

** Typed with scalar index **

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

Use the SN, DN, and QN directives to create typed and scalar registers. See QN, DN, and SN on page 7-13 for details.

5.5.6 Polynomial arithmetic over \( \{0,1\} \)

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

- \( 0 + 0 = 1 + 1 = 0 \)
- \( 0 + 1 = 1 + 0 = 1 \)
- \( 0 \times 0 = 0 \times 1 = 1 \times 0 = 0 \)
- \( 1 \times 1 = 1 \).

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

The 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. See Chapter 4 *Floating-point Support* in the *Libraries Guide* for more information.

You can use short vectors of up to eight single-precision or four double-precision numbers, but this is deprecated. See *VFP vector mode* on page 5-104 for further information.
5.6 Instructions shared by NEON and VFP

This section contains the following subsections:

- **VLDR and VSTR** on page 5-21
  Extension register load and store.

- **VLDM, VSTM, VPOP, and VPUSH** on page 5-22
  Extension register load and store multiple.

- **VMOV (between two ARM registers and an extension register)** on page 5-24
  Transfer contents between two ARM registers and a 64-bit extension register.

- **VMOV (between an ARM register and a NEON scalar)** on page 5-25
  Transfer contents between an ARM register and a half of a 64-bit extension register.

- **VMOV (between one ARM register and single precision VFP)** on page 5-26
  Transfer contents between a 32-bit extension register and an ARM register.

- **VMRS and VMSR** on page 5-27
  Transfer contents between an ARM register and a NEON and VFP system register.
5.6.1 VLDR and VSTR

Extension register load and store.

**Syntax**

\[
\begin{align*}
\text{VLDR}\{\text{cond}\}\{.\text{size}\} \ Fd, [\text{Rn}, \#\text{offset}] \\
\text{VSTR}\{\text{cond}\}\{.\text{size}\} \ Fd, [\text{Rn}, \#\text{offset}] \\
\text{VLDR}\{\text{cond}\}\{.\text{size}\} \ Fd, \text{label} \\
\text{VSTR}\{\text{cond}\}\{.\text{size}\} \ Fd, \text{label}
\end{align*}
\]

where:

- `cond` is an optional condition code (see *Condition codes* on page 5-12).
- `size` is an optional data size specifier. Must be 32 if `Fd` is a single precision VFP register, or 64 otherwise.
- `Fd` is the extension register to be loaded or saved. For a NEON instruction, it must be a D register. For a VFP instruction, it can be either a D or S register.
- `Rn` is the ARM register holding the base address for the transfer.
- `offset` is an optional numeric expression. It must evaluate to a numeric 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 3-37 for more information. `label` must be within ±1KB of the current instruction.

**Usage**

The VLDR instruction loads an extension register from memory. The VSTR instruction saves the contents of an extension register to memory.

One word is transferred if `Fd` is a single precision register (VFP only). Two words are transferred otherwise.

There is also an VLDR pseudo-instruction (see *VLDR pseudo-instruction* on page 5-81).
5.6.2 VLDM, VSTM, VPOP, and VPUSH

Extension register load multiple, store multiple, pop from stack, push onto stack.

Syntax

VLDM{mode}{cond} Rn[!], Registers
VSTM{mode}{cond} Rn[!], Registers
VPOP{cond} Registers
VPUSH{cond} Registers

where:

mode must be one of:

IA  meaning Increment address After each transfer. IA is the default, and can be omitted.
DB  meaning Decrement address Before each transfer.
EA  meaning Empty Ascending stack operation. This is the same as DB for loads, 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.

See Table 2-9 on page 2-41 for equivalent addressing mode suffixes.

cond is an optional condition code (see Condition codes on page 5-12).

Rn is the ARM register holding the base address for the transfer.

! is optional. ! specifies that the updated base address must be written back to Rn. If ! is not specified, mode must be IA.

Registers is a list of consecutive extension registers enclosed in braces, { and }. The list can be comma-separated, or in range format. There must be at least one register in the list.

You can specify S, D, or Q registers, but they must not be mixed. The number of registers must not exceed 16 D registers, or 8 Q registers. If Q registers are specified, on disassembly they are shown as D registers.
--- Note ---

VPOP Registers is equivalent to VLDM sp!, Registers.

VPUSH Registers is equivalent to VSTMDB sp!, Registers.

You can use either form of these instructions. They disassemble to VPOP and VPUSH.
5.6.3 VMOV (between two ARM registers and an extension register)

Transfer contents between two ARM registers and a 64-bit extension register, or two consecutive 32-bit VFP registers.

Syntax

\[
\begin{align*}
\text{VMOV}\{\text{cond}\} Dm, Rd, Rn \\
\text{VMOV}\{\text{cond}\} Rd, Rn, Dm \\
\text{VMOV}\{\text{cond}\} Sm, Sm1, Rd, Rn \\
\text{VMOV}\{\text{cond}\} Rd, Rn, Sm, Sm1
\end{align*}
\]

where:
- \(\text{cond}\) is an optional condition code (see Condition codes on page 5-12).
- \(Dm\) is a 64-bit extension register.
- \(Sm\) is a VFP 32-bit register.
- \(Sm1\) is the next consecutive VFP 32-bit register after \(Sm\).
- \(Rd, Rn\) are the ARM registers. Do not use r15.

Usage

\[
\begin{align*}
\text{VMOV} Dm, Rd, Rn & \text{ transfers the contents of } Rd \text{ into the low half of } Dm, \text{ and the contents of } Rn \text{ into the high half of } Dm. \\
\text{VMOV} Rd, Rn, Dm & \text{ transfers the contents of the low half of } Dm \text{ into } Rd, \text{ and the contents of the high half of } Dm \text{ into } Rn. \\
\text{VMOV} Rd, Rn, Sm, Sm1 & \text{ transfers the contents of } Sm \text{ into } Rd, \text{ and the contents of } Sm1 \text{ into } Rn. \\
\text{VMOV} Sm, Sm1, Rd, Rn & \text{ transfers the contents of } Rd \text{ into } Sm, \text{ and the contents of } Rn \text{ into } Sm1.
\end{align*}
\]

Architectures

The 64-bit instructions are available in:
- NEON
- VFPv2 and above.

The 2 x 32-bit instructions are available in VFPv2 and above.
5.6.4 VMOV (between an ARM register and a NEON scalar)

Transfer contents between an ARM register and a NEON scalar.

Syntax

\[
\text{VMOV}\{\text{cond}\}\{.size\} \ Dn[x], \ Rd
\]

\[
\text{VMOV}\{\text{cond}\}\{.datatype\} \ Rd, \ Dn[x]
\]

where:

- \textit{cond} is an optional condition code (see \textit{Condition codes} on page 5-12).
- \textit{size} the data size. Can be 8, 16, or 32. If omitted, \textit{size} is 32. For VFP instructions, \textit{size} must be 32 or omitted.
- \textit{datatype} the data type. Can be U8, S8, U16, S16, or 32. If omitted, \textit{datatype} is 32. For VFP instructions, \textit{datatype} must be 32 or omitted.
- \textit{Dn[x]} is the NEON scalar (see \textit{NEON Scalars} on page 5-17).
- \textit{Rd} is the ARM register. \textit{Rd} must not be r15.

Usage

\textit{VMOV} \ Rd, \ Dn[x] transfers the contents of \textit{Dn[x]} into the least significant byte, halfword, or word of \textit{Rd}. The remaining bits of \textit{Rd} is either zero or sign extended.

\textit{VMOV} \ Dn[x], \ Rd transfers the contents of the least significant byte, halfword, or word of \textit{Rd} into \textit{Sn}. 

5.6.5 VMOV (between one ARM register and single precision VFP)

Transfer contents between a single-precision floating-point register and an ARM register.

Syntax

\[
\begin{align*}
\text{VMOV}\{\text{cond}\} & \quad Rd, \quad Sn \\
\text{VMOV}\{\text{cond}\} & \quad Sn, \quad Rd
\end{align*}
\]

where:

\text{cond} is an optional condition code (see \textit{Condition codes} on page 5-12).

\text{Sn} is the VFP single-precision register.

\text{Rd} is the ARM register. \text{Rd} must not be r15.

Usage

\text{VMOV} \; Rd, \; Sn \text{ transfers the contents of } Sn \text{ into } Rd.

\text{VMOV} \; Sn, \; Rd \text{ transfers the contents of } Rd \text{ into } Sn.
5.6.6 VMRS and VMSR

Transfer contents between an ARM register and a NEON and VFP system register.

Syntax

\[
\text{VMRS}\{\text{cond}\} \ Rd, \ extsysreg \\
\text{VMSR}\{\text{cond}\} \ extsysreg, \ Rd \\
\]

where:

- \( \text{cond} \) is an optional condition code (see \textit{Condition codes} on page 5-12).
- \( \text{extsysreg} \) is the NEON and VFP system register, usually \texttt{FPSCR}, \texttt{FPSID}, or \texttt{FPEXC} (see \textit{NEON and VFP system registers} on page 5-87).
- \( \text{Rd} \) is the ARM register. \( \text{Rd} \) must not be \texttt{r15}.

It can be \texttt{APSR\_nzcv}, if \( \text{extsysreg} \) is \texttt{FPSCR}. In this case, the floating-point status flags are transferred into the corresponding flags in the ARM APSR.

Usage

The VMRS instruction transfers the contents of \( \text{extsysreg} \) into \( \text{Rd} \).

The VMSR instruction transfers the contents of \( \text{Rd} \) into \( \text{extsysreg} \).

\[\text{Note}\]

These instructions stall the ARM until all current NEON or VFP operations complete.

Examples

\[
\begin{align*}
\text{VMRS} & \quad r2, \text{FPCID} \\
\text{VMRS} & \quad \text{APSR\_nzcv}, \text{FPSCR} \quad \text{; transfer FP status register to ARM APSR} \\
\text{VMSR} & \quad \text{FPSCR}, \ r4
\end{align*}
\]
5.7 NEON logical and compare operations

This section contains the following subsections:

- **VAND, VBIC, VEOR, VORN, and VORR (register)** on page 5-29
  Bitwise AND, Bit Clear, Exclusive OR, OR Not, and OR (register).

- **VBIC and VORR (immediate)** on page 5-30
  Bitwise Bit Clear and OR (immediate).

- **VBIF, VBIT, and VBSL** on page 5-31
  Bitwise Insert if False, Insert if True, and Select.

- **VMOV, VMVN (register)** on page 5-32
  Move, and Move NOT.

- **VACGE and VACGT** on page 5-33
  Compare Absolute.

- **VCEQ, VCGE, VCGT, VCLE, and VCLT** on page 5-34
  Compare.

- **VTST** on page 5-35
  Test bits.
5.7.1 **VAND, VBIC, VEOR, VORN, and VORR (register)**

VAND (Bitwise AND), VBIC (Bit Clear), VEOR (Bitwise Exclusive OR), VORN (Bitwise OR NOT), and VORR (Bitwise OR) instructions perform bitwise logical operations between two registers, and place the results in the destination register.

**Syntax**

\[
\text{Vop}\{\text{cond}\}.\{\text{datatype}\} \{Qd\}, Qn, Qm \\
\text{Vop}\{\text{cond}\}.\{\text{datatype}\} \{Dd\}, Dn, Dm
\]

where:

- **op** must be one of:
  - **AND** Logical AND
  - **ORR** Logical OR
  - **EOR** Logical exclusive OR
  - **BIC** Logical AND complement
  - **ORN** Logical OR complement.

- **cond** is an optional condition code (see *Condition codes* on page 5-12).
- **datatype** is an optional data type. The assembler ignores **datatype**.

- **Qd, Qn, Qm** specifies the destination register, the first operand register, and the second operand register, for a quadword operation.

- **Dd, Dn, Dm** specifies the destination register, the first operand register, and the second operand register, for a doubleword operation.

--- **Note** ---

VORR with the same register for both operands is a VMOV instruction. You can use VORR in this way, but disassembly of the resulting code produces the VMOV syntax. See VMOV, VMVN (register) on page 5-32 for details.
5.7.2 **VBIC and VORR (immediate)**

VBIC (Bit Clear immediate) takes each element of the destination vector, performs a bitwise AND Complement with an immediate constant, and returns the result into the destination vector.

VORR (Bitwise OR immediate) takes each element of the destination vector, performs a bitwise OR with an immediate constant, and returns the result into the destination vector.

See also the pseudo-instructions *VAND and VORN (immediate)* on page 5-84.

**Syntax**

\[
\text{V} \text{op} \{\text{cond}\}.\text{datatype} \text{Qd}, \#imm \\
\text{V} \text{op} \{\text{cond}\}.\text{datatype} \text{Dd}, \#imm
\]

where:

- \( \text{op} \) must be either BIC or ORR.
- \( \text{cond} \) is an optional condition code (see *Condition codes* on page 5-12).
- \( \text{datatype} \) must be either I8, I16, I32, or I64.
- \( \text{Qd} \) or \( \text{Dd} \) is the NEON register for the source and result.
- \( \text{imm} \) is the immediate constant.

**Immediate constants**

You can either specify \( \text{imm} \) as a pattern which the assembler repeats to fill the destination register, or you can directly specify the immediate constant (that conforms to the pattern) in full. The pattern for \( \text{imm} \) depends on \( \text{datatype} \) as shown in Table 5-8:

<table>
<thead>
<tr>
<th>( \text{I16} )</th>
<th>( \text{I32} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00XY</td>
<td>0x000000XY</td>
</tr>
<tr>
<td>0xXY00</td>
<td>0x0000XY00</td>
</tr>
<tr>
<td></td>
<td>0x00XY0000</td>
</tr>
<tr>
<td></td>
<td>0xXY000000</td>
</tr>
</tbody>
</table>

If you use the I8 or I64 datatypes, the assembler will convert it to either the I16 or I32 instruction to match the pattern of \( \text{imm} \). If the immediate constant does not match any of the patterns in Table 5-8, the assembler generates an error.
5.7.3 VBIF, VBIT, and VBSL

VBIT (Bitwise Insert if True) inserts each bit from the first operand into the destination if the corresponding bit of the second operand is 1, otherwise leaves the destination bit unchanged.

VBIF (Bitwise Insert if False) inserts each bit from the first operand into the destination if the corresponding bit of the second operand is 0, otherwise leaves the destination bit unchanged.

VBSL (Bitwise Select) selects each bit for the destination from the first operand if the corresponding bit of the destination is 1, or from the second operand if the corresponding bit of the destination is 0.

Syntax

\[
\text{V}op\{\text{cond}\}\{.\text{datatype}\}\{Qd\}, Qn, Qm
\]
\[
\text{V}op\{\text{cond}\}\{.\text{datatype}\}\{Dd\}, Dn, Dm
\]

where:

- **op** must be one of BIT, BIF, or BSL.
- **cond** is an optional condition code (see Condition codes on page 5-12).
- **datatype** is an optional datatype. The assembler ignores datatype.
- **Qd, Qn, Qm** specifies the destination register, the first operand register, and the second operand register, for a quadword operation.
- **Dd, Dn, Dm** specifies the destination register, the first operand register, and the second operand register, for a doubleword operation.
5.7.4 VMOV, VMVN (register)

Vector Move (register) copies a value from the source register into the destination register.

Vector Move Not (register) inverts the value of each bit from the source register and places the results into the destination register.

**Syntax**

```
VMOV{cond}{.datatype} Qd, Qm
VMOV{cond}{.datatype} Dd, Dm
VMVN{cond}{.datatype} Qd, Qm
VMVN{cond}{.datatype} Dd, Dm
```

where:

- `cond` is an optional condition code (see *Condition codes* on page 5-12).
- `datatype` is an optional datatype. The assembler ignores `datatype`.
- `Qd, Qm` specifies the destination vector and the source vector, for a quadword operation.
- `Dd, Dm` specifies the destination vector and the source vector, for a doubleword operation.
5.7.5 **VACGE and VACGT**

Vector Absolute Compare takes the absolute value of each element in a vector, and compares it with the absolute value of the corresponding element of a second vector. If the condition is true, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

See also the pseudo-instructions **VACLE and VACLT** on page 5-85.

**Syntax**

\[
\text{VAC}\{\text{cond}\}.F32 \{Qd\}, Qn, Qm \\
\text{VAC}\{\text{cond}\}.F32 \{Dd\}, Dn, Dm
\]

where:

- \(\text{op}\) must be one of:
  - \(\text{GE}\) Absolute Greater than or Equal
  - \(\text{GT}\) Absolute Greater Than.

- \(\text{cond}\) is an optional condition code (see **Condition codes** on page 5-12).

- \(Qd\), \(Qn\), \(Qm\) specifies the destination register, the first operand register, and the second operand register, for a quadword operation.

- \(Dd\), \(Dn\), \(Dm\) specifies the destination register, the first operand register, and the second operand register, for a doubleword operation.

The result datatype is \(\text{I32}\).
5.7.6 VCEQ, VCGE, VCGT, VCLE, and VCLT

Vector Compare takes the value of each element in a vector, and compares it with the value of the corresponding element of a second vector, or zero. If the condition is true, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

See also the pseudo-instructions VCLE and VCLT on page 5-86.

Syntax

VCop{cond}.datatype {Qd}, Qn, Qm
VCop{cond}.datatype {Dd}, Dn, Dm
VCop{cond}.datatype {Qd}, Qn, #0
VCop{cond}.datatype {Dd}, Dn, #0

where:

*op* must be one of:

- **EQ** Equal
- **GE** Greater than or Equal
- **GT** Greater Than
- **LE** Less than or Equal (only if the second operand is #0)
- **LT** Less Than (only if the second operand is #0).

*cond* is an optional condition code (see *Condition codes* on page 5-12).

*datatype* must be one of:

- I8, I16, I32, or F32 for EQ
- S8, S16, S32, U8, U16, U32, or F32 for GE, GT, LE, or LT (except #0 form)
- S8, S16, S32, or F32 for GE, GT, LE, or LT (#0 form).

The result datatype is:

- I32 for operand datatypes I32, S32, U32, or F32
- I16 for operand datatypes I16, S16, or U16
- I8 for operand datatypes I8, S8, or U8.

Qd, Qn, Qm specifies the destination register, the first operand register, and the second operand register, for a quadword operation.

Dd, Dn, Dm specifies the destination register, the first operand register, and the second operand register, for a doubleword operation.

#0 replaces Qm or Dm for comparisons with zero.
5.7.7 VTST

VTST (Vector Test Bits) takes each element in a vector, and bitwise logical ANDs them with the corresponding element of a second vector. If the result is not zero, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

Syntax

VTST{cond}.size {Qd}, Qn, Qm
VTST{cond}.size {Dd}, Dn, Dm

where:

cond is an optional condition code (see Condition codes on page 5-12).
size must be one of 8, 16, or 32.
Qd, Qn, Qm specifies the destination register, the first operand register, and the second operand register, for a quadword operation.
Dd, Dn, Dm specifies the destination register, the first operand register, and the second operand register, for a doubleword operation.
5.8 NEON general data processing instructions

This section contains the following subsections:

- **VVCVT (between fixed-point or integer, and floating-point)** on page 5-37
  Vector convert between fixed-point or integer and floating-point.

- **VVCVT (between half-precision and single-precision floating-point)** on page 5-38
  Vector convert between half-precision and single-precision floating-point.

- **VDUP** on page 5-39
  Duplicate scalar to all lanes of vector.

- **VEXT** on page 5-40
  Extract.

- **VMOV, VMVN (immediate)** on page 5-41
  Move and Move Negative (immediate).

- **VMOVL, V[Q]MOVN, VQMOVUN** on page 5-42
  Move (register).

- **VREV** on page 5-43
  Reverse elements within a vector.

- **VSWP** on page 5-44
  Swap vectors.

- **VTBL, VTBX** on page 5-45
  Vector table look-up.

- **VTRN** on page 5-46
  Vector transpose.

- **VUZP, VZIP** on page 5-47
  Vector interleave and de-interleave.
5.8.1 VCVT (between fixed-point or integer, and floating-point)

VCVT (Vector Convert) converts each element in a vector in one of the following ways, and places the results in the destination vector:

- from floating-point to integer
- from integer to floating-point
- from floating-point to fixed-point
- from fixed-point to floating-point.

Syntax

```
VCVT{cond}.type Qd, Qm {, #fbits}
VCVT{cond}.type Dd, Dm {, #fbits}
```

where:

- `cond` is an optional condition code (see Condition codes on page 5-12).
- `type` specifies the data types for the elements of the vectors. It must be one of:
  - `S32.F32` floating-point to signed integer or fixed-point
  - `U32.F32` floating-point to unsigned integer or fixed-point
  - `F32.S32` signed integer or fixed-point to floating-point
  - `F32.U32` unsigned integer or fixed-point to floating-point
- `Qd, Qm` specifies the destination vector and the operand vector, for a quadword operation.
- `Dd, Dm` specifies the destination vector and the operand vector, for a doubleword operation.
- `fbits` if present, specifies the number of fraction bits in the fixed point number. Otherwise, the conversion is between floating-point and integer. `fbits` must lie in the range 0-32. If `fbits` is omitted, the number of fraction bits is 0.

Rounding

Integer or fixed-point to floating-point conversions use round to nearest.

Floating-point to integer or fixed-point conversions use round towards zero.
5.8.2 VCVT (between half-precision and single-precision floating-point)

VCVT (Vector Convert), with half-precision extension, converts each element in a vector in one of the following ways, and places the results in the destination vector:

- from half-precision floating-point to single-precision floating-point (F32.F16)
- from single-precision floating-point to half-precision floating-point (F16.F32).

**Syntax**

\[
\text{VCVT}\{\text{cond}\}.F32.F16 \ Qd, \ Dm \\
\text{VCVT}\{\text{cond}\}.F16.F32 \ Dd, \ Qm
\]

where:

- \(\text{cond}\) is an optional condition code (see Condition codes on page 5-12).
- \(Qd, \ Dm\) specifies the destination vector for the single-precision results and the half-precision operand vector.
- \(Dd, \ Qm\) specifies the destination vector for half-precision results and the single-precision operand vector.

**Architectures**

This instruction is only available in NEON systems with the half-precision extension.
5.8.3 VDUP

VDUP (Vector Duplicate) duplicates a scalar into every element of the destination vector. The source can be a NEON scalar or an ARM register.

**Syntax**

VDUP{cond}.size Qd, Dm[x]
VDUP{cond}.size Dd, Dm[x]
VDUP{cond}.size Qd, Rm
VDUP{cond}.size Dd, Rm

where:

- **cond**  is an optional condition code (see *Condition codes* on page 5-12).
- **size**  must be 8, 16, or 32.
- **Qd**  specifies the destination register for a quadword operation.
- **Dd**  specifies the destination register for a doubleword operation.
- **Dm[x]**  specifies the NEON scalar.
- **Rm**  specifies the ARM register. Rm must not be the pc.
5.8.4 VEXT

VEXT (Vector Extract) extracts 8-bit elements from the bottom end of the second operand vector and the top end of the first, concatenates them, and places the result in the destination vector. See Figure 5-2 for an example.

![Figure 5-2 Operation of doubleword VEXT for imm = 3](image)

**Syntax**

```
VEXT{cond}.8 {Qd}, Qn, Qm, #imm
VEXT{cond}.8 {Dd}, Dn, Dm, #imm
```

where:

- `cond` is an optional condition code (see Condition codes on page 5-12).
- `Qd, Qn, Qm` specifies the destination register, the first operand register, and the second operand register, for a quadword operation.
- `Dd, Dn, Dm` specifies the destination register, the first operand register, and the second operand register, for a doubleword operation.
- `imm` is the number of 8-bit elements to extract from the bottom of the second operand vector, in the range 0-7 for doubleword operations, or 0-15 for quadword operations.

**VEXT pseudo-instruction**

You can specify a datatype of 16, 32, or 64 instead of 8. In this case, `#imm` refers to halfwords, words, or doublewords instead of referring to bytes, and the allowed ranges are correspondingly reduced.
5.8.5 VMOV, VMVN (Immediate)

VMOV (Vector Move) and VMVN (Vector Move Negative) immediate generate an immediate constant into the destination register.

Syntax

\[
\begin{align*}
\text{V}\text{op}\{\text{cond}\}.\text{datatype}\ \text{Qd}, \#\text{imm} \\
\text{V}\text{op}\{\text{cond}\}.\text{datatype}\ \text{Dd}, \#\text{imm}
\end{align*}
\]

where:

- \(\text{op}\) must be either MOV or MNV.
- \(\text{cond}\) is an optional condition code (see Condition codes on page 5-12).
- \(\text{datatype}\) must be one of I8, I16, I32, I64, or F32.
- \(\text{Qd}\) or \(\text{Dd}\) is the NEON register for the result.
- \(\text{imm}\) is a constant of the type specified by \(\text{datatype}\). This is replicated to fill the destination register.

<table>
<thead>
<tr>
<th>datatype</th>
<th>VMOV</th>
<th>VMVN</th>
</tr>
</thead>
<tbody>
<tr>
<td>I8</td>
<td>0xXY</td>
<td>-</td>
</tr>
<tr>
<td>I16</td>
<td>0x00XY, 0xXY00</td>
<td>0xFFFFY, 0xXYFF</td>
</tr>
<tr>
<td>I32</td>
<td>0x000000XY, 0x0000XY00, 0x00XY0000, 0xXY000000</td>
<td>0xFFFFXYFF, 0xFFFFXYFF, 0xFFFFYFFF, 0xFFFFYYYY</td>
</tr>
<tr>
<td></td>
<td>0x0000XYFF, 0x00XYFFFF</td>
<td>0xFFFFXY00, 0xFFFFXY00</td>
</tr>
<tr>
<td>I64</td>
<td>byte masks, 0xCGHJJKLLLMMNNPP (^a)</td>
<td>-</td>
</tr>
<tr>
<td>F32</td>
<td>floating-point numbers (^b)</td>
<td>-</td>
</tr>
</tbody>
</table>

\(^a\) Each of 0xGG, 0xHH, 0xJJ, 0xKK, 0xLL, 0xMM, 0xNN, and 0xPP must be either 0x00 or 0xFFF.

\(^b\) Any number that can be expressed as +/-n \(*2^r\), where \(n\) and \(r\) are integers, 16 <= \(n\) <= 31, 0 <= \(r\) <= 7.
5.8.6 VMOVL, V{Q}MOVN, VQMOVUN

VMOVL (Vector Move Long) takes each element in a doubleword vector, sign or zero extends them to twice their original length, and places the results in a quadword vector.

VMOVN (Vector Move and Narrow) copies the least significant half of each element of a quadword vector into the corresponding elements of a doubleword vector.

VQMOVN (Vector Saturating Move and Narrow) copies each element of the operand vector to the corresponding element of the destination vector. The result element is half the width of the operand element, and values are saturated to the result width.

VQMOVUN (Vector Saturating Move and Narrow, signed operand with Unsigned result) copies each element of the operand vector to the corresponding element of the destination vector. The result element is half the width of the operand element, and values are saturated to the result width.

Syntax

VMOVL{cond}.datatype Qd, Dm
V{Q}MOVN{cond}.datatype Dd, Qm
VQMOVUN{cond}.datatype Dd, Qm

where:

Q if present, specifies that the results are saturated.
cond is an optional condition code (see Condition codes on page 5-12).
datatype must be one of:

\begin{align*}
\text{S8, S16, S32} & \quad \text{for VMOVL} \\
\text{U8, U16, U62} & \quad \text{for VMOVL} \\
\text{I16, I32, I64} & \quad \text{for VMOVN} \\
\text{S16, S32, S64} & \quad \text{for VQMOVN or VQMOVUN} \\
\text{U16, U32, U64} & \quad \text{for VQMOVN.}
\end{align*}

Qd, Dm specifies the destination vector and the operand vector for VMOVL.

Dd, Qm specifies the destination vector and the operand vector for V{Q}MOV{U}N.
5.8.7 VREV

VREV16 (Vector Reverse within halfwords) reverses the order of 8-bit elements within each halfword of the vector, and places the result in the corresponding destination vector.

VREV32 (Vector Reverse within words) reverses the order of 8-bit or 16-bit elements within each word of the vector, and places the result in the corresponding destination vector.

VREV64 (Vector Reverse within doublewords) reverses the order of 8-bit, 16-bit, or 32-bit elements within each doubleword of the vector, and places the result in the corresponding destination vector.

Syntax

```
VREVn{cond}.size Qd, Qm
VREVn{cond}.size Dd, Dm
```

where:

- \( n \) must be one of 16, 32, or 64.
- \( cond \) is an optional condition code (see Condition codes on page 5-12).
- \( size \) must be one of 8, 16, or 32, and must be less than \( n \).
- \( Qd, Qm \) specifies the destination vector and the operand vector, for a quadword operation.
- \( Dd, Dm \) specifies the destination vector and the operand vector, for a doubleword operation.
5.8.8 VSWP

VSWP (Vector Swap) exchanges the contents of two vectors. The vectors can be either doubleword or quadword. There is no distinction between data types.

Syntax

VSWP{cond}\{.datatype\} Qd, Qm
VSWP{cond}\{.datatype\} Dd, Dm

where:

- \(cond\) is an optional condition code (see Condition codes on page 5-12).
- \(datatype\) is an optional datatype. The assembler ignores \(datatype\).
- \(Qd, Qm\) specifies the vectors for a quadword operation.
- \(Dd, Dm\) specifies the vectors for a doubleword operation.
5.8.9 VTBL, VTBX

VTBL (Vector Table Lookup) uses byte indexes in a control vector to look up byte values in a table and generate a new vector. Indexes out of range return 0.

VTBX (Vector Table Extension) works in the same way, except that indexes out of range leave the destination element unchanged.

Syntax

\[ \text{V\{cond\}8 \ Dd, list, Dm} \]

where:

- \( \text{op} \) must be either TBL or TBX.
- \( \text{cond} \) is an optional condition code (see Condition codes on page 5-12).
- \( \text{Dd} \) specifies the destination vector.
- \( \text{list} \) specifies the vectors containing the table. It must be one of:
  - \{Dn\}
  - \{Dn,D(n+1)\}
  - \{Dn,D(n+1),D(n+2)\}
  - \{Dn,D(n+1),D(n+2),D(n+3)\}
  All the registers in list must be in the range D0-D31.
- \( \text{Dm} \) specifies the index vector.
5.8.10 VTRN

VTRN (Vector Transpose) treats the elements of its operand vectors as elements of 2 x 2 matrices, and transposes the matrices. Figure 5-3 and Figure 5-4 show examples of the operation of VTRN.

**Syntax**

VTRN{cond}.size Qd, Qm
VTRN{cond}.size Dd, Dm

where:
- *cond* is an optional condition code (see Condition codes on page 5-12).
- *size* must be one of 8, 16, or 32.
- *Qd, Qm* specifies the vectors, for a quadword operation.
- *Dd, Dm* specifies the vectors, for a doubleword operation.
5.8.11 VUZP, VZIP

VZIP (Vector Zip) interleaves the elements of two vectors.

VUZP (Vector Unzip) de-interleaves the elements of two vectors.

See De-interleaving an array of 3-element structures on page 5-72 for an example of de-interleaving. Interleaving is the inverse process.

Syntax

\[ Vop\{cond\}.size\ Qd,\ Qm \]
\[ Vop\{cond\}.size\ Dd,\ Dm \]

where:

- \( op \) must be either UZP or ZIP.
- \( cond \) is an optional condition code (see Condition codes on page 5-12).
- \( size \) must be one of 8, 16, or 32.
- \( Qd,\ Qm \) specifies the vectors, for a quadword operation.
- \( Dd,\ Dm \) specifies the vectors, for a doubleword operation.

Note

The following are all the same instruction:

- \( \text{VZIP.32}\ Dd,\ Dm \)
- \( \text{VUZP.32}\ Dd,\ Dm \)
- \( \text{VTRN.32}\ Dd,\ Dm \)

The instruction is disassembled as \( \text{VTRN.32}\ Dd,\ Dm \). See also \( \text{VTRN} \) on page 5-46.
5.9 NEON shift instructions

This section contains the following subsections:

- **VSHL, VQSHL, VQSHLU, and VSHLL (by immediate)** on page 5-49
  Shift Left by immediate value.

- **V{Q}{R}SHL (by signed variable)** on page 5-50
  Shift left by signed variable.

- **V{R}SHR{N}, V{R}SRA (by immediate)** on page 5-51
  Shift Right by immediate value.

- **VQ{R}SHR{U}N (by immediate)** on page 5-52
  Shift Right by immediate value, and saturate.

- **VSLI and VSRI** on page 5-53
  Shift Left and Insert, and Shift Right and Insert.
5.9.1 VSHL, VQSHL, VQSHLU, and VSHLL (by immediate)

Vector Shift Left (by immediate) instructions take each element in a vector of integers, left shift them by an immediate value, and place the results in the destination vector.

For VSHL (Vector Shift Left), bits shifted out of the left of each element are lost.

For VQSHL (Vector Saturating Shift Left) and VQSHLU (Vector Saturating Shift Left Unsigned), the sticky QC flag (FPSCR bit[27]) is set if saturation occurs.

For VSHLL (Vector Shift Left Long), values are sign or zero extended.

Syntax

\[
\begin{align*}
V\{Q\}\text{SHL}\{U\}\{\text{cond}\}\cdot\text{datatype}\ {Qd}, \ Qm, \ #\text{imm} \\
V\{Q\}\text{SHL}\{U\}\{\text{cond}\}\cdot\text{datatype}\ {Dd}, \ Dm, \ #\text{imm} \\
\text{VSHLL\{cond\}\cdot\text{datatype}\ Qd, \ Dm, \ #\text{imm}}
\end{align*}
\]

where:

- **Q** if present, indicates that if any of the results overflow, they are saturated.
- **U** only allowed if Q is also present. Indicates that the results are unsigned even though the operands are signed.
- **cond** is an optional condition code (see Condition codes on page 5-12).
- **datatype** must be one of:
  - I8, I16, I32, I64 for VSHL
  - S8, S16, S32 for VSHLL, VQSHL, or VQSHLU
  - U8, U16, U32 for VSHLL or VQSHL
  - S64 for VQSHL or VQSHLU
  - U64 for VQSHL.
- **Qd, Qm** are the destination and operand vectors, for a quadword operation.
- **Dd, Dm** are the destination and operand vectors, for a doubleword operation.
- **Qd, Dm** are the destination and operand vectors, for a long operation.
- **imm** is the immediate constant specifying the size of the shift, in the range:
  - 1 to size(datatype) for VSHLL
  - 1 to (size(datatype) – 1) for VSHL, VQSHL, or VQSHLU.

0 is permitted, but the resulting code disassembles to VMOV or VMOV.
5.9.2 \texttt{V\{Q\}{R}SHL} (by signed variable)

\texttt{VSHL} (Vector Shift Left by signed variable) takes each element in a vector, shifts them by a value from the least significant byte of the corresponding element of a second vector, and places the results in the destination vector. If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift.

The results can be optionally saturated, rounded, or both. The sticky QC flag (FPSCR bit[27]) is set if saturation occurs.

**Syntax**

\[
\texttt{V\{Q\}{R}SHL\{cond\}.datatype \{Qd\}, Qm, Qn} \\
\texttt{V\{Q\}{R}SHL\{cond\}.datatype \{Dd\}, Dm, Dn}
\]

where:

- \texttt{Q} if present, indicates that If any of the results overflow, they are saturated.
- \texttt{R} if present, indicates that each result is rounded. Otherwise, each result is truncated.
- \texttt{cond} is an optional condition code (see \textit{Condition codes} on page 5-12).
- \texttt{datatype} must be one of \texttt{S8}, \texttt{S16}, \texttt{S32}, \texttt{S64}, \texttt{U8}, \texttt{U16}, \texttt{U32}, or \texttt{U64}.
- \texttt{Qd, Qm, Qn} are the destination vector, the first operand vector, and the second operand vector, for a quadword operation.
- \texttt{Dd, Dm, Dn} are the destination vector, the first operand vector, and the second operand vector, for a doubleword operation.
5.9.3 \( V\{R\}SHR\{N\}, V\{R\}SRA \) (by immediate)

\( V\{R\}SHR\{N\} \) (Vector Shift Right by immediate value) takes each element in a vector, right shifts them by an immediate value, and places the results in the destination vector. The results can be optionally rounded, or narrowed, or both.

\( V\{R\}SRA \) (Vector Shift Right by immediate value and Accumulate) takes each element in a vector, right shifts them by an immediate value, and accumulates the results into the destination vector. The results can be optionally rounded.

Syntax

\[
\begin{align*}
V\{R\}SHR\{cond\} & . \text{datatype} \{Qd\}, Qm, \#imm \\
V\{R\}SHR\{cond\} & . \text{datatype} \{Dd\}, Dm, \#imm \\
V\{R\}SRA\{cond\} & . \text{datatype} \{Qd\}, Qm, \#imm \\
V\{R\}SRA\{cond\} & . \text{datatype} \{Dd\}, Dm, \#imm \\
V\{R\}SHRN\{cond\} & . \text{datatype} \{Dd\}, Qm, \#imm \\
\end{align*}
\]

where:

- \( R \) if present, indicates that the results are rounded. Otherwise, the results are truncated.
- \( cond \) is an optional condition code (see Condition codes on page 5-12).
- \text{datatype} must be one of:
  - \( S8, S16, S32, S64 \) for \( V\{R\}SHR \) or \( V\{R\}SRA \)
  - \( U8, U16, U32, U64 \) for \( V\{R\}SHR \) or \( V\{R\}SRA \)
  - \( I16, I32, I64 \) for \( V\{R\}SHRN \).
- \( Qd, Qm \) are the destination vector and the operand vector, for a quadword operation.
- \( Dd, Dm \) are the destination vector and the operand vector, for a doubleword operation.
- \( Dd, Qm \) are the destination vector and the operand vector, for a narrow operation.
- \( imm \) is the immediate constant specifying the size of the shift, in the range 0 to \((\text{size}\,(\text{datatype}) - 1)\).
5.9.4 \textbf{VQ\{R\}SHR\{U\}N (by immediate)}

\(\text{VQ\{R\}SHR\{U\}N}\) (Vector Saturating Shift Right, Narrow, by immediate value, with optional Rounding) takes each element in a quadword vector of integers, right shifts them by an immediate value, and places the results in a doubleword vector.

The sticky QC flag (FPSCR bit[27]) is set if saturation occurs.

\textbf{Syntax}

\[
\text{VQ\{R\}SHR\{U\}N}\{\text{cond}\}.\text{datatype} \ Dd, \ Qm, \ #imm
\]

\textbf{where:}

- \(\text{R}\) if present, indicates that the results are rounded. Otherwise, the results are truncated.
- \(\text{U}\) if present, indicates that the results are unsigned, although the operands are signed. Otherwise, the results are the same type as the operands.
- \(\text{cond}\) is an optional condition code (see \textit{Condition codes} on page 5-12).
- \(\text{datatype}\) must be one of:
  - \(\text{S16, S32, S64}\) for \(\text{VQ\{R\}SHRN}\) or \(\text{VQ\{R\}SHRUN}\)
  - \(\text{U16, U32, U64}\) for \(\text{VQ\{R\}SHRN}\) only.
- \(\text{Dd, Qm}\) are the destination vector and the operand vector.
- \(\text{imm}\) is the immediate constant specifying the size of the shift, in the range 0 to \((\text{size(\text{datatype}) – 1})\).
5.9.5 VSLI and VSRI

VSLI (Vector Shift Left and Insert) takes each element in a vector, left shifts them by an immediate value, and inserts the results in the destination vector. Bits shifted out of the left of each element are lost.

VSRI (Vector Shift Right and Insert) takes each element in a vector, right shifts them by an immediate value, and inserts the results in the destination vector. Bits shifted out of the right of each element are lost.

Syntax

\[ \text{Vop} \{\text{cond}\}.\text{size} \{Qd\}, \text{Qm}, \#\text{imm} \]
\[ \text{Vop} \{\text{cond}\}.\text{size} \{Dd\}, \text{Dm}, \#\text{imm} \]

where:

- \( \text{op} \) must be either SLI or SRI.
- \( \text{cond} \) is an optional condition code (see Condition codes on page 5-12).
- \( \text{size} \) must be one of 8, 16, 32, or 64.
- \( \text{Qd}, \text{Qm} \) are the destination vector and the operand vector, for a quadword operation.
- \( \text{Dd}, \text{Dm} \) are the destination vector and the operand vector, for a doubleword operation.
- \( \text{imm} \) is the immediate constant specifying the size of the shift, in the range:
  - 0 to \( (\text{size} - 1) \) for VSLI
  - 1 to \( \text{size} \) for VSRI.
5.10 NEON general arithmetic instructions

This section contains the following subsections:

- **VABA[L] and VABD[L] on page 5-55**
  Vector Absolute Difference and Accumulate, and Absolute Difference.

- **V(Q)ABS and V(Q)NEG on page 5-56**
  Vector Absolute value, and Negate.

- **V(Q)ADD, VADDL, VADDW, V(Q)SUB, VSUBL, and VSUBW on page 5-57**
  Vector Add and Subtract.

- **V(R)ADDDN and V(R)SUBHN on page 5-58**
  Vector Add selecting High half, and Subtract selecting High Half.

- **V(R)HADD and VHSUB on page 5-59**
  Vector Halving Add and Subtract.

- **VPADD[L], VPADAL on page 5-60**
  Vector Pairwise Add, Add and Accumulate.

- **VMAX, VMIN, VPMAX, and VPMIN on page 5-62**
  Vector Maximum, Minimum, Pairwise Maximum, and Pairwise Minimum.

- **VCLS, VCLZ, and VCNT on page 5-63**
  Vector Count Leading Sign bits, Count Leading Zeros, and Count set bits.

- **VRECPE and VRSQRTE on page 5-64**
  Vector Reciprocal Estimate and Reciprocal Square Root Estimate.

- **VRECP$ and VRSQRTS on page 5-65**
  Vector Reciprocal Step and Reciprocal Square Root Step.
5.10.1 VABA{L} and VABD{L}

VABA (Vector Absolute Difference and Accumulate) subtracts the elements of one vector from the corresponding elements of another vector, and accumulates the absolute values of the results into the elements of the destination vector.

VABD (Vector Absolute Difference) subtracts the elements of one vector from the corresponding elements of another vector, and places the absolute values of the results into the elements of the destination vector.

Long versions of both instructions are available.

Syntax

\[ \text{Vop}\{\text{cond}\}.\text{datatype}\ \{\text{Qd}, \ Qn, \ Qm}\]
\[ \text{Vop}\{\text{cond}\}.\text{datatype}\ \{\text{Dd}, \ Dn, \ Dm}\]
\[ \text{VopL}\{\text{cond}\}.\text{datatype}\ \text{Qd, Dn, Dm}\]

where:

- \(\text{op}\) must be either ABA or ABD.
- \(\text{cond}\) is an optional condition code (see Condition codes on page 5-12).
- \(\text{datatype}\) must be one of:
  - \(\text{S8}, \text{S16}, \text{S32}, \text{U8}, \text{U16}, \text{U32}\) for VABA, VABAL, or VABDL.
  - \(\text{S8}, \text{S16}, \text{S32}, \text{U8}, \text{U16}, \text{U32}\) or \(\text{F32}\) for VABD.

\(\text{Qd}, \ Qn, \ Qm\) are the destination vector, the first operand vector, and the second operand vector, for a quadword operation.

\(\text{Dd}, \ Dn, \ Dm\) are the destination vector, the first operand vector, and the second operand vector, for a doubleword operation.

\(\text{Qd}, \ Dn, \ Dm\) are the destination vector, the first operand vector, and the second operand vector, for a long operation.
5.10.2 **V(Q)ABS and V(Q)NEG**

VABS (Vector Absolute) takes the absolute value of each element in a vector, and places the results in a second vector. (The floating-point version only clears the sign bit.)

VNEG (Vector Negate) negates each element in a vector, and places the results in a second vector. (The floating-point version only inverts the sign bit.)

Saturating versions of both instructions are available. The sticky QC flag (FPSCR bit[27]) is set if saturation occurs.

**Syntax**

\[
V\{Q\}op\{cond\}.datatype\ Qd,\ Qm\\
V\{Q\}op\{cond\}.datatype\ Dd,\ Dm
\]

where:

- **Q** if present, indicates that If any of the results overflow, they are saturated.
- **op** must be either ABS or NEG.
- **cond** is an optional condition code (see *Condition codes* on page 5-12).
- **datatype** must be one of:
  - S8, S16, S32 for VABS, VNEG, VQABS, or VQNEG
  - F32 for VABS and VNEG only.

- **Qd, Qm** are the destination vector and the operand vector, for a quadword operation.
- **Dd, Dm** are the destination vector and the operand vector, for a doubleword operation.
5.10.3 \textit{VQ}ADD, VADDL, VADDW, V(Q)SUB, VSUBL, and VSUBW

\textit{VADD} (Vector Add) adds corresponding elements in two vectors, and places the results in the destination vector.

\textit{VSUB} (Vector Subtract) subtracts the elements of one vector from the corresponding elements of another vector, and places the results in the destination vector.

Saturating, Long, and Wide versions are available. The sticky QC flag (FPSCR bit[27]) is set if saturation occurs.

\textbf{Syntax}

\begin{align*}
\textit{V(Q)op} \{\textit{cond}\}.\textit{datatype} \{Qd\}, Qn, Qm \\
\textit{V(Q)op} \{\textit{cond}\}.\textit{datatype} \{Dd\}, Dn, Dm \\
\textit{VopL}\{\textit{cond}\}.\textit{datatype} Qd, Dn, Dm \\
\textit{VopW}\{\textit{cond}\}.\textit{datatype} \{Qd\}, Qn, Dm
\end{align*}

where:

\begin{itemize}
  \item \textit{Q} if present, indicates that if any of the results overflow, they are saturated.
  \item \textit{op} must be either \textit{ADD} or \textit{SUB}.
  \item \textit{cond} is an optional condition code (see \textit{Condition codes} on page 5-12).
  \item \textit{datatype} must be one of:
    \begin{itemize}
      \item I8, I16, I32, I64, F32 for \textit{VADD} or \textit{VSUB}
      \item S8, S16, S32 for \textit{VQADD}, \textit{VQSUB}, \textit{VADDL}, \textit{VADDW}, \textit{VSUBL}, or \textit{VSUBW}
      \item U8, U16, U32 for \textit{VQADD}, \textit{VQSUB}, \textit{VADDL}, \textit{VADDW}, \textit{VSUBL}, or \textit{VSUBW}
      \item S64, U64 for \textit{VQADD} or \textit{VQSUB}.
    \end{itemize}
  \end{itemize}

\textit{Qd}, Qn, Qm are the destination vector, the first operand vector, and the second operand vector, for a quadword operation.

\textit{Dd}, Dn, Dm are the destination vector, the first operand vector, and the second operand vector, for a doubleword operation.

\textit{Qd}, Dn, Dm are the destination vector, the first operand vector, and the second operand vector, for a long operation.

\textit{Qd}, Qn, Dm are the destination vector, the first operand vector, and the second operand vector, for a wide operation.
5.10.4 \texttt{V\{R\}ADDHN and V\{R\}SUBHN}

\texttt{V\{R\}ADDH} (Vector Add and Narrow, selecting High half) adds corresponding elements in two vectors, selects the most significant halves of the results, and places the final results in the destination vector. Results can be either rounded or truncated.

\texttt{V\{R\}SUBH} (Vector Subtract and Narrow, selecting High half) subtracts the elements of one vector from the corresponding elements of another vector, selects the most significant halves of the results, and places the final results in the destination vector. Results can be either rounded or truncated.

**Syntax**

\begin{verbatim}
V\{R\}opHN{cond}.datatype Dd, Qn, Qm
\end{verbatim}

where:

- \texttt{R} if present, indicates that each result is rounded. Otherwise, each result is truncated.
- \texttt{op} must be either \texttt{ADD} or \texttt{SUB}.
- \texttt{cond} is an optional condition code (see \textit{Condition codes} on page 5-12).
- \texttt{datatype} must be one of \texttt{I16}, \texttt{I32}, or \texttt{I64}.
- \texttt{Dd, Qn, Qm} are the destination vector, the first operand vector, and the second operand vector.
5.10.5 V{R}HADD and VHSUB

VHADD (Vector Halving Add) adds corresponding elements in two vectors, shifts each result right one bit, and places the results in the destination vector. Results can be either rounded or truncated.

VHSUB (Vector Halving Subtract) subtracts the elements of one vector from the corresponding elements of another vector, shifts each result right one bit, and places the results in the destination vector. Results are always truncated.

Syntax

\[
\begin{align*}
\text{VHADD}\{\text{cond}\}.\text{datatype} \{\text{Qd}, \text{Qn}, \text{Qm}\} \\
\text{VHADD}\{\text{cond}\}.\text{datatype} \{\text{Dd}, \text{Dn}, \text{Dm}\} \\
\text{VHSUB}\{\text{cond}\}.\text{datatype} \{\text{Qd}, \text{Qn}, \text{Qm}\} \\
\text{VHSUB}\{\text{cond}\}.\text{datatype} \{\text{Dd}, \text{Dn}, \text{Dm}\}
\end{align*}
\]

where:

R if present, indicates that each result is rounded. Otherwise, each result is truncated.

\text{cond} is an optional condition code (see Condition codes on page 5-12).

\text{datatype} must be one of S8, S16, S32, U8, U16, or U32.

\text{Qd, Qn, Qm} are the destination vector, the first operand vector, and the second operand vector, for a quadword operation.

\text{Dd, Dn, Dm} are the destination vector, the first operand vector, and the second operand vector, for a doubleword operation.
5.10.6 **VPADD{L}, VPADAL**

VPADD (Vector Pairwise Add) adds adjacent pairs of elements of two vectors, and places the results in the destination vector.

![Figure 5-5 Example of operation of VPADD (in this case, for data type I16)](image)

VPADDL (Vector Pairwise Add Long) adds adjacent pairs of elements of a vector, sign or zero extends the results to twice their original width, and places the final results in the destination vector.

![Figure 5-6 Example of operation of doubleword VPADDL (in this case, for data type S16)](image)

VPADAL (Vector Pairwise Add and Accumulate Long) adds adjacent pairs of elements of a vector, and accumulates the absolute values of the results into the elements of the destination vector.

![Figure 5-7 Example of operation of VPADAL (in this case for data type S16)](image)
Syntax

VPADD{\(\text{cond}\)}.\(\text{datatype}\) \{\(Dd\), \(Dn\), \(Dm\)\}

VPopl{\(\text{cond}\)}.\(\text{datatype}\) \{\(Qd\), \(Qm\)\}

VPopl{\(\text{cond}\)}.\(\text{datatype}\) \{\(Dd\), \(Dm\)\}

where:

\(\text{op}\) must be either ADD or ADA.

\(\text{cond}\) is an optional condition code (see Condition codes on page 5-12).

\(\text{datatype}\) must be one of:

- I8, I16, I32, F32 for VPADD
- S8, S16, S32 for VPADDL or VPADAL
- U8, U16, U32 for VPADDL or VPADAL.

\(Dd\), \(Dn\), \(Dm\) are the destination vector, the first operand vector, and the second operand vector, for a VPADD instruction.

\(Qd\), \(Qm\) are the destination vector and the operand vector, for a quadword VPADDL or VPADAL.

\(Dd\), \(Dm\) are the destination vector and the operand vector, for a doubleword VPADDL or VPADAL.
5.10.7 VMAX, VMIN, VPMAX, and VPMIN

VMAX (Vector Maximum) compares corresponding elements in two vectors, and copies the larger of each pair into the corresponding element in the destination vector.

VMIN (Vector Minimum) compares corresponding elements in two vectors, and copies the smaller of each pair into the corresponding element in the destination vector.

VPMAX (Vector Pairwise Maximum) compares adjacent pairs of elements in two vectors, and copies the larger of each pair into the corresponding element in the destination vector. Operands and results must be doubleword vectors.

VPMIN (Vector Pairwise Minimum) compares adjacent pairs of elements in two vectors, and copies the smaller of each pair into the corresponding element in the destination vector. Operands and results must be doubleword vectors.

See Figure 5-5 on page 5-60 for a diagram of a pairwise operation.

Syntax

V<op>{cond}.datatype Qd, Qn, Qm
V<op>{cond}.datatype Dd, Dn, Dm
VP<op>{cond}.datatype Dd, Dn, Dm

where:

<op> must be either MAX or MIN.
<cond> is an optional condition code (see Condition codes on page 5-12).
<datatype> must be one of S8, S16, S32, U8, U16, U32, or F32.
Qd, Qn, Qm are the destination vector, the first operand vector, and the second operand vector, for a quadword operation.
Dd, Dn, Dm are the destination vector, the first operand vector, and the second operand vector, for a doubleword operation.

Floating-point maximum and minimum

max(+0.0, –0.0) = +0.0.
min(+0.0, –0.0) = –0.0

If any input is a NaN, the corresponding result element is the default NaN.
5.10.8 VCLS, VCLZ, and VCNT

VCLS (Vector Count Leading Sign bits) counts the number of consecutive bits following the topmost bit, that are the same as the topmost bit, in each element in a vector, and places the results in a second vector.

VCLZ (Vector Count Leading Zeros) counts the number of consecutive zeros, starting from the top bit, in each element in a vector, and places the results in a second vector.

VCNT (Vector Count set bits) counts the number of bits that are one in each element in a vector, and places the results in a second vector.

Syntax

\texttt{V\texttt{oper}}\{\texttt{cond}\}.\texttt{datatype} \texttt{Qd, Qm}

\texttt{V\texttt{oper}}\{\texttt{cond}\}.\texttt{datatype} \texttt{Dd, Dm}

where:

- \textit{op} must be one of CLS, CLZ, or CNT.
- \textit{cond} is an optional condition code (see \textit{Condition codes} on page 5-12).
- \textit{datatype} must be one of:
  - S8, S16, or S32 for CLS.
  - I8, I16, or I32 for CLZ.
  - I8 for CNT.

\texttt{Qd, Qm} are the destination vector and the operand vector, for a quadword operation.

\texttt{Dd, Dm} are the destination vector and the operand vector, for a doubleword operation.
5.10.9 VRECPE and VRSQRTE

VRECPE (Vector Reciprocal Estimate) finds an approximate reciprocal of each element in a vector, and places the results in a second vector.

VRSQRTE (Vector Reciprocal Square Root Estimate) finds an approximate reciprocal square root of each element in a vector, and places the results in a second vector.

Syntax

\[ \text{Vop}\{\text{cond}\}.\text{datatype}\ Qd,\ Qm \]
\[ \text{Vop}\{\text{cond}\}.\text{datatype}\ Dd,\ Dm \]

where:
- \( \text{op} \) must be either \( \text{RECPE} \) or \( \text{RSQRTE} \).
- \( \text{cond} \) is an optional condition code (see Condition codes on page 5-12).
- \( \text{datatype} \) must be either \( \text{U32} \) or \( \text{F32} \).
- \( Qd,\ Qm \) are the destination vector and the operand vector, for a quadword operation.
- \( Dd,\ Dm \) are the destination vector and the operand vector, for a doubleword operation.

Results for out-of-range inputs

Table 5-10 shows the results where input values are out of range.

<table>
<thead>
<tr>
<th>Operand element (VRECPE)</th>
<th>Operand element (VRSQRTE)</th>
<th>Result element</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer ( \leq 0x7FFFFFFF )</td>
<td>Integer ( \leq 0x3FFFFFFF )</td>
<td>0xFFFFFFFF</td>
</tr>
<tr>
<td>Floating-point</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NaN</td>
<td>NaN, Negative Normal, Negative Infinity</td>
<td>Default NaN</td>
</tr>
<tr>
<td>Negative 0, Negative Denormal</td>
<td>Negative 0, Negative Denormal</td>
<td>Negative Infinity a</td>
</tr>
<tr>
<td>Positive 0, Positive Denormal</td>
<td>Positive 0, Positive Denormal</td>
<td>Positive Infinity a</td>
</tr>
<tr>
<td>Positive infinity</td>
<td>Positive infinity</td>
<td>Positive 0</td>
</tr>
<tr>
<td>Negative infinity</td>
<td>Negative 0</td>
<td></td>
</tr>
</tbody>
</table>

a. The Division by Zero exception bit in the FPSCR (FPSCR[1]) is set
5.10.10 VRECPS and VRSQRTS

VRECPS (Vector Reciprocal Step) multiplies the elements of one vector by the corresponding elements of another vector, subtracts each of the results from 2, and places the final results into the elements of the destination vector.

VRSQRTS (Vector Reciprocal Square Root Step) multiplies the elements of one vector by the corresponding elements of another vector, subtracts each of the results from 3, divides these results by two, and places the final results into the elements of the destination vector.

Syntax

\[
Vop(\text{cond}).F32 \{Qd\}, Qn, Qm \\
Vop(\text{cond}).F32 \{Dd\}, Dn, Dm
\]

where:

- \( op \) must be either RECPS or RSQRTS.
- \( \text{cond} \) is an optional condition code (see Condition codes on page 5-12).
- \( Qd, Qn, Qm \) are the destination vector, the first operand vector, and the second operand vector, for a quadword operation.
- \( Dd, Dn, Dm \) are the destination vector, the first operand vector, and the second operand vector, for a doubleword operation.

Results for out-of-range inputs

Table 5-11 shows the results where input values are out of range.

<table>
<thead>
<tr>
<th>1st operand element</th>
<th>2nd operand element</th>
<th>Result element (VRECPS)</th>
<th>Result element (VRSQRTS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>NaN</td>
<td>-</td>
<td>Default NaN</td>
<td>Default NaN</td>
</tr>
<tr>
<td>-</td>
<td>NaN</td>
<td>Default NaN</td>
<td>Default NaN</td>
</tr>
<tr>
<td>+/- 0.0 or denormal</td>
<td>+/- infinity</td>
<td>2.0</td>
<td>1.5</td>
</tr>
<tr>
<td>+/- infinity</td>
<td>+/- 0.0 or denormal</td>
<td>2.0</td>
<td>1.5</td>
</tr>
</tbody>
</table>
Usage

The Newton-Raphson iteration:

\[ x_{n+1} = x_n (2 - dx_n) \]

converges to \((1/d)\) if \(x_0\) is the result of \texttt{VRECPE} applied to \(d\).

The Newton-Raphson iteration:

\[ x_{n+1} = x_n (3 - dx_n^2) / 2 \]

converges to \((1/\sqrt{d})\) if \(x_0\) is the result of \texttt{VRSQRTE} applied to \(d\).
5.11 NEON multiply instructions

This section contains the following subsections:

- **VMUL{L}, VMLA{L}, and VMLS{L}** on page 5-68.  
  Vector Multiply, Multiply Accumulate, and Multiply Subtract.

- **VMUL{L}, VMLA{L}, and VMLS{L} (by scalar)** on page 5-69.  
  Vector Multiply, Multiply Accumulate, and Multiply Subtract (by scalar).

- **VQDMULL, VQDMLAL, and VQDMLSL (by vector or by scalar)** on page 5-70  
  Vector Saturating Doubling Multiply, Multiply Accumulate, and Multiply Subtract (by vector or scalar).

- **VQ[R]DMULH (by vector or by scalar)** on page 5-71  
  Vector Saturating Doubling Multiply returning High half (by vector or scalar).
5.11.1 VMUL{L}, VMLA{L}, and VMLS{L}

VMUL (Vector Multiply) multiplies corresponding elements in two vectors, and places the results in the destination vector.

VMLA (Vector Multiply Accumulate) multiplies corresponding elements in two vectors, and accumulates the results into the elements of the destination vector.

VMLS (Vector Multiply Subtract) multiplies corresponding elements in two vectors, subtracts the results from corresponding elements of the destination vector, and places the final results in the destination vector.

Syntax

\[
\text{Vop\{cond\}.datatype \{Qd\}, Qn, Qm}
\]

\[
\text{Vop\{cond\}.datatype \{Dd\}, Dn, Dm}
\]

\[
\text{VopL\{cond\}.datatype Qd, Dn, Dm}
\]

where:

- \(op\) must be one of:
  - MUL Multiply
  - MLA Multiply Accumulate
  - MLS Multiply Subtract.

- \(cond\) is an optional condition code (see Condition codes on page 5-12).

- \(datatype\) must be one of:
  - I8, I16, I32, F32 for MUL, MLA, or MLS
  - S8, S16, S32 for MULL, MLAL, or MLSL
  - U8, U16, U32 for MULL, MLAL, or MLSL
  - P8 for MUL or MULL.

See Polynomial arithmetic over \{0,1\} on page 5-18 for information about datatype P8.

- \(Qd, Qn, Qm\) are the destination vector, the first operand vector, and the second operand vector, for a quadword operation.

- \(Dd, Dn, Dm\) are the destination vector, the first operand vector, and the second operand vector, for a doubleword operation.

- \(Qd, Dn, Dm\) are the destination vector, the first operand vector, and the second operand vector, for a long operation.
5.11.2 VMUL{L}, VMLA{L}, and VMLS{L} (by scalar)

VMUL (Vector Multiply by scalar) multiplies each element in a vector by a scalar, and places the results in the destination vector.

VMLA (Vector Multiply Accumulate) multiplies each element in a vector by a scalar, and accumulates the results into the corresponding elements of the destination vector.

VMLS (Vector Multiply Subtract) multiplies each element in a vector by a scalar, and subtracts the results from the corresponding elements of the destination vector, and places the final results in the destination vector.

Syntax

\[
\text{V}op\{cond\}.\text{datatype}\{Qd, Qn, Dm[x]\}
\]

\[
\text{V}op\{cond\}.\text{datatype}\{Dd, Dn, Dm[x]\}
\]

\[
\text{V}opL\{cond\}.\text{datatype}\{Qd, Dn, Dm[x]\}
\]

where:

- \(op\) must be one of:
  - MUL Multiply
  - MLA Multiply Accumulate
  - MLS Multiply Subtract.

- \(cond\) is an optional condition code (see Condition codes on page 5-12).

- \(datatype\) must be one of:
  - I16, I32, F32 for MUL, MLA, or MLS
  - S16, S32 for MULL, MLAL, or MLSL
  - U16, U32 for MULL, MLAL, or MLSL.

- \(Qd, Qn\) are the destination vector and the first operand vector, for a quadword operation.

- \(Dd, Dn\) are the destination vector and the first operand vector, for a doubleword operation.

- \(Qd, Dn\) are the destination vector and the first operand vector, for a long operation.

- \(Dm[x]\) is the scalar holding the second operand.
5.11.3 VQDMULL, VQDMLAL, and VQDMLSL (by vector or by scalar)

Vector Saturating Doubling Multiply instructions multiply their operands and double the results. VQDMULL places the results in the destination register. VQDMLAL adds the results to the values in the destination register. VQDMLSL subtracts the results from the values in the destination register.

If any of the results overflow, they are saturated. The sticky QC flag (FPSCR bit[27]) is set if saturation occurs.

Syntax

VQD<op>{cond}.datatype Qd, Dn, Dm
VQD<op>{cond}.datatype Qd, Dn, Dm[x]

where:

<op> must be one of:

MUL Multiply
MLA Multiply Accumulate
MLS Multiply Subtract.

<cond> is an optional condition code (see Condition codes on page 5-12).

datatype must be either S16 or S32.

Qd, Dn are the destination vector and the first operand vector.

Dm is the vector holding the second operand, for a by vector operation.

Dm[x] is the scalar holding the second operand, for a by scalar operation.
5.11.4 **VQ(R)DMULH (by vector or by scalar)**

Vector Saturating Doubling Multiply instructions multiply their operands and double the results. They return only the high half of the results.

If any of the results overflow, they are saturated. The sticky QC flag (FPSCR bit[27]) is set if saturation occurs.

**Syntax**

VQ(R)DMULH\{cond\}.datatype {Qd}, Qn, Qm

VQ(R)DMULH\{cond\}.datatype {Dd}, Dn, Dm

VQ(R)DMULH\{cond\}.datatype {Qd}, Qn, Dm[x]

VQ(R)DMULH\{cond\}.datatype {Dd}, Dn, Dm[x]

where:

- **R** if present, indicates that each result is rounded. Otherwise, each result is truncated.
- **cond** is an optional condition code (see *Condition codes* on page 5-12).
- **datatype** must be either S16 or S32.
- **Qd, Qn** are the destination vector and the first operand vector, for a quadword operation.
- **Dd, Dn** are the destination vector and the first operand vector, for a doubleword operation.
- **Qm or Dm** is the vector holding the second operand, for a *by vector* operation.
- **Dm[x]** is the scalar holding the second operand, for a *by scalar* operation.
5.12 NEON load / store element and structure instructions

This section contains the following subsections:

- Interleaving.
- Alignment restrictions in load and store, element and structure instructions on page 5-73.
- VLDn and VSTn (single n-element structure to one lane) on page 5-74. This is used for almost all data accesses. A normal vector can be loaded \((n = 1)\).
- VLDn (single n-element structure to all lanes) on page 5-76.
- VLDn and VSTn (multiple n-element structures) on page 5-78.

5.12.1 Interleaving

Many instructions in this group provide interleaving when structures are stored to memory, and de-interleaving when structures are loaded from memory. Figure 5-8 shows an example of de-interleaving. Interleaving is the inverse process.

![Diagram showing de-interleaving an array of 3-element structures](image)
5.12.2 Alignment restrictions in load and store, element and structure instructions

Many of these instructions allow memory alignment restrictions to be specified. When the alignment is not specified in the instruction, the alignment restriction is controlled by the A bit (CP15 register 1 bit[1]):

- if the A bit is 0, there are no alignment restrictions (except for strongly ordered or device memory, where accesses must be element aligned or the result is unpredictable)
- if the A bit is 1, accesses must be element aligned.

If an address is not correctly aligned, an alignment fault occurs.
5.12.3 VLDn and VSTn (single n-element structure to one lane)

Vector Load single n-element structure to one lane. It loads one n-element structure from memory into one or more NEON registers. Elements of the register that are not loaded are unaltered.

Vector Store single n-element structure to one lane. It stores one n-element structure into memory from one or more NEON registers.

Syntax

\[ V_{opn}\{cond\}.datatype\ list, [Rn@align]!\]
\[ V_{opn}\{cond\}.datatype\ list, [Rn@align], Rm \]

where:

- \( op \) must be either LD or ST.
- \( n \) must be one of 1, 2, 3, or 4.
- \( cond \) is an optional condition code (see Condition codes on page 5-12).
- \( datatype \) see Table 5-12 on page 5-75.
- \( list \) specifies the NEON register list. See Table 5-12 on page 5-75 for options.
- \( Rn \) is the ARM register containing the base address. \( Rn \) cannot be r15.
- \( align \) specifies an optional alignment. See Table 5-12 on page 5-75 for options.
- \(!\) if ! is present, \( Rn \) is updated to \((Rn + \) the number of bytes transferred by the instruction). The update occurs after all the loads or stores have taken place.
- \( Rm \) is an ARM register containing an offset from the base address. If \( Rm \) is present, \( Rn \) is updated to \((Rn + Rm)\) after the address is used to access memory. \( Rm \) cannot be r13 or r15.
Table 5-12 Permitted combinations of parameters

<table>
<thead>
<tr>
<th>n</th>
<th>datatype</th>
<th>list a</th>
<th>align b</th>
<th>alignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8</td>
<td>{Dd[x]}</td>
<td>-</td>
<td>Standard only</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>{Dd[x]}</td>
<td>016</td>
<td>2-byte</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>{Dd[x]}</td>
<td>032</td>
<td>4-byte</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>{Dd[x], D(d+1)[x]}</td>
<td>016</td>
<td>2-byte</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>{Dd[x], D(d+1)[x]}</td>
<td>032</td>
<td>4-byte</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{Dd[x], D(d+2)[x]}</td>
<td>032</td>
<td>4-byte</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>{Dd[x], D(d+1)[x]}</td>
<td>064</td>
<td>8-byte</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{Dd[x], D(d+2)[x]}</td>
<td>064</td>
<td>8-byte</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>{Dd[x], D(d+1)[x], D(d+2)[x]}</td>
<td>-</td>
<td>Standard only</td>
</tr>
<tr>
<td></td>
<td>16 or 32</td>
<td>{Dd[x], D(d+1)[x], D(d+2)[x]}</td>
<td>-</td>
<td>Standard only</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{Dd[x], D(d+2)[x], D(d+4)[x]}</td>
<td>-</td>
<td>Standard only</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>{Dd[x], D(d+1)[x], D(d+2)[x], D(d+3)[x]}</td>
<td>032</td>
<td>4-byte</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>{Dd[x], D(d+1)[x], D(d+2)[x], D(d+3)[x]}</td>
<td>064</td>
<td>8-byte</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{Dd[x], D(d+2)[x], D(d+4)[x], D(d+6)[x]}</td>
<td>064</td>
<td>8-byte</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>{Dd[x], D(d+1)[x], D(d+2)[x], D(d+3)[x]}</td>
<td>064 or 0128</td>
<td>8-byte or 16-byte</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{Dd[x], D(d+2)[x], D(d+4)[x], D(d+6)[x]}</td>
<td>064 or 0128</td>
<td>8-byte or 16-byte</td>
</tr>
</tbody>
</table>

a. Every register in the list must be in the range D0-D31.
b. Align can be omitted. In this case, standard alignment rules apply, see Alignment restrictions in load and store, element and structure instructions on page 5-73.
5.12.4 \textbf{VLDn (single }n\text{-element structure to all lanes)}

Vector Load single }n\text{-element structure to all lanes. It loads multiple copies of one }n\text{-element structure from memory into one or more NEON registers.}

\textbf{Syntax}

\begin{itemize}
\item \texttt{VLDn\{cond\}.datatype list, [Rn[@align]]\{!\}}
\item \texttt{VLDn\{cond\}.datatype list, [Rn[@align]], Rm}
\end{itemize}

where:

- \texttt{n} must be one of 1, 2, 3, or 4.
- \texttt{cond} is an optional condition code (see \textit{Condition codes} on page 5-12).
- \texttt{datatype} see Table 5-13 on page 5-77.
- \texttt{list} specifies the NEON register list. See Table 5-13 on page 5-77 for options.
- \texttt{Rn} is the ARM register containing the base address. \texttt{Rn} cannot be r15.
- \texttt{align} specifies an optional alignment. See Table 5-13 on page 5-77 for options.
- \texttt{!} if \texttt{!} is present, \texttt{Rn} is updated to \((\texttt{Rn} + \text{the number of bytes transferred by the instruction})\). The update occurs after all the loads or stores have taken place.
- \texttt{Rm} is an ARM register containing an offset from the base address. If \texttt{Rm} is present, \texttt{Rn} is updated to \((\texttt{Rn} + \texttt{Rm})\) \textit{after} the address is used to access memory. \texttt{Rm} cannot be r13 or r15.
### Table 5-13 Permitted combinations of parameters

<table>
<thead>
<tr>
<th>n</th>
<th>datatype</th>
<th>list a</th>
<th>align b</th>
<th>alignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8</td>
<td>{Dd[]}</td>
<td>-</td>
<td>Standard only</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{Dd[]}, D(d+1)[]</td>
<td>-</td>
<td>Standard only</td>
</tr>
<tr>
<td>16</td>
<td>8</td>
<td>{Dd[]}</td>
<td>016</td>
<td>2-byte</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{Dd[]}, D(d+1)[]</td>
<td>016</td>
<td>2-byte</td>
</tr>
<tr>
<td>32</td>
<td>8</td>
<td>{Dd[]}</td>
<td>032</td>
<td>4-byte</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{Dd[]}, D(d+1)[]</td>
<td>032</td>
<td>4-byte</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>{Dd[]}, D(d+1)[]</td>
<td>08</td>
<td>byte</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{Dd[]}, D(d+2)[]</td>
<td>08</td>
<td>byte</td>
</tr>
<tr>
<td>16</td>
<td></td>
<td>{Dd[]}, D(d+1)[]</td>
<td>016</td>
<td>2-byte</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{Dd[]}, D(d+2)[]</td>
<td>016</td>
<td>2-byte</td>
</tr>
<tr>
<td>32</td>
<td></td>
<td>{Dd[]}, D(d+1)[]</td>
<td>032</td>
<td>4-byte</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{Dd[]}, D(d+2)[]</td>
<td>032</td>
<td>4-byte</td>
</tr>
<tr>
<td>3</td>
<td>8, 16, or 32</td>
<td>{Dd[]}, D(d+1)[], D(d+2)[]</td>
<td>-</td>
<td>Standard only</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{Dd[]}, D(d+2)[][], D(d+4)[]</td>
<td>-</td>
<td>Standard only</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>{Dd[]}, D(d+1)[][], D(d+2)[][], D(d+3)[]</td>
<td>032</td>
<td>4-byte</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{Dd[]}, D(d+2)[][], D(d+4)[][], D(d+6)[]</td>
<td>032</td>
<td>4-byte</td>
</tr>
<tr>
<td>16</td>
<td></td>
<td>{Dd[]}, D(d+1)[][], D(d+2)[][], D(d+3)[]</td>
<td>064</td>
<td>8-byte</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{Dd[]}, D(d+2)[][], D(d+4)[][], D(d+6)[]</td>
<td>064</td>
<td>8-byte</td>
</tr>
<tr>
<td>32</td>
<td></td>
<td>{Dd[]}, D(d+1)[][], D(d+2)[][], D(d+3)[]</td>
<td>064 or 0128</td>
<td>8-byte or 16-byte</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{Dd[]}, D(d+2)[][], D(d+4)[][], D(d+6)[]</td>
<td>064 or 0128</td>
<td>8-byte or 16-byte</td>
</tr>
</tbody>
</table>

a. Every register in the list must be in the range D0-D31.
b. Align can be omitted. In this case, standard alignment rules apply, see Alignment restrictions in load and store, element and structure instructions on page 5-73.
5.12.5  VLDn and VSTn (multiple n-element structures)

Vector Load multiple n-element structures. It loads multiple n-element structures from memory into one or more NEON registers, with de-interleaving (unless n == 1). Every element of each register is loaded.

Vector Store multiple n-element structures. It stores multiple n-element structures to memory from one or more NEON registers, with interleaving (unless n == 1). Every element of each register is stored.

Syntax

\[ \text{V} opn \{\text{cond}\}.\text{datatype list}, [Rn[@align]]\{!\} \]
\[ \text{V} opn\{\text{cond}\}.\text{datatype list}, [Rn[@align]], Rm \]

where:

- **op** must be either LD or ST.
- **n** must be one of 1, 2, 3, or 4.
- **cond** is an optional condition code (see Condition codes on page 5-12).
- **datatype** see Table 5-14 on page 5-79 for options.
- **list** specifies the NEON register list. See Table 5-14 on page 5-79 for options.
- **Rn** is the ARM register containing the base address. **Rn** cannot be r15.
- **align** specifies an optional alignment. See Table 5-14 on page 5-79 for options.
- **!** if ! is present, **Rn** is updated to (\(Rn + \) the number of bytes transferred by the instruction). The update occurs after all the loads or stores have taken place.
- **Rm** is an ARM register containing an offset from the base address. If **Rm** is present, **Rn** is updated to (\(Rn + Rm\) after the address is used to access memory. **Rm** cannot be r13 or r15.
### Table 5-14 Permitted combinations of parameters

<table>
<thead>
<tr>
<th>n</th>
<th>datatype</th>
<th>list a</th>
<th>align b</th>
<th>alignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8, 16, 32, or 64</td>
<td>{Dd}</td>
<td>064</td>
<td>8-byte</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{Dd, D(d+1)}</td>
<td>064 or 0128</td>
<td>8-byte or 16-byte</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{Dd, D(d+1), D(d+2)}</td>
<td>064</td>
<td>8-byte</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{Dd, D(d+1), D(d+2), D(d+3)}</td>
<td>064, 0128, or 0256</td>
<td>8-byte, 16-byte, or 32-byte</td>
</tr>
<tr>
<td>2</td>
<td>8, 16, or 32</td>
<td>{Dd, D(d+1)}</td>
<td>064, 0128</td>
<td>8-byte or 16-byte</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{Dd, D(d+2)}</td>
<td>064, 0128</td>
<td>8-byte or 16-byte</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{Dd, D(d+1), D(d+2), D(d+3)}</td>
<td>064, 0128, or 0256</td>
<td>8-byte, 16-byte, or 32-byte</td>
</tr>
<tr>
<td>3</td>
<td>8, 16, or 32</td>
<td>{Dd, D(d+1), D(d+2)}</td>
<td>064</td>
<td>8-byte</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{Dd, D(d+2), D(d+4)}</td>
<td>064</td>
<td>8-byte</td>
</tr>
<tr>
<td>4</td>
<td>8, 16, or 32</td>
<td>{Dd, D(d+1), D(d+2), D(d+3)}</td>
<td>064, 0128, or 0256</td>
<td>8-byte, 16-byte, or 32-byte</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{Dd, D(d+2), D(d+4), D(d+6)}</td>
<td>064, 0128, or 0256</td>
<td>8-byte, 16-byte, or 32-byte</td>
</tr>
</tbody>
</table>

a. Every register in the list must be in the range D0-D31.
b. Align can be omitted. In this case, standard alignment rules apply, see Alignment restrictions in load and store, element and structure instructions on page 5-73.
5.13 NEON and VFP pseudo-instructions

This section contains the following subsections:

- *VLDR pseudo-instruction* on page 5-81 (NEON and VFP)
- *VLDR and VSTR (post-increment and pre-decrement)* on page 5-82 (NEON and VFP)
- *VMOV2* on page 5-83 (NEON only)
- *VAND and VORN (immediate)* on page 5-84 (NEON only)
- *VACLE and VACLT* on page 5-85 (NEON only)
- *VCLE and VCLT* on page 5-86 (NEON only).
5.13.1 VLDR pseudo-instruction

The VLDR pseudo-instruction loads a constant value into every element of a 64-bit NEON vector, or into a VFP single-precision or double-precision register.

--- Note ---
This section describes the VLDR pseudo-instruction only. See VLDR and VSTR on page 5-21 for information on the VLDR instruction.

Syntax

VLDR{cond}.datatype Dd,=constant
VLDR{cond}.datatype Sd,=constant

where:

datatype must be one of:

<table>
<thead>
<tr>
<th>In</th>
<th>NEON only</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sn</td>
<td>NEON only</td>
</tr>
<tr>
<td>Un</td>
<td>NEON only</td>
</tr>
<tr>
<td>F32</td>
<td>NEON or VFP</td>
</tr>
<tr>
<td>F64</td>
<td>VFP only</td>
</tr>
</tbody>
</table>

n must be one of 8, 16, 32, or 64.

cond is an optional condition code (see Condition codes on page 5-12).

Dd or Sd is the extension register to be loaded.

constant is a constant of the appropriate type for datatype.

Usage

If an instruction (for example, VMOV) is available that can generate the constant directly into the register, the assembler uses it. Otherwise, it generates a doubleword literal pool entry containing the constant and loads the constant using a VLDR instruction.
5.13.2 VLDR and VSTR (post-increment and pre-decrement)

Pseudo-instructions that load or store extension registers with post-increment and pre-decrement.

--- Note ---
See VLDR and VSTR on page 5-21 for information on the VLDR and VSTR instructions without post-increment and pre-decrement.

**Syntax**

```plaintext
op{cond}{.size} Fd, [Rn], #offset ; post-increment
op{cond}{.size} Fd, [Rn, #-offset]! ; pre-decrement
```

where:

- **op** can be:
  - VLDR - load extension register from memory
  - VSTR - store contents of extension register to memory.
- **cond** is an optional condition code (see Condition codes on page 5-12).
- **size** is an optional data size specifier. Must be 32 if Fd is a single precision VFP register, or 64 if Fd is a double precision register.
- **Fd** is the extension register to be loaded or saved. For a NEON instruction, it must be a double precision (Dd) register. For a VFP instruction, it can be either a double precision (Dd) or a single precision (Sd) register.
- **Rn** is the ARM register holding the base address for the transfer.
- **offset** is a numeric expression that must evaluate to a numeric constant at assembly time. The value must be 4 if Fd is a single precision VFP register, or 8 if Fd is a double precision register.

**Usage**

The post-increment instruction increments the base address in the register by the offset value, after the transfer. The pre-decrement instruction decrements the base address in the register by the offset value, and then performs the transfer using the new address in the register. These pseudo-instructions assemble to VLDM or VSTM instructions (see VLDM, VSTM, VPOP, and VPUSH on page 5-22).
5.13.3 VMOV2

The VMOV2 pseudo-instruction generates a constant and places it in every element of a NEON vector, without loading a value from a literal pool. It always assembles to exactly two instructions.

VMOV2 can generate any 16-bit constant, and a restricted range of 32-bit and 64-bit constants.

Syntax

VMOV2{cond}.datatype Qd, #constant
VMOV2{cond}.datatype Dd, #constant

where:

datatype must be one of:
- I8, I16, I32, or I64
- S8, S16, S32, or S64
- U8, U16, U32, or U64
- F32.

cond is an optional condition code (see Condition codes on page 5-12).

Qd or Dd is the extension register to be loaded.

constant is a constant of the appropriate type for datatype.

Usage

VMOV2 typically assembles to a VMOV or VMVN instruction, followed by a VBIC or VORR instruction. See VMOV, VMVN (immediate) on page 5-41 and VBIC and VORR (immediate) on page 5-30 for details.
5.13.4 VAND and VORN (immediate)

VAND (Bitwise AND immediate) takes each element of the destination vector, performs a bitwise AND with an immediate constant, and returns the result into the destination vector.

VORN (Bitwise OR NOT immediate) takes each element of the destination vector, performs a bitwise OR Complement with an immediate constant, and returns the result into the destination vector.

**Note**
On disassembly, these pseudo-instructions are disassembled to the corresponding VBIC and VORR instructions, with the complementary immediate constants.

**Syntax**

\[ \text{Vop}\{\text{cond}\}.\text{datatype}\ Qd, \#imm \]
\[ \text{Vop}\{\text{cond}\}.\text{datatype}\ Dd, \#imm \]

where:

- **op** must be either VAND or VORN.
- **cond** is an optional condition code (see *Condition codes* on page 5-12).
- **datatype** must be either I16, I16, I32, or I64.
- **Qd** or **Dd** is the NEON register for the result.
- **imm** is the immediate constant.

**Immediate constants**

If **datatype** is I16, the immediate constant must have one of the following forms:

- 0xFFFXY
- 0xFFFFYFF.

If **datatype** is I32, the immediate constant must have one of the following forms:

- 0xFFFFFFFFXY
- 0xFFFFFFFFYFF
- 0xFFFFFFFFFYFF
- 0xFFFFFFFFYFFFF.

See *VBIC and VORR (immediate)* on page 5-30 for more information.
5.13.5 VACLE and VACLT

Vector Absolute Compare takes the absolute value of each element in a vector, and compares it with the absolute value of the corresponding element of a second vector. If the condition is true, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

Note

On disassembly, these pseudo-instructions are disassembled to the corresponding VACGE and VACGT instructions, with the operands reversed.

Syntax

VACop{cond}.datatype {Qd}, Qn, Qm
VACop{cond}.datatype {Dd}, Dn, Dm

where:

- \textit{op} must be one of:
  - LE: Absolute Less than or Equal
  - LT: Absolute Less Than.

- \textit{cond} is an optional condition code (see Condition codes on page 5-12).

- \textit{datatype} must be F32.

- \textit{Qd} or \textit{Dd} is the NEON register for the result.
  The result datatype is I32.

- \textit{Qn} or \textit{Dn} is the NEON register holding the first operand.

- \textit{Qm} or \textit{Dm} is the NEON register holding the second operand.
Vector Compare takes the value of each element in a vector, and compares it with the value of the corresponding element of a second vector, or zero. If the condition is true, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

Note
On disassembly, these pseudo-instructions are disassembled to the corresponding VCGE and VCGT instructions, with the operands reversed.

Syntax

\[ \text{VC\{cond\}.\text{datatype\} \{Qd\}, Qn, Qm} \]
\[ \text{VC\{cond\}.\text{datatype\} \{Dd\}, Dn, Dm} \]

where:

- \( op \) must be one of:
  - LE  Less than or Equal
  - LT  Less Than.

- \( cond \) is an optional condition code (see Condition codes on page 5-12).

- \( datatype \) must be one of S8, S16, S32, U8, U16, U32, or F32.

- \( Qd \) or \( Dd \) is the NEON register for the result.
  The result datatype is:
  - I32 for operand datatypes I32, S32, U32, or F32
  - I16 for operand datatypes I16, S16, or U16
  - I8 for operand datatypes I8, S8, or U8.

- \( Qn \) or \( Dn \) is the NEON register holding the first operand.

- \( Qm \) or \( Dm \) is the NEON register holding the second operand.
5.14 NEON and VFP system registers

Three NEON and VFP system registers are accessible to you in all implementations of NEON and VFP:

- **FPSCR, the floating-point status and control register**
- **FPEXC, the floating-point exception register** on page 5-90
- **FPSID, the floating-point system ID register** on page 5-90
- **Modifying individual bits of a NEON and VFP system register** on page 5-91.

A particular implementation of NEON or VFP can have additional registers (see the technical reference manual for the VFP coprocessor you are using).

5.14.1 FPSCR, the floating-point status and control register

The FPSCR contains all the user-level NEON and VFP status and control bits. NEON only uses bits[31:27]. The bits are used as follows:

- **bits[31:28]** Are the N, Z, C, and V flags. These are the NEON and VFP status flags. They cannot be used to control conditional execution until they have been copied into the status flags in the CPSR (see Condition codes on page 5-12).

- **bit[27]** Is the QC, cumulative saturation flag. This is set if saturation occurs in NEON saturating instructions.

- **bit[25]** Is the Default NaN (DN) mode control bit:
  
  0  Disabled. NaN operands propagate through to the output of a floating-point operation.
  
  1  Enabled. Any operation involving one or more NaNs returns the Default NaN.

  Note: NEON always uses the Default NaN enabled setting regardless of this bit.

- **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 5-92).

  Note: NEON always uses flush-to-zero mode regardless of this bit.
Flush-to-zero mode must not be used when IEEE 754 compatibility is a requirement.

**bits[23:22]** Control rounding mode as follows:
- 0b00  \textit{Round to Nearest} (RN) mode.
- 0b01  \textit{Round towards Plus infinity} (RP) mode.
- 0b10  \textit{Round towards Minus infinity} (RM) mode.
- 0b11  \textit{Round towards Zero} (RZ) mode.

**Note**
NEON always uses the Round to Nearest mode regardless of these bits.

**bits[21:20]** STRIDE is the distance between successive values in a vector (see \textit{Vectors} on page 5-105). Stride is controlled as follows:
- 0b00  \texttt{STRIDE = 1}
- 0b11  \texttt{STRIDE = 2}.

**bits[18:16]** LEN is the number of registers used by each vector (see \textit{Vectors} on page 5-105). It is 1 + the value of bits[18:16]:
- 0b000  \texttt{LEN = 1}
- ...
- 0b111  \texttt{LEN = 8}.

**bits[15, 12:8]** Are the exception trap enable bits:
- IDE  input denormal exception enable
- 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[7, 4:0]** Are the cumulative exception bits:
- IDC  input denormal exception
- 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 NEON and 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 NEON and VFP system register* on page 5-91).

**Note**

The use of vector mode is deprecated. Set LEN and STRIDE to 1.
5.14.2 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 NEON or 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 NEON or 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 NEON or VFP implementations.

If the value is 1, NEON (if present) and VFP (if present) are enabled and operate normally.

If the value is 0, NEON and VFP are disabled. When they are disabled, you can read or write the FPSID or FPEXC registers, but other NEON or 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 NEON and VFP system register on page 5-91).

5.14.3 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 NEON or VFP architecture your program is running on.
5.14.4 Modifying individual bits of a NEON and VFP system register

To change some bits of a NEON and VFP system register without affecting other bits, use a read-modify-write procedure similar to the following example:

\[
\begin{align*}
\text{VMRS} & \quad \text{r10,FPSCR} \quad ; \text{copy FPSCR into r10} \\
\text{BIC} & \quad \text{r10,r10,#0x00370000} \quad ; \text{clears STRIDE and LEN} \\
\text{ORR} & \quad \text{r10,r10,#0x00030000} \quad ; \text{sets STRIDE = 1, LEN = 4} \\
\text{VMSR} & \quad \text{FPSCR,r10} \quad ; \text{copy r10 back into FPSCR}
\end{align*}
\]

See \textit{VMRS and VMSR} on page 5-27.
5.15 **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.

NEON and VFPv3 flush-to-zero preserves the sign bit. VFPv2 flush-to-zero flushes to +0.

NEON always uses flush-to-zero mode.

5.15.1 **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.

5.15.2 **The effects of using flush-to-zero mode**

With certain exceptions (see *Operations not affected by flush-to-zero mode* on page 5-93), 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.

5.15.3 Operations not affected by flush-to-zero mode

The following NEON and VFP 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 VMOV, VMVN (register) on page 5-32, VABS, VNNEG, and VSQRT on page 5-95, and V[Q]ABS and V[Q]NEG on page 5-56).
- Duplicate (see VDUP on page 5-39).
- Swap (see VSWP on page 5-44).
- Load and store (see VLDR and VSTR on page 5-21).
- Load multiple and store multiple (see VLDM, VSTM, VPOP, and VPUSH on page 5-22).
- Transfer between extension registers and ARM general-purpose registers (see VMOV (between two ARM registers and an extension register) on page 5-24, VMOV (between an ARM register and a NEON scalar) on page 5-25, and VMOV (between one ARM register and single precision VFP) on page 5-26).
5.16 VFP instructions

This section contains the following subsections:

- **VABS, VNEG, and VSQRT** on page 5-95
  Floating-point absolute value, negate, and square root.

- **VADD, VSUB, and VDIV** on page 5-96
  Floating-point add, subtract, and divide.

- **VMUL, VMLA, VMLS, VNMLU, VNMLA, and VNMLS** on page 5-97
  Floating-point multiply and multiply accumulate, with optional negation.

- **VCMP** on page 5-98
  Floating-point compare.

- **VCVT (between single-precision and double-precision)** on page 5-99
  Convert between single-precision and double-precision.

- **VCVT (between floating-point and integer)** on page 5-100
  Convert between floating-point and integer.

- **VCVT (between floating-point and fixed-point)** on page 5-101
  Convert between floating-point and fixed-point.

- **VCVTB, VCVTT (half-precision extension)** on page 5-102
  Convert between half-precision and single-precision floating-point.

- **VMOV** on page 5-103
  Insert a floating-point constant in a single-precision or double-precision register.
5.16.1 VABS, VNEG, and VSQRT

Floating-point absolute value, negate, and square root.

These instructions can be scalar, vector, or mixed (see VFP vector and scalar operations on page 5-106).

**Syntax**

\[
Vop\{\text{cond}\}.F32 \quad Sd, \quad Sm
\]

\[
Vop\{\text{cond}\}.F64 \quad Dd, \quad Dm
\]

where:

- \( op \) is one of \text{ABS}, \text{NEG}, or \text{SQRT}.
- \( cond \) is an optional condition code (see Condition codes on page 5-12).
- \( Sd, \quad Sm \) are the single-precision registers for the result and operand.
- \( Dd, \quad Dm \) are the double-precision registers for the result and operand.

**Usage**

The \text{VABS} instruction takes the contents of \( Sm \) or \( Dm \), clears the sign bit, and places the result in \( Sd \) or \( Dd \). This gives the absolute value.

The \text{VNEG} instruction takes the contents of \( Sm \) or \( Dm \), changes the sign bit, and places the result in \( Sd \) or \( Dd \). This gives the negation of the value.

The \text{VSQRT} instruction takes the square root of the contents of \( Sm \) or \( Dm \), and places the result in \( Sd \) or \( Dd \).

In the case of a \text{VABS} and \text{VNEG} instruction, if the operand is a NaN, the sign bit is determined in each case as above, but no exception is produced.

**Floating-point exceptions**

\text{VABS} and \text{VNEG} instructions cannot produce any exceptions.

\text{VSQRT} instructions can produce Invalid Operation or Inexact exceptions.
5.16.2 VADD, VSUB, and VDIV

Floating-point add, subtract, and divide.

These instructions can be scalar, vector, or mixed (see VFP vector and scalar operations on page 5-106).

Syntax

\[ \text{V} \text{op}\{\text{cond}\}.\text{F32} \{\text{Sd}, \text{Sn}, \text{Sm}\} \]
\[ \text{V} \text{op}\{\text{cond}\}.\text{F64} \{\text{Dd}, \text{Dn}, \text{Dm}\} \]

where:

\text{op} is one of ADD, SUB, or DIV.
\text{cond} is an optional condition code (see Condition codes on page 5-12).
\text{Sd}, \text{Sn}, \text{Sm} are the single-precision registers for the result and operands.
\text{Dd}, \text{Dn}, \text{Dm} are the double-precision registers for the result and operands.

Usage

The VADD instruction adds the values in the operand registers and places the result in the destination register.

The VSUB instruction subtracts the value in the second operand register from the value in the first operand register, and places the result in the destination register.

The VDIV instruction divides the value in the first operand register by the value in the second operand register, and places the result in the destination register.

Floating-point exceptions

VADD and VSUB instructions can produce Invalid Operation, Overflow, or Inexact exceptions.

VDIV operations can produce Division by Zero, Invalid Operation, Overflow, Underflow, or Inexact exceptions.
5.16.3 VMUL, VMLA, VMLS, VN MUL, VNMLA, and VNMLS

Floating-point multiply and multiply accumulate, with optional negation.

These instructions can be scalar, vector, or mixed (see VFP vector and scalar operations on page 5-106).

Syntax

\[
\begin{align*}
V\{N\}MUL\{\text{cond}\}.F32 & \{Sd,\} \ Sn, \ Sm \\
V\{N\}MUL\{\text{cond}\}.F64 & \{Dd,\} \ Dn, \ Dm \\
V\{N\}MLA\{\text{cond}\}.F32 & \ Sd, \ Sn, \ Sm \\
V\{N\}MLA\{\text{cond}\}.F64 & \ Dd, \ Dn, \ Dm \\
V\{N\}MLS\{\text{cond}\}.F32 & \ Sd, \ Sn, \ Sm \\
V\{N\}MLS\{\text{cond}\}.F64 & \ Dd, \ Dn, \ Dm
\end{align*}
\]

where:

- \( N \) negates the final result.
- \( \text{cond} \) is an optional condition code (see Condition codes on page 5-12).
- \( Sd, \ Sn, \ Sm \) are the single-precision registers for the result and operands.
- \( Dd, \ Dn, \ Dm \) are the double-precision registers for the result and operands.

Usage

The MUL operation multiplies the values in the operand registers and places the result in the destination register.

The MLA operation multiplies the values in the operand registers, adds the value in the destination register, and places the final result in the destination register.

The MLS operation multiplies the values in the operand registers, subtracts the result from the value in the destination register, and places the final result in the destination register.

In each case, the final result is negated if the \( N \) option is used.

Floating-point exceptions

These instructions can produce Invalid Operation, Overflow, Underflow, Inexact, or Input Denormal exceptions.
5.16.4 VCMP

Floating-point compare.

VCMP is always scalar.

Syntax

VCMP{cond}.F32  Sd, Sm
VCMP{cond}.F32  Sd, #0
VCMP{cond}.F64  Dd, Dm
VCMP{cond}.F64  Dd, #0

where:

cond is an optional condition code (see Condition codes on page 5-12).
Sd, Sm are the single-precision registers holding the operands.
Dd, Dm are the double-precision registers holding the operands.

Usage

The VCMP instruction subtracts the value in the second operand register (or 0 if the second operand is #0) from the value in the first operand register, and sets the VFP condition flags on the result (see Condition codes on page 5-12).

Floating-point exceptions

VCMP instructions can produce Invalid Operation exceptions.
5.16.5 VCVT (between single-precision and double-precision)

Convert between single-precision and double-precision numbers.

VCVT is always scalar.

Syntax

VCVT{cond}.F64.F32  Dd, Sm
VCVT{cond}.F32.F64  Sd, Dm

where:

cond is an optional condition code (see Condition codes on page 5-12).

Dd is a double-precision register for the result.

Sm is a single-precision register holding the operand.

Sd is a single-precision register for the result.

Dm is a double-precision register holding the operand.

Usage

These instructions convert the single-precision value in Sm to double-precision and places the result in Dd, or the double-precision value in Dm to single-precision and place the result in Sd.

Floating-point exceptions

These instructions can produce Invalid Operation, Input Denormal, Overflow, Underflow, or Inexact exceptions.
5.16.6 VCVT (between floating-point and integer)

Convert between floating-point numbers and integers.

VCVT is always scalar.

Syntax

VCVT{R}{cond}.type.F64 Sd, Dm
VCVT{R}{cond}.type.F32 Sd, Sm
VCVT{cond}.F64.type Dd, Sm
VCVT{cond}.F32.type Sd, Sm

where:

R makes the operation use the rounding mode specified by the FPSCR. Otherwise, the operation rounds towards zero.

cond is an optional condition code (see Condition codes on page 5-12).

type can be either U32 (unsigned 32-bit integer) or S32 (signed 32-bit integer).

Sd is a single-precision register for the result.

Dd is a double-precision register for the result.

Sm is a single-precision register holding the operand.

Dm is a double-precision register holding the operand.

Usage

The first two forms of this instruction convert from floating-point to integer.

The third and fourth forms convert from integer to floating-point.

Floating-point exceptions

These instructions can produce Input Denormal, Invalid Operation, or Inexact exceptions.
5.16.7 VCVT (between floating-point and fixed-point)

Convert between floating-point and fixed-point numbers.

VCVT is always scalar.

Syntax

\[
\begin{align*}
& \text{VCVT}\{\text{cond}\}.\text{type.F64 } Dd, Dd, \#fbits \\
& \text{VCVT}\{\text{cond}\}.\text{type.F32 } Sd, Sd, \#fbits \\
& \text{VCVT}\{\text{cond}\}.\text{type.F64 } Dd, Dd, \#fbits \\
& \text{VCVT}\{\text{cond}\}.\text{type.F32 } Sd, Sd, \#fbits
\end{align*}
\]

where:

- \( \text{cond} \) is an optional condition code (see Condition codes on page 5-12).
- \( \text{type} \) can be any one of:
  - S16 16-bit signed fixed-point number
  - U16 16-bit unsigned fixed-point number
  - S32 32-bit signed fixed-point number
  - U32 32-bit unsigned fixed-point number.
- \( Sd \) is a single-precision register for the operand and result.
- \( Dd \) is a double-precision register for the operand and result.
- \( \#fbits \) is the number of fraction bits in the fixed-point number, in the range 0-16 if \( \text{type} \) is S16 or U16, or in the range 1-32 if \( \text{type} \) is S32 or U32.

Usage

The first two forms of this instruction convert from floating-point to fixed-point.

The third and fourth forms convert from fixed-point to floating-point.

In all cases the fixed-point number is contained in the least significant 16 or 32 bits of the register.

Floating-point exceptions

These instructions can produce Input Denormal, Invalid Operation, or Inexact exceptions.
5.16.8 VCVTB, VCVTT (half-precision extension)

Converts between half-precision and single-precision floating-point numbers in the following ways:

- **VCVTB** uses the bottom half (bits[15:0]) of the single word register to obtain or store the half-precision value.
- **VCVTT** uses the top half (bits[31:16]) of the single word register to obtain or store the half-precision value.

VCVTB and VCVTT are always scalar.

**Syntax**

VCVTB\{cond\}.type Sd, Sm

VCVTT\{cond\}.type Sd, Sm

where:

- **cond** is an optional condition code (see *Condition codes* on page 5-12).
- **type** can be any one of:
  - F32.F16 convert from half-precision to single-precision
  - F16.F32 convert from single-precision to half-precision.
- **Sd** is a single word register for the result.
- **Sm** is a single word register for the operand.

**Architectures**

The instructions are only available in VFPv3 systems with the half-precision extension.

**Floating-point exceptions**

These instructions can produce Input Denormal, Invalid Operation, Overflow, Underflow, or Inexact exceptions.
5.16.9 VMOV

Insert a floating-point constant in a single-precision or double-precision register, or copy one register into another register.

This instruction is always scalar.

Syntax

\[ \text{VMOV}\{\text{cond}\}.F32 \ Sd, \ #\text{imm} \]
\[ \text{VMOV}\{\text{cond}\}.F64 \ Dd, \ #\text{imm} \]
\[ \text{VMOV}\{\text{cond}\}.F32 \ Sd, \ S\text{m} \]
\[ \text{VMOV}\{\text{cond}\}.F64 \ Dd, \ D\text{m} \]

where:

\[ \text{cond} \] is an optional condition code (see \textit{Condition codes} on page 5-12).

\[ Sd \] is the single-precision destination register.

\[ Dd \] is the double-precision destination register.

\[ \text{imm} \] is the floating-point constant.

\[ S\text{m} \] is the single-precision source register.

\[ D\text{m} \] is the double-precision source register.

Constant values

Any number that can be expressed as \(+/-n \times 2^r\), where \(n\) and \(r\) are integers,
\[ 16 \leq n \leq 31, \ 0 \leq r \leq 7. \]

Architectures

This instruction is available in VFPv3.
5.17 VFP vector mode

Most arithmetic instructions can be used on 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.

For more details of the VFP coprocessor, see the ARM Architecture Reference Manual.

--- Note ---
The use of VFP vector mode is deprecated.

5.17.1 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
- eight (four in VFPv2) banks of four double-precision registers, d0 to d3, d4 to d7, d8 to d11, d12 to d15, d16 to d19, d20 to d23, d24 to d27, and d28 to d31
- any combination of single-precision and double-precision registers.

See Figure 5-9 and Figure 5-10 for further clarification.

--- Figure 5-9 VFPv2 register banks ---

--- Figure 5-10 VFPv3 register banks ---
5.17.2 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 5-87).

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 5-87). 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.
5.17.3 VFP 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 5-87).

When LEN is 1 all operations are scalar.

Vectors can have a stride of 1 or 2, controlled by the STRIDE bits in the FPSCR. When STRIDE is 1, the elements of the vector occupy consecutive registers in the bank. When STRIDE is 2, the elements of the vector occupy alternate registers in the bank.
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 5-104).

Given instructions of the following general forms:

\[
\begin{align*}
\text{Op} & \ Fd, Fn, Fm \\
\text{Op} & \ Fd, Fm
\end{align*}
\]

the behavior is as follows:

- If \( Fd \) is in the first or fifth bank of registers, \( s0 \) to \( s7 \), \( d0 \) to \( d3 \), or \( d16 \) to \( d19 \), the operation is scalar.
- If the \( Fm \) is in the first or fifth bank of registers, but \( Fd \) is not, the operation is mixed.
- If neither \( Fd \) nor \( Fm \) are in the first or fifth bank of registers, the operation is vector.

**Scalar operations**

\( \text{Op} \) acts on the value in \( Fm \), and the value in \( Fn \) if present. The result is placed in \( Fd \).

**Vector operations**

\( \text{Op} \) 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, \( \text{Op} \) acts on the single value in \( Fm \). LEN copies of the result are placed in the vector starting at \( Fd \).

For multiple-operand instructions, \( \text{Op} \) 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 \( Fd \).
5.17.4 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.

The use of VFP vector mode is deprecated, and vector notation is not supported in UAL. To use vector notation, you must use the pre-UAL VFP mnemonics. See Pre-UAL VFP mnemonics on page 5-109 for details. You can mix pre-UAL VFP mnemonics and UAL VFP mnemonics.

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

- VFPASSERT SCALAR on page 5-112
- VFPASSERT VECTOR on page 5-113.

If you use VFPASSERT directives, you must specify vector details in all VFP data processing instructions written using pre-UAL mnemonics. The vector notation is described in Vector notation on page 5-111. If you do not use VFPASSERT directives you must not use this notation.
Pre-UAL VFP mnemonics

Where UAL mnemonics use .F32 to specify single-precision data, pre-UAL mnemonics use S appended to the instruction mnemonic. For example, VABS.32 was FABS.

Where UAL mnemonics use .F64 to specify double-precision data, pre-UAL mnemonics use D appended to the instruction mnemonic. For example, VMPNE.64 was FOMPED.

Table 5-15 shows the pre-UAL mnemonics of those instructions that are affected by VFP vector mode. All other VFP instructions are always scalar regardless of the settings of LEN and STRIDE.

<table>
<thead>
<tr>
<th>UAL mnemonic</th>
<th>Equivalent pre-UAL mnemonic</th>
</tr>
</thead>
<tbody>
<tr>
<td>VABS</td>
<td>FABS</td>
</tr>
<tr>
<td>VADD</td>
<td>FADD</td>
</tr>
<tr>
<td>VDIV</td>
<td>FDIV</td>
</tr>
<tr>
<td>VMLA</td>
<td>FMAC</td>
</tr>
<tr>
<td>VMLS</td>
<td>FNMAC</td>
</tr>
<tr>
<td>VMOV (immediate)</td>
<td>FCONST a</td>
</tr>
<tr>
<td>VMOV (register)</td>
<td>FCPY</td>
</tr>
<tr>
<td>VMUL</td>
<td>FMUL</td>
</tr>
<tr>
<td>VNEG</td>
<td>FNEG</td>
</tr>
<tr>
<td>VNMLA</td>
<td>FNMSC</td>
</tr>
<tr>
<td>VNMLS</td>
<td>FMSC</td>
</tr>
<tr>
<td>VNMLUL</td>
<td>FNMLUL</td>
</tr>
<tr>
<td>VSQRT</td>
<td>FSQRT</td>
</tr>
<tr>
<td>VSUB</td>
<td>FSUB</td>
</tr>
</tbody>
</table>

a. The immediate in VMOV (immediate) is the floating-point number you want to load. The immediate in FCONST is the number encoded in the instruction to produce the floating-point number you want to load. See Immediate values in FCONST on page 5-110 for details.
Immediate values in FCONST

Table 5-16 shows the floating-point constants you can load using FCONST. Trailing zeroes are omitted for clarity. The immediate value you must put in the FCONST instruction is the decimal representation of the binary number abcd efgh, where:

- a is 0 for positive numbers, or 1 for negative numbers
- bcd is shown in the column headings
- efgh is shown in the row headings.

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

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

In pre-UAL VFP data processing instructions, specify vectors of VFP registers using angle brackets:

- \( s_n \) is a single-precision scalar register \( n \)
- \( s_n<> \) is a single-precision vector whose length and stride are given by the current vector length and stride, starting at register \( n \)
- \( s_n<L> \) is a single-precision vector of length \( L \), stride 1, starting at register \( n \)
- \( s_n<L:S> \) is a single-precision vector of length \( L \), stride \( S \), starting at register \( n \)
- \( d_n \) is a double-precision scalar register \( n \)
- \( d_n<> \) is a double-precision vector whose length and stride are given by the current vector length and stride, starting at register \( n \)
- \( d_n<L> \) is a double-precision vector of length \( L \), stride 1, starting at register \( n \)
- \( d_n<L:S> \) 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 \( QN \), DN, and SN on page 7-13).

You must not use this vector notation in the DN and SN directives themselves.
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 Procedure Call Standard for the ARM Architecture specification, aapcs.pdf, in install_directory\Documentation\Specifications\... for more information.

See also:
- Vector notation on page 5-111
- VFPASSERT VECTOR on page 5-113.

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)
**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**

\[
\text{VFPASSERT VECTOR}[[n[:s]]]
\]

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 Procedure Call Standard for the ARM Architecture specification, aapcs.pdf, in install_directory\Documentation\Specifications\... for more information.

See:
- Vector notation on page 5-111
- VFPASSERT SCALAR on page 5-112.

**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**

- \( \text{VMRS} \ r10,\text{FPSCR} \) ; UAL mnemonic - could be FMRX instead.
- \( \text{BIC} \ r10,r10,#0x00370000 \)
- \( \text{ORR} \ r10,r10,#0x00020000 \) ; set length = 3, stride = 1
- \( \text{VMSR} \ \text{FPSCR},r10 \)
- \( \text{VFPASSERT VECTOR} \) ; assert vector mode, unspecified length & stride
- \( \text{fadd} \ d4, d4, d0 \) ; ERROR, scalar in vector mode
- \( \text{fadds} \ s16<3>, s0, s8<3> \) ; okay
- \( \text{fabss} \ s24<1>, s28<1> \) ; wrong length, but not faulted (unspecified)
VMRS r10,FPSCR
BIC r10,r10,#0x00370000
ORR r10,r10,#0x00030000    ; set length = 4, stride = 1
VMSR FPSCR,r10
VFPASSERT VECTOR<4>        ; assert vector mode, length 4, stride 1
fadds s24<4>, s0, s8<4>  ; okay
fabss s24<2>, s24<2>      ; ERROR, wrong length
VMRS r10,FPSCR
BIC r10,r10,#0x00370000
ORR r10,r10,#0x00130000    ; set length = 4, stride = 2
VMSR FPSCR,r10
VFPASSERT VECTOR<4:2>      ; assert vector mode, length 4, stride 2
fadds s8<4>, 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
Wireless MMX Technology Instructions

This chapter describes support for Wireless MMX™ Technology instructions. It contains the following sections:

- Introduction on page 6-2
- ARM support for Wireless MMX Technology on page 6-3
- Wireless MMX instructions on page 6-7.
6.1 Introduction

Wireless MMX Technology is a set of Single Instruction Multiple Data (SIMD) instructions available on selected XScale processors that improve the performance of some multimedia applications. Wireless MMX Technology uses 64-bit registers to enable it to operate on multiple data elements in a packed format.

Wireless MMX Technology uses ARM coprocessors 0 and 1 to support its instruction set and data types. You can assemble source code that uses Wireless MMX Technology instructions to run on the PXA270 processor.

Wireless MMX 2 Technology is an upgraded version of Wireless MMX Technology.

When using the ARM assembler, be aware that:

- Wireless MMX Technology instructions are only assembled if you specify the supported processor (`armasm --cpu PXA270`).
- The PXA270 processor supports code written in ARM or Thumb® only.
- Most Wireless MMX Technology instructions can be executed conditionally, depending on the state of the ARM flags. The Wireless MMX Technology condition codes are identical to the ARM condition codes.

This chapter gives information on the Wireless MMX Technology support provided by the ARM assembler in RealView Compilation Tools. It does not provide a detailed description of the Wireless MMX Technology. See Wireless MMX Technology Developer Guide for information about the programmers’ model and a full description of the Wireless MMX Technology instruction set.
6.2 ARM support for Wireless MMX Technology

This section gives information on the assembler support for Wireless MMX and MMX 2 Technology. It describes:

- Registers
- Directives, WRN and WCN on page 6-4
- Frame directives on page 6-4
- Wireless MMX load and store instructions on page 6-5
- Wireless MMX Technology and XScale instructions on page 6-6.

6.2.1 Registers

Wireless MMX Technology supports two register types:

Status and Control registers

Control registers map onto coprocessor 1 and include the general-purpose registers wCGR0 - wCGR3 and the SIMD flags. See Table 6-1 for details of these registers.

Use the Wireless MMX Technology instructions TMCR and TMRC to read and write to these registers.

<table>
<thead>
<tr>
<th>Type</th>
<th>Wireless MMX Technology registers</th>
<th>CP1 registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coprocessor ID</td>
<td>wCID</td>
<td>c0</td>
</tr>
<tr>
<td>Control</td>
<td>wCon</td>
<td>c1</td>
</tr>
<tr>
<td>Saturation SIMD flag</td>
<td>wCSSF</td>
<td>c2</td>
</tr>
<tr>
<td>Arithmetic SIMD flag</td>
<td>wCASF</td>
<td>c3</td>
</tr>
<tr>
<td>Reserved</td>
<td>-</td>
<td>c4 - c7</td>
</tr>
<tr>
<td>General-purpose</td>
<td>wCGR0 - wCGR3</td>
<td>c8 - c11</td>
</tr>
<tr>
<td>Reserved</td>
<td>-</td>
<td>c12 - c15</td>
</tr>
</tbody>
</table>

SIMD Data registers

Data registers (wR0 - wR15) map onto coprocessor 0 and hold 16x64-bit packed data. Use the Wireless MMX Technology pseudo-instructions TMRRC and TMCRR to move data between these registers and the ARM registers.
See *Wireless MMX Technology Developer Guide* for a detailed description of registers.

When assembling Wireless MMX Technology instructions, the assembler accepts register specifications in:

- mixed case where this matches exactly the Wireless MMX Technology specification, for example, \( wR0 \), \( wCID \), \( wCon \)
- all lowercase, for example, \( wr0 \), \( wcid \), \( wcon \)
- all uppercase, for example, \( WR0 \), \( WCID \), \( WCON \).

The assembler supports the \( WRN \) and \( WCN \) directives to specify your own register names (see *Directives, WRN and WCN*).

### 6.2.2 Directives, WRN and WCN

Directives are available to support Wireless MMX Technology:

- **WCN** Defines a name for a specified Control register, for example:
  
  \[
  \text{speed WCN wcgr0} \quad \text{defines speed as a symbol for control reg 0}
  \]

- **WRN** Defines a name for a specified SIMD Data register, for example:
  
  \[
  \text{rate WRN wr6} \quad \text{defines rate as a symbol for data reg 6}
  \]

Avoid conflicting uses of the same register under different names. Do not use any of the predefined names listed in *Predefined register and coprocessor names* on page 3-23 or the register names described in *Registers* on page 6-3.

### 6.2.3 Frame directives

Wireless MMX Technology registers can be used with \texttt{FRAME} directives in the usual way to add debug information into your object files (see *Frame directives* on page 7-41 for details). Be aware of the following restrictions:

- A warning is given if you try to push Wireless MMX Technology registers \( wR0 \) - \( wR9 \) or \( wCGR0 \) - \( wCGR3 \) onto the stack (see *FRAME PUSH* on page 7-45).

- Wireless MMX Technology registers cannot be used as address offsets (see *FRAME ADDRESS* on page 7-43 and *FRAME RETURN ADDRESS* on page 7-49).
6.2.4 Wireless MMX load and store instructions

Load and store byte, halfword, word or doublewords to and from Wireless MMX coprocessor registers.

**Syntax**

\[
\text{op}<\text{type}\{\text{cond}\} \text{wRd}, [\text{Rn}, \#\{-\}\text{offset}]\}\}
\]

\[
\text{op}<\text{type}\{\text{cond}\} \text{wRd}, [\text{Rn}] \{, \#\{-\}\text{offset}\}
\]

\[
\text{opW}\{\text{cond}\} \text{wRd}, \text{label}
\]

\[
\text{opW} \text{wCd}, [\text{Rn}, \#\{-\}\text{offset}]\}\}
\]

\[
\text{opW} \text{wCd}, [\text{Rn}] \{, \#\{-\}\text{offset}\}
\]

\[
\text{opD}\{\text{cond}\} \text{wRd}, \text{label}
\]

\[
\text{opD} \text{wCd}, [\text{Rn}, \{-\}\text{Rm} \{, \text{LSL } \text{imm4}\}]\}\}
\]

\[
\text{opD} \text{wCd}, [\text{Rn}], \{-\}\text{Rm} \{, \text{LSL } \text{imm4}\}
\]

where:

- \text{op} can be either:
  - \text{WLDR} Load Wireless MMX Register
  - \text{WSTR} Store Wireless MMX Register.

- \text{<type>} can be any one of:
  - \text{B} Byte
  - \text{H} Halfword
  - \text{W} Word
  - \text{D} Doubleword.

- \text{cond} is an optional condition code (see Conditional execution on page 2-18).

- \text{wRd} is the Wireless MMX SIMD data register to load or save.

- \text{wCd} is the Wireless MMX Status and Control register to load or save.

- \text{Rn} is the register on which the memory address is based.

- \text{offset} is an immediate offset. If offset is omitted, the instruction is a zero offset instruction.

- \text{!} is an optional suffix. If \text{!} is present, the instruction is a pre-indexed instruction.
\textit{label} is a program-relative expression. See Register-relative and program-relative expressions on page 3-37 for more information.  
\textit{label} must be within +/- 1020 bytes of the current instruction.

\textit{Rm} is a register containing a value to be used as the offset. \textit{Rm} must not be r15.

\textit{imm4} contains the number of bits to shift \textit{Rm} left, in the range 0-15.

**Loading constants into SIMD registers**

The assembler also supports the \texttt{WLDRW} and \texttt{WLDRD} literal load pseudo-instructions, for example:

\begin{verbatim}
WLDRW wr0, =0x114
\end{verbatim}

Be aware that:

- The assembler cannot load byte and halfword literals. These produce a downgradable error. If downgraded, the instruction is converted to a \texttt{WLDRW} and a 32-bit literal is generated. This is the same as a byte literal load, but uses a 32-bit word instead.

- If the literal to be loaded is zero, and the destination is a SIMD Data register, the assembler converts the instruction to a \texttt{WZERO}.

- Doubleword loads that are not 8-byte aligned are unpredictable.

6.2.5 *Wireless MMX Technology and XScale instructions*

Wireless MMX Technology instructions overlap with XScale instructions. To avoid conflicts, the assembler has the following restrictions:

- You cannot mix the XScale instructions with Wireless MMX Technology instructions in the same assembly.

- Wireless MMX Technology \texttt{TMIA} instructions have a \texttt{MIA} mnemonic that overlaps with the XScale \texttt{MIA} instructions. Be aware that:
  - \texttt{MIA acc0, Rm, Rs} is accepted in XScale, but faulted in Wireless MMX Technology.
  - \texttt{MIA wr0, Rm, Rs} and \texttt{TMIA wr0, Rm, Rs} are accepted in Wireless MMX Technology.
  - \texttt{TMIA acc0, Rm, Rs} is faulted in XScale (XScale has no \texttt{TMIA} instruction).

For more details on the XScale instructions, see \textit{MIA, MIAPH, and MIAxy} on page 4-91, and \textit{MAR} and \textit{MRA} on page 4-147.
### 6.3 Wireless MMX instructions

Table 6-2 gives a list of the Wireless MMX Technology instruction set. Use it to locate individual instructions described in *Wireless MMX Technology Developer Guide*. See also pseudo-instructions (Table 6-3 on page 6-9).

In this section, Wireless MMX Technology registers are indicated by \( wRn, wRd \), ARM registers are shown as \( Rn, Rd \).

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>TANDC</td>
<td>TANDCB  ( r15 )</td>
</tr>
<tr>
<td>TBCST</td>
<td>TBCSTB  ( wr15, r1 )</td>
</tr>
<tr>
<td>TEXTRC</td>
<td>TEXTRCB ( r15, #0 )</td>
</tr>
<tr>
<td>TEXTRM</td>
<td>TEXTRMUCS ( r3, wr7, #7 )</td>
</tr>
<tr>
<td>TINSR</td>
<td>TINSRB ( wr6, r11, #0 )</td>
</tr>
</tbody>
</table>
| TMIA, TMIAPH, TMIAbb | TMIANE \( wr1, r2, r3 \)  
  TMIAPH \( wr4, r5, r6 \)  
  TMIAbb \( wr4, r5, r6 \)  
  MIAPNE \( wr4, r5, r6 \) |
| TMVMSK   | TMVMSKBNE \( r14, wr15 \) |
| TORC     | TORCB   \( r15 \) |
| WACC     | WACCDBG \( wr1, wr2 \) |
| WADD     | WADDBG \( wr1, wr2, wr13 \) |
| WALIGNI, WALIGNR | WALIGNI \( wr7, wr6, wr5, #3 \)  
  WALIGNR \( wr4, wr8, wr12 \) |
| WAND, WANDN | WAND \( wr1, wr2, wr3 \)  
  WANDN \( wr5, wr5, wr9 \) |
| WAVG2    | WAVG2B \( wr3, wr6, wr9 \)  
  WAVG2BR \( wr4, wr7, wr10 \) |
<p>| WHCPEQ   | WHCPEQB ( wr0, wr4, wr2 ) |
| WCHCPEQ  | WCHCPEQB ( wr0, wr4, wr2 ) |
| WHLDR    | WHLDRB  ( wr1, [r2, #0] ) |
| WMAC     | WMACU   ( wr3, wr4, wr5 ) |</p>
<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>WMADD</td>
<td>WMADDU wr3, wr4, wr5</td>
</tr>
<tr>
<td>WMUL</td>
<td>WMULUL wr4, wr2, wr3</td>
</tr>
<tr>
<td>WMIN</td>
<td>WMINwr0, wr4, wr2</td>
</tr>
<tr>
<td>WMAX</td>
<td>WMAXUB wr0, wr4, wr2</td>
</tr>
<tr>
<td>WMINSB</td>
<td>WMINSB wr0, wr4, wr2</td>
</tr>
<tr>
<td>WSADD</td>
<td>WSADB wr3, wr5, wr8</td>
</tr>
<tr>
<td>WSHUFH</td>
<td>WSHUFH wr8, wr15, #17</td>
</tr>
<tr>
<td>WSLH, WSRH</td>
<td>WSLHwr3, wr1, wr4</td>
</tr>
<tr>
<td>WSRA</td>
<td>WSRAH wr3, wr1, wr4</td>
</tr>
<tr>
<td>WSTR</td>
<td>WSTRB wr1, [r2, #0]</td>
</tr>
<tr>
<td>WSUB</td>
<td>WSUBGE wr1, wr2, wr13</td>
</tr>
<tr>
<td>WUNPKCEH</td>
<td>WUNPKCEHUB wr0, wr4</td>
</tr>
<tr>
<td>WUNPKCEL</td>
<td>WUNPKCELBSB wr0, wr4</td>
</tr>
<tr>
<td>WUNPKCH</td>
<td>WUNPKCHHB wr0, wr4, wr2</td>
</tr>
<tr>
<td>WUNPKCIH</td>
<td>WUNPKCIHHB wr1, wr5, wr3</td>
</tr>
<tr>
<td>WXOR</td>
<td>WXOR wr3, wr1, wr4</td>
</tr>
</tbody>
</table>
6.3.1 Pseudo-instructions

Table 6-3 gives an overview of the Wireless MMX Technology pseudo-instructions. Use it to locate instructions described in the Wireless MMX Technology Developer Guide and in Chapter 4 ARM and Thumb Instructions.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Brief description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>TMCR</td>
<td>Moves the contents of source register, ( Rn ), to Control register, ( wCn ). Maps onto the ARM MCR coprocessor instruction (page 4-125).</td>
<td>TMCR ( wc1, r10 )</td>
</tr>
<tr>
<td>TMCRR</td>
<td>Moves the contents of two source registers, ( RnLo ) and ( RnHi ), to destination register, ( wRd ). Do not use ( r15 ) for either ( RnLo ) or ( RnHi ). Maps onto the ARM MCRR coprocessor instruction (page 4-125).</td>
<td>TMCRR ( wr4, r5, r6 )</td>
</tr>
<tr>
<td>TMRC</td>
<td>Moves the contents of Control register, ( wCn ), to destination register, ( Rd ). Do not use ( r15 ) for ( Rd ). Maps onto the ARM MRC coprocessor instruction (page 4-127).</td>
<td>TMRC ( r1, wc2 )</td>
</tr>
<tr>
<td>TMRRC</td>
<td>Moves the contents of source register, ( wRn ), to two destination registers, ( RdLo ) and ( RdHi ). Do not use ( r15 ) for either destination register. ( RdLo ) and ( RdHi ) must be distinct registers, otherwise the result is unpredictable. Maps onto the ARM MRRC coprocessor instruction (page 4-127).</td>
<td>TMRRC ( r1, r0, wr2 )</td>
</tr>
<tr>
<td>WMOV</td>
<td>Moves the contents of source register, ( wRn ), to destination register, ( wRd ). This instruction is a form of WOR (see Table 6-2 on page 6-7).</td>
<td>WMOV ( wr1, wr8 )</td>
</tr>
<tr>
<td>WZERO</td>
<td>Zeros destination register, ( wRd ). This instruction is a form of WANDN (see Table 6-2 on page 6-7).</td>
<td>WZERO ( wr1 )</td>
</tr>
</tbody>
</table>
Chapter 7
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 7-2
- Symbol definition directives on page 7-3
- Data definition directives on page 7-15
- Assembly control directives on page 7-31
- Frame directives on page 7-41
- Reporting directives on page 7-56
- Instruction set and syntax selection directives on page 7-62
- Miscellaneous directives on page 7-65.

Note
None of these directives is available in the inline assemblers in the ARM C and C++ compilers.
# 7.1 Alphabetical list of directives

Table 7-1 shows a complete list of the directives. Use it to locate individual directives.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ALIAS</td>
<td>page 7-66</td>
<td>EQU</td>
<td>page 7-77</td>
<td>LTORG</td>
<td>page 7-17</td>
</tr>
<tr>
<td>ALIGN</td>
<td>page 7-67</td>
<td>EXPORT or GLOBAL</td>
<td>page 7-78</td>
<td>MACRO and MEND</td>
<td>page 7-32</td>
</tr>
<tr>
<td>ARM and CODE32</td>
<td>page 7-63</td>
<td>EXPORTAS</td>
<td>page 7-80</td>
<td>MAP</td>
<td>page 7-18</td>
</tr>
<tr>
<td>AREA</td>
<td>page 7-70</td>
<td>EXTERN</td>
<td>page 7-82</td>
<td>MEND see MACRO</td>
<td>page 7-32</td>
</tr>
<tr>
<td>ASSERT</td>
<td>page 7-56</td>
<td>FIELD</td>
<td>page 7-19</td>
<td>MEXIT</td>
<td>page 7-36</td>
</tr>
<tr>
<td>ATTR</td>
<td>page 7-74</td>
<td>FRAME ADDRESS</td>
<td>page 7-43</td>
<td>NOPF</td>
<td>page 7-86</td>
</tr>
<tr>
<td>CN</td>
<td>page 7-11</td>
<td>FRAME POP</td>
<td>page 7-44</td>
<td>OPT</td>
<td>page 7-59</td>
</tr>
<tr>
<td>CODE16</td>
<td>page 7-63</td>
<td>FRAME PUSH</td>
<td>page 7-45</td>
<td>PRESERVE8 see REQUIRE8</td>
<td>page 7-87</td>
</tr>
<tr>
<td>COMMON</td>
<td>page 7-29</td>
<td>FRAME REGISTER</td>
<td>page 7-47</td>
<td>PROC see FUNCTION</td>
<td>page 7-54</td>
</tr>
<tr>
<td>CP</td>
<td>page 7-12</td>
<td>FRAME RESTORE</td>
<td>page 7-48</td>
<td>QN</td>
<td>page 7-13</td>
</tr>
<tr>
<td>DATA</td>
<td>page 7-30</td>
<td>FRAME SAVE</td>
<td>page 7-50</td>
<td>RELOC</td>
<td>page 7-8</td>
</tr>
<tr>
<td>DCB and DCQ</td>
<td>page 7-21</td>
<td>FRAME STATE REMEMBER</td>
<td>page 7-51</td>
<td>REQUIRE</td>
<td>page 7-86</td>
</tr>
<tr>
<td>DCD and DCU</td>
<td>page 7-22</td>
<td>FRAME STATE RESTORE</td>
<td>page 7-52</td>
<td>REQUIRE8 and PRESERVE8</td>
<td>page 7-87</td>
</tr>
<tr>
<td>DCDO</td>
<td>page 7-23</td>
<td>FRAME UNMIND ON or OFF</td>
<td>page 7-53</td>
<td>RLIST</td>
<td>page 7-10</td>
</tr>
<tr>
<td>DCFD and DCFDU</td>
<td>page 7-24</td>
<td>FUNCTION or PROC</td>
<td>page 7-54</td>
<td>RN</td>
<td>page 7-9</td>
</tr>
<tr>
<td>DCFS and DCFSU</td>
<td>page 7-25</td>
<td>GBLA, GBLL, and GBLS</td>
<td>page 7-4</td>
<td>ROUT</td>
<td>page 7-88</td>
</tr>
<tr>
<td>DCI</td>
<td>page 7-26</td>
<td>GET or INCLUDE</td>
<td>page 7-81</td>
<td>SETA, SETL, and SETS</td>
<td>page 7-7</td>
</tr>
<tr>
<td>DCQ and DCQU</td>
<td>page 7-27</td>
<td>GLOBAL see EXPORT</td>
<td>page 7-78</td>
<td>SN</td>
<td>page 7-13</td>
</tr>
<tr>
<td>DCW and DCWU</td>
<td>page 7-28</td>
<td>IF, ELSE, ENDIF, and ELIF</td>
<td>page 7-37</td>
<td>SPACE or FILL</td>
<td>page 7-20</td>
</tr>
<tr>
<td>DN</td>
<td>page 7-13</td>
<td>IMPORT</td>
<td>page 7-82</td>
<td>SUBT</td>
<td>page 7-61</td>
</tr>
<tr>
<td>ELIF, ELSE see IF</td>
<td>page 7-37</td>
<td>INCBIN</td>
<td>page 7-84</td>
<td>THUMB</td>
<td>page 7-63</td>
</tr>
<tr>
<td>END</td>
<td>page 7-76</td>
<td>INCLUDE see GET</td>
<td>page 7-81</td>
<td>THUMBX</td>
<td>page 7-63</td>
</tr>
<tr>
<td>ENDFUNC or ENDP</td>
<td>page 7-55</td>
<td>INFO</td>
<td>page 7-57</td>
<td>TTL</td>
<td>page 7-61</td>
</tr>
<tr>
<td>ENDIF see IF</td>
<td>page 7-37</td>
<td>KEEP</td>
<td>page 7-85</td>
<td>WHILE and WEND</td>
<td>page 7-40</td>
</tr>
<tr>
<td>ENTRY</td>
<td>page 7-76</td>
<td>LCLA, LCLL, and LCLS</td>
<td>page 7-6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
7.2 Symbol definition directives

This section describes the following directives:

- **GBLA, GBLL, and GBLS** on page 7-4
  Declare a global arithmetic, logical, or string variable.

- **LCLA, LCLL, and LCLS** on page 7-6
  Declare a local arithmetic, logical, or string variable.

- **SETA, SETL, and SETS** on page 7-7
  Set the value of an arithmetic, logical, or string variable.

- **RELOC** on page 7-8
  Encode an ELF relocation in an object file.

- **RN** on page 7-9
  Define a name for a specified register.

- **RLIST** on page 7-10
  Define a name for a set of general-purpose registers.

- **CN** on page 7-11
  Define a coprocessor register name.

- **CP** on page 7-12
  Define a coprocessor name.

- **QN, DN, and SN** on page 7-13
  Define a double-precision or single-precision VFP register name.
7.2.1 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

```
<gblx> variable
```

where:

- `<gblx>` is one of GBLA, GBLL, or GBLS.
- `variable` is the name of the variable. `variable` must be unique among symbols within a source file.

Usage

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 7-7).

See LCLA, LCLL, and LCLS on page 7-6 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 7-1 declares a variable `objectsize`, sets the value of `objectsize` to 0xFF, and then uses it later in a SPACE directive.

Example 7-1

```plaintext
GBLA objectsize ; declare the variable name
objectsize SETA 0xFF ; set its value
.
. ; other code
.
SPACE objectsize ; quote the variable
```

Example 7-2 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 7-2

```plaintext
armasm --predefine "objectsize SETA 0xFF" sourcefile -o objectfile
```
7.2.2 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

```<lclx> variable```

where:

- `<lclx>` is one of LCLA, LCLL, or LCLS.
- `variable` is the name of the variable. `variable` must be unique within the macro that contains it.

Usage

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 7-32).

Set the value of the variable with a SETA, SETL, or SETS directive (see SETA, SETL, and SETS on page 7-7).

See GBLA, GBLL, and GBLS on page 7-4 for information on declaring global variables.

Example

```MACRO
$label message $a ; Declare a macro
LCLS err ; Macro prototype line
     ; Declare local string
     ; variable err.
err SETS "error no: " ; Set value of err
$label ; code
INFO 0, "err":CC::STR:$a ; Use string
MEND```
7.2.3 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

\[ \text{variable } \text{<setx> expr} \]

where:

- \text{<setx>} is one of SETA, SETL, or SETS.
- \text{variable} is the name of a variable declared by a GBLA, GBLL, GBLS, LCLA, LCLL, or LCLS directive.
- \text{expr} is an expression that is:
  - numeric, for SETA (see Numeric expressions on page 3-34)
  - logical, for SETL (see Logical expressions on page 3-37)
  - string, for SETS (see String expressions on page 3-33).

Usage

You must declare \text{variable} using a global or local declaration directive before using one of these directives. See GBLA, GBLL, and GBLS on page 7-4 and LCLA, LCLL, and LCLS on page 7-6 for more information.

You can also predefine variable names on the command line. See Command syntax on page 3-2 for more information.

Examples

<table>
<thead>
<tr>
<th></th>
<th>GBLA</th>
<th>VersionNumber</th>
</tr>
</thead>
<tbody>
<tr>
<td>VersionNumber</td>
<td>SETA</td>
<td>21</td>
</tr>
<tr>
<td>Debug</td>
<td>SETL</td>
<td>{TRUE}</td>
</tr>
<tr>
<td>VersionString</td>
<td>SETS</td>
<td>&quot;Version 1.0&quot;</td>
</tr>
</tbody>
</table>
7.2.4 RELOC

The RELOC directive explicitly encodes an ELF relocation in an object file.

**Syntax**

RELOC \( n \), symbol

RELOC \( n \)

where:

- \( n \) must be in the range 0 to 255.
- symbol can be any program-relative label.

**Usage**

Use RELOC \( n \), symbol to create a relocation with respect to the address labeled by symbol.

If used immediately after an ARM or Thumb instruction, RELOC results in a relocation at that instruction. If used immediately after a DCB, DCW, or DCD, or any other data generating directive, RELOC results in a relocation at the start of the data. Any addend to be applied must be encoded in the instruction or in the DCI or DCD.

If the assembler has already emitted a relocation at that place, the relocation is updated with the details in the RELOC directive, for example:

```
DCD sym2 ; R_ARM_ABS32 to sym32
RELOC 55 ; ... makes it R_ARM_ABS32_NOI
```

RELOC is faulted in all other cases, for example, after any non-data generating directive, LTORG, ALIGN, or as the first thing in an AREA.

Use RELOC \( n \) to create a relocation with respect to the anonymous symbol, that is, symbol 0 of the symbol table. If you use RELOC \( n \) without a preceding assembler generated relocation, the relocation is with respect to the anonymous symbol. For more information, see the *Application Binary Interface for the ARM Architecture*.

**Examples**

```
IMPORT impsym
LDR r0,[pc,#-8]
RELOC 4, impsym
DCD 0
RELOC 2, sym
DCD 0,1,2,3,4 ; the final word is relocated
RELOC 38,sym2 ; R_ARM_TARGET1
```
7.2.5 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 3-23.

- `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
```
7.2.6 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 3-23.

list-of-registers is a comma-delimited list of register names and register ranges. The register list must be enclosed in braces.

Usage

Use RLIST to give a name to a set of registers to be transferred by the LDM or STM instructions.

LDM and STM always put the lowest physical register numbers at the lowest address in memory, regardless of the order they are supplied to the LDM or STM instruction. If you have defined your own symbolic register names it can be less apparent that a register list is not in increasing register order.

Use the --diag_warning 1206 assembler option to ensure that the registers in a register list are supplied in increasing register order. If registers are not supplied in increasing register order, a warning is issued.

Example

Context RLIST {r0-r6,r8,r10-r12,r15}
7.2.7 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 3-23.
- `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
```
7.2.8 CP

The CP directive defines a name for a specified coprocessor. The coprocessor number must be within the range 0 to 15.

Syntax

\[
\text{name CP expr}
\]

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 3-23.

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

\[
dmu \text{ CP 6 } \quad ; \text{ defines dmu as a symbol for coprocessor 6}
\]
7.2.9 QN, DN, and SN

The QN directive defines a name for a specified 128-bit extension register.
The DN directive defines a name for a specified 64-bit extension register.
The SN directive defines a name for a specified single-precision VFP register.

Syntax

$name$ directive $expr$.type}{[}$x$$]$

where:

directive is QN, DN, or SN.

name is the name to be assigned to the extension register. name cannot be the
same as any of the predefined names listed in Predefined register and
coprocessor names on page 3-23.

expr Can be:

- an expression that evaluates to a number in the range 0-15 for a
double-precision VFPv2 register or a NEON 128-bit register, or
0-31 otherwise.
- a predefined register name, or a register name that has already been
defined in a previous directive.

isse any datatype described in NEON and VFP data types on page 5-14.

[x] is only available for NEON code. [x] is a scalar index into a register.
type and [x] are Extended notation. See Extended notation on page 5-18 for more
information, and Extended notation example on page 7-14 for an example of usage.

Usage

Use QN, DN, or SN to allocate convenient names to extension 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 5-108).
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

Extended notation example

varA DN d1.U16
varB DN d2.U16
varC DN d3.U16
  VADD varA,varB,varC ; VADD.U16 d1,d2,d3
index DN d4.U16[0]
result QN q5.I32
  VMULL result,varA,index ; VMULL.U16 q5,d1,d3[2]
7.3 Data definition directives

This section describes the following directives to allocate memory, define data structures, set initial contents of memory:

- **LTORG** on page 7-17
  Set an origin for a literal pool.

- **MAP** on page 7-18
  Set the origin of a storage map.

- **FIELD** on page 7-19
  Define a field within a storage map.

- **SPACE or FILL** on page 7-20
  Allocate a zeroed block of memory.

- **DCB** on page 7-21
  Allocate bytes of memory, and specify the initial contents.

- **DCD and DCDU** on page 7-22
  Allocate words of memory, and specify the initial contents.

- **DCDO** on page 7-23
  Allocate words of memory, and specify the initial contents as offsets from the static base register.

- **DCFD and DCFDU** on page 7-24
  Allocate doublewords of memory, and specify the initial contents as double-precision floating-point numbers.

- **DCFS and DCFSU** on page 7-25
  Allocate words of memory, and specify the initial contents as single-precision floating-point numbers.

- **DCI** on page 7-26
  Allocate words of memory, and specify the initial contents. Mark the location as code not data.

- **DCQ and DCQU** on page 7-27
  Allocate doublewords of memory, and specify the initial contents as 64-bit integers.
• *DCW and DCWU* on page 7-28
  Allocate halfwords of memory, and specify the initial contents.

• *COMMON* on page 7-29
  Allocate a block of memory at a symbol, and specify the alignment.

• *DATA* on page 7-30
  Mark data within a code section. Obsolete, for backwards compatibility only.
7.3.1 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, VLDR, and WLDR pseudo-instructions. Use LTORG to ensure that a literal pool is assembled within range. For more information on pseudo-instructions, see:

- LDR pseudo-instruction on page 4-159
- VLDR pseudo-instruction on page 5-81
- Wireless MMX load and store instructions on page 6-5.

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
  ; 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.
  ; code
  LDR    r1,=0x55555555  ; => LDR R1, [pc, #offset to Literal Pool 2]
  ; code
  MOV    pc,lr          ; end function
  LTORG                   ; Literal Pool 2 contains literal &55555555.

data    SPACE   4200            ; Clears 4200 bytes of memory,
  ; starting at current location.
  END                     ; Default literal pool is empty.
```
7.3.2  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 7-19 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
7.3.3 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:

- \textit{label} is an optional label. If specified, \textit{label} is assigned the value of the storage location counter, \{VAR\}. The storage location counter is then incremented by the value of \textit{expr}.

- \textit{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 7-18).

Example

The following example shows how register-relative labels are defined using the MAP and FIELD directives.

\begin{verbatim}
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,\([r9,#4]\)
\end{verbatim}
7.3.4 SPACE or FILL

The SPACE directive reserves a zeroed block of memory. % is a synonym for SPACE.

The FILL directive reserves a block of memory to fill with the given value.

Syntax

\{label\} SPACE \ expr
\{label\} FILL \ expr{,value{,valuesize}}

where:

- label is an optional label.
- expr evaluates to the number of bytes to fill or zero (see Numeric expressions on page 3-34).
- value evaluates to the value to fill the reserved bytes with. value is optional and if omitted, it is 0. value must be 0 in a NOINIT area.
- valuesize is the size, in bytes, of value. It can be any of 1, 2, or 4. valuesize is optional and if omitted, it is 1.

Usage

Use the ALIGN directive to align any code following a SPACE or FILL directive. See ALIGN on page 7-67 for more information.

See also:
- DCB on page 7-21
- DCD and DCDU on page 7-22
- DCDO on page 7-23
- DCW and DCWU on page 7-28.

Example

\begin{verbatim}
AREA    MyData, DATA, READWRITE
\hline
data1   SPACE 255 ; defines 255 bytes of zeroed store
data2   FILL 50,0xAB,1 ; defines 50 bytes containing 0xAB
\end{verbatim}
7.3.5  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 3-34).
- 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 7-67 for more information.

See also:

- DCD and DCDU on page 7-22
- DCQ and DCQU on page 7-27
- DCW and DCWU on page 7-28
- SPACE or FILL on page 7-20.

Example

Unlike C strings, ARM assembler strings are not nul-terminated. You can construct a nul-terminated C string using DCB as follows:

C_string  DCB "C_string",0
7.3.6 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 3-34).
- 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 7-21
- DCI on page 7-26
- DCW and DCWU on page 7-28
- DCQ and DCQU on page 7-27
- SPACE or FILL on page 7-20.

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
```
7.3.7 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**

```c
{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**

```c
IMPORT externsym
DCDO externsym ; 32-bit word relocated by offset of
                ; externsym from base of SB section.
```
7.3.8 DCFD and DCFDU

The **DCF** 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.

**DCFDU** is the same, except that the memory alignment is arbitrary.

**Syntax**

```
{label} DCFD{U} {fpliteral",fpliteral}...
```

where:

**fpliteral** is a double-precision floating-point literal (see *Floating-point literals* on page 3-36).

**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 **DCF** 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 7-25.

**Examples**

```
DCFD  1E308,-4E-100
DCFDU 10000,-.1,3.1E26
```
7.3.9 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.

DCFSU is the same, except that the memory alignment is arbitrary.

Syntax

\{label\} \texttt{DCFS\{U\} fpliteral\{,fpliteral\}...}

where:

\textit{fpliteral} is a single-precision floating-point literal (see Floating-point literals on page 3-36).

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 $3.40282347 \times 10^{38}$
- minimum $1.17549435 \times 10^{-38}$.

See also DCFD and DCFDU on page 7-24.

Example

\begin{verbatim}
DCFS 1E3,-4E-9
DCFSU 1.0,-1,3.1E6
\end{verbatim}
7.3.10 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\{.W\} expr\{,expr\}

where:

expr is a numeric expression (see Numeric expressions on page 3-34).

.W if present, indicates that four bytes must be inserted in Thumb code.

Usage

The DCI directive is very like the DCD or DCW directives, but the location is marked as code instead of data. Use DCI when writing macros for new instructions not supported by the version of the assembler you are using.

In ARM code, DCI 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 7-22 and DCW and DCWU on page 7-28.

Example macro

MACRO
    newinstr $Rd,$Rm
    ; this macro translates newinstr Rd,Rm
    ; to the appropriate machine code

    DCI 0xe16f0f10 :OR: ($Rd:SHL:12) :OR: $Rm

MEND

Thumb-2 example

DCI.W 0xf3af8000 ; inserts 32-bit NOP, 2-byte aligned.
7.3.11 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

\{\textit{label}\} DCQ\{U\} {-}\{\textit{literal}\},{-}\{\textit{literal}\}...

where:

\textit{literal} is a 64-bit numeric literal (see \textit{Numeric literals} on page 3-35).

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 \textit{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:

- \textit{DCB} on page 7-21
- \textit{DCD and DCDU} on page 7-22
- \textit{DCW and DCWU} on page 7-28
- \textit{SPACE or FILL} on page 7-20.

Example

\begin{verbatim}
AREA MiscData, DATA, READWRITE

data  DCQ -225,2_101 ; 2_101 means binary 101.
      DCQU number+4 ; number must already be defined.
\end{verbatim}
7.3.12 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{U} expr{,expr}...

where:

expr is a numeric expression that evaluates to an integer in the range –32768 to 65535 (see Numeric expressions on page 3-34).

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 7-21
- DCD and DCDU on page 7-22
- DCQ and DCQU on page 7-27
- SPACE or FILL on page 7-20.

Example

data    DCW     -225,2*number   ; number must already be defined
DCWU    number+4
7.3.13 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**

```plaintext
COMMON symbol{,size{,alignment}} {{[attr]}}
```

where:

- `symbol` is the symbol name. The symbol name is case-sensitive.
- `size` is the number of bytes to reserve.
- `alignment` is the alignment.
- `attr` can be any one of:
  - `DYNAMIC` sets the ELF symbol visibility to STV_DEFAULT.
  - `PROTECTED` sets the ELF symbol visibility to STV_PROTECTED.
  - `HIDDEN` sets the ELF symbol visibility to STV_HIDDEN.
  - `INTERNAL` sets the ELF symbol visibility to STV_INTERNAL.

**Usage**

The linker allocates the required space as zero initialized memory during the link stage. You cannot define, IMPORT or EXTERN a symbol that has already been created by the COMMON directive. In the same way, if a symbol has already been defined or used with the IMPORT or EXTERN directive, you cannot use the same symbol for the COMMON directive.

**Example**

```plaintext
LDR r0, =xyz
COMMON xyz,255,4   ; defines 255 bytes of ZI store, word-aligned
```

**Incorrect examples**

```plaintext
COMMON foo,4,4
COMMON bar,4,4
foo DCD 0           ; cannot define label with same name as COMMON
IMPORT bar          ; cannot import label with same name as COMMON
```
7.3.14 DATA

The DATA directive is no longer required. It is ignored by the assembler.
7.4 Assembly control directives

This section describes the following directives to control conditional assembly, looping, inclusions, and macros:

- **MACRO and MEND** on page 7-32
- **MEXIT** on page 7-36
- **IF, ELSE, ENDIF, and ELIF** on page 7-37
- **WHILE and WEND** on page 7-40.

7.4.1 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.
7.4.2 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 2-46 for more information.

Syntax

Two directives are used to define a macro. The syntax is:

MACRO

{$label} macroname{$cond} {$parameter{,}$parameter}...

; code

MEND

where:

$label is a parameter that is substituted with a symbol given when the macro is invoked. The symbol is usually a label.

macroname is the name of the macro. It must not begin with an instruction or directive name.

$cond is a special parameter designed to contain a condition code. Values other than valid condition codes are permitted.

$parameter is a parameter that is substituted when the macro is invoked. A default value for a parameter can be set using this format:

$parameter="default value"

Double quotes must be used if there are any spaces within, or at either end of, the default value.

Usage

If you start any WHILE...WEND loops or IF...ENDIF conditions within a macro, they must be closed before the MEND directive is reached. See MEXIT on page 7-36 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 or $cond can be used in the same way as other variables (see Assembly time substitution of variables on page 3-28). They are given new values each time the macro is invoked. Parameters must begin with $ to distinguish them from ordinary symbols. Any number of parameters can be used.

$label is optional. It is useful if the macro defines internal labels. It is treated as a parameter to the macro. It does not necessarily represent the first instruction in the macro expansion. The macro defines the locations of any labels.
Use | as the argument to use the default value of a parameter. An empty string is used if the argument is omitted.

In a macro that uses several internal labels, it is useful to define each internal label as the base label with a different suffix.

Use a dot between a parameter and following text, or a following parameter, if a space is not required in the expansion. Do not use a dot between preceding text and a parameter.

You can use the $cond parameter for condition codes. Use the unary operator :REVERSE_CC: to find the inverse condition code, and :CC_ENCODING: to find the 4-bit encoding of the condition code.

Macros define the scope of local variables (see LCLA, LCLL, and LCLS on page 7-6).

Macros can be nested (see Nesting directives on page 7-31).

Examples

; macro definition
MACRO
$label   xmac    $p1,$p2
    ; code
$label.loop1 ; code
    ; code
    BGE     $label.loop1
$label.loop2 ; code
    BL      $p1
    BGT     $label.loop2
    ; code
    ADR     $p2
    ; code
MEND

; macro invocation
abc   xmac    subr1,de
    ; code
    ; this is what is
    ; produced when
    ; the xmac macro is
    BGE     abcloop1
    ; code
abcloop1 ; code
    ; this is what is
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

--- Note ---
When variables are also being passed in as arguments, use of | might leave some
variables unsubstituted. To workaround this, define the | in a LCLS or GBLS variable and
pass this variable as an argument instead of l. For example:

MACRO                        ; Macro definition
m2 $a,$b=r1,$c             ; The default value for $b is r1
add $a,$b,$c               ; The macro adds $b and $c and puts result in $a
MEND                         ; Macro end

MACRO                        ; Macro definition
m1 $a,$b                   ; This macro adds $b to r1 and puts result in $a
LCLS def                   ; Declare a local string variable for |
def SETS "|"                ; Define |
m2 $a,$def,$b              ; Invoke macro m2 with $def instead of |
                          ; to use the default value for the second argument.
MEND                         ; Macro end

--- Conditional macro example ---

AREA  codx, CODE, READONLY

; macro definition

MACRO
Return$cond
[ {ARCHITECTURE} <> "4"
BX$cond lr
|
MOV$cond pc,lr
]
MEND

; macro invocation

fun PROC
CMP    r0,#0
MOVEQ  r0,#1
ReturnEQ
MOV r0, #0
Return
ENDP
END
7.4.3 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 7-32.

Example

MACRO
$abc example abc $param1,$param2
 ; code
WHILE condition1
 ; code
IF condition2
 ; code
MEXIT
ELSE
 ; code
ENDIF
WEND
 ; code
MEND
7.4.4 IF, ELSE, ENDIF, and ELIF

The IF directive introduces a condition that is used to decide whether to assemble a sequence of instructions and directives. { is a synonym for IF.

The ELSE directive marks the beginning of a sequence of instructions or directives that you want to be assembled if the preceding condition fails. | is a synonym for ELSE.

The ENDIF directive marks the end of a sequence of instructions or directives that you want to be conditionally assembled. } is a synonym for ENDIF.

The ELIF directive creates a structure equivalent to ELSE IF, without the requirement for nesting or repeating the condition. See Using ELIF on page 7-38 for details.

**Syntax**

```
IF logical-expression
   ;;code
{ELSE
   ;;code}
ENDIF
```

where:

- `logical-expression`
  - is an expression that evaluates to either {TRUE} or {FALSE}.

See Relational operators on page 3-42.

**Usage**

Use IF with ENDIF, and optionally with ELSE, for sequences of instructions or directives that are only to be assembled or acted on under a specified condition.

IF...ENDIF conditions can be nested (see Nesting directives on page 7-31).
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
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 7-3 assembles the first set of instructions if NEWVERSION is defined, or the alternative set otherwise.

Example 7-3 Assembly conditional on a variable being defined

```
IF :DEF:NEWVERSION
   ; first set of instructions or directives
ELSE
   ; alternative set of instructions or directives
ENDIF
```

Invoking armasm as follows defines NEWVERSION, so the first set of instructions and directives are assembled:

```
armasm --predefine "NEWVERSION SETL {TRUE}" test.s
```

Invoking armasm as follows leaves NEWVERSION undefined, so the second set of instructions and directives are assembled:

```
armasm test.s
```

Example 7-4 assembles the first set of instructions if NEWVERSION has the value {TRUE}, or the alternative set otherwise.

Example 7-4 Assembly conditional on a variable value

```
IF NEWVERSION = {TRUE}
   ; first set of instructions or directives
ELSE
   ; alternative set of instructions or directives
ENDIF
```

Invoking armasm as follows causes the first set of instructions and directives to be assembled:

```
armasm --predefine "NEWVERSION SETL {TRUE}" test.s
```

Invoking armasm as follows causes the second set of instructions and directives to be assembled:

```
armasm --predefine "NEWVERSION SETL {FALSE}" test.s
```
7.4.5 **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**

```plaintext
WHILE logical-expression
code
WEND
```

where:

- `logical-expression` is an expression that can evaluate to either `{TRUE}` or `{FALSE}` (see *Logical expressions* on page 3-37).

**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 7-31).

**Example**

```plaintext
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
```
7.5 Frame directives

This section describes the following directives:

- FRAME ADDRESS on page 7-43
- FRAME POP on page 7-44
- FRAME PUSH on page 7-45
- FRAME REGISTER on page 7-47
- FRAME RESTORE on page 7-48
- FRAME RETURN ADDRESS on page 7-49
- FRAME SAVE on page 7-50
- FRAME STATE REMEMBER on page 7-51
- FRAME STATE RESTORE on page 7-52
- FRAME UNWIND ON on page 7-53
- FRAME UNWIND OFF on page 7-53
- FUNCTION or PROC on page 7-54
- ENDFUNC or ENDP on page 7-55.

Correct use of these directives:

- enables the arm|link --callgraph option to calculate stack usage of assembler functions.
- 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.

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.
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.
7.5.1 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:

reg  is the register on which the canonical frame address is to be based. This is sp unless the function uses a separate frame pointer.

offset  is the offset of the canonical frame address from reg. If offset is zero, you can omit it.

Usage

Use FRAME ADDRESS if your code alters which register the canonical frame address is based on, or if it changes the offset of the canonical frame address from the register. You must use FRAME ADDRESS immediately after the instruction that changes the calculation of the canonical frame address.

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 7-45).

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 7-44).

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      ; New base register
FRAME ADDRESS fp,4     ; code using fp to base call-frame on, instead of sp
7.5.2 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 three alternative syntaxes for FRAME POP:

\[
\text{FRAME POP \{reglist\}}
\]
\[
\text{FRAME POP \{reglist\},n}
\]
\[
\text{FRAME POP n}
\]

where:

\[\text{reglist}\] 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.

If \(n\) is not specified or is zero, the assembler calculates the new offset for the canonical frame address from \{reglist\}. It assumes that:

- each ARM register popped occupies four bytes on the stack
- each VFP single-precision register popped occupies four bytes on the stack, plus an extra four-byte word for each list
- each VFP double-precision register popped occupies eight bytes on the stack, plus an extra four-byte word for each list.

See FRAME ADDRESS on page 7-43 and FRAME RESTORE on page 7-48.
7.5.3 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 {reglist}, n
- FRAME PUSH n

where:

- reglist is a list of registers stored consecutively below the canonical frame address. There must be at least one register in the list.
- n is the number of bytes that the stack pointer moves.

Usage

FRAME PUSH is equivalent to a FRAME ADDRESS and a FRAME SAVE directive. You can use it when a single instruction saves registers and alters the stack pointer.

You must use FRAME PUSH immediately after the instruction it refers to.

If n is not specified or is zero, the assembler calculates the new offset for the canonical frame address from {reglist}. It assumes that:

- each ARM register pushed occupies four bytes on the stack
- each VFP single-precision register pushed occupies four bytes on the stack, plus an extra four-byte word for each list
- each VFP double-precision register popped occupies eight bytes on the stack, plus an extra four-byte word for each list.

See FRAME ADDRESS on page 7-43 and FRAME SAVE on page 7-50.
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
7.5.4 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.
7.5.5 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 7-44).
7.5.6 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.
7.5.7 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 7-45).
7.5.8 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 7-52
- FUNCTION or PROC on page 7-54.

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
7.5.9 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 7-51
- FUNCTION or PROC on page 7-54.
7.5.10 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.

Note

A FRAME UNWIND directive is not sufficient to turn on exception table generation. Furthermore a FRAME UNWIND directive, without other FRAME directives, is not sufficient information for the assembler to generate the unwind information.

See also Controlling exception table generation on page 3-20.

7.5.11 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 3-20.
7.5.12 FUNCTION or PROC

The FUNCTION directive marks the start of a function. PROC is a synonym for FUNCTION.

Syntax

\[ \text{label} \text{ FUNCTION} \{\{\text{reglist1}\}, \{\text{reglist2}\}\} \]

where:

- \text{reglist1} is an optional list of callee saved ARM registers. If \text{reglist1} is not present, and your debugger checks register usage, it will assume that the AAPCS is in use.

- \text{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 r13 (sp), and the frame state stack to be empty.

Each FUNCTION directive must have a matching ENDFUNC directive. You must not nest FUNCTION and ENDFUNC pairs, and they must not contain PROC or ENDP directives.

You can use the optional \text{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 7-43 to FRAME STATE RESTORE on page 7-52.

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 7-67 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

7.5.13 ENDFUNC or ENDP

The ENDFUNC directive marks the end of an AAPCS-conforming function (see FUNCTION or PROC on page 7-54). ENDP is a synonym for ENDFUNC.
### 7.6 Reporting directives

This section describes the following directives:

- **ASSERT** generates an error message if an assertion is false during assembly.
- **INFO** on page 7-57 generates diagnostic information during assembly.
- **OPT** on page 7-59 sets listing options.
- **TTL and SUBT on page 7-61** insert titles and subtitles in listings.

#### 7.6.1 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 7-57.

**Example**

```
ASSERT label1 <= label2    ; Tests if the address
; represented by label1
; is <= the address
; represented by label2.
```
7.6.2  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[, severity]

where:

numeric-expression

is a numeric expression that is evaluated during assembly. If the
expression evaluates to zero:

- no action is taken during pass one
- string-expression is printed as a warning during pass two if
  severity is 1
- string-expression is printed as a message during pass two if
  severity is 0 or not specified.

If the expression does not evaluate to zero:

- string-expression is printed as an error message and the assembly
  fails irrespective of whether severity is specified or not (non-zero
  values for severity are reserved in this case).

string-expression

is an expression that evaluates to a string.

severity

is an optional number that controls the severity of the message. Its value
can be either 0 or 1. All other values are reserved.

Usage

INFO provides a flexible means of creating custom error messages. See Numeric
expressions on page 3-34 and String expressions on page 3-33 for additional
information on numeric and string expressions.

See also ASSERT on page 7-56.
Examples

INFO 0, "Version 1.0"
IF endofdata <= label1
    INFO 4, "Data overrun at label1"
ENDIF
7.6.3 OPT

The OPT directive sets listing options from within the source code.

**Syntax**

```
OPT n
```

where:

- **n** is the OPT directive setting. Table 7-2 lists valid settings.

<table>
<thead>
<tr>
<th>OPT n</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Turns on normal listing.</td>
</tr>
<tr>
<td>2</td>
<td>Turns off normal listing.</td>
</tr>
<tr>
<td>4</td>
<td>Page throw. Issues an immediate form feed and starts a new page.</td>
</tr>
<tr>
<td>8</td>
<td>Resets the line number counter to zero.</td>
</tr>
<tr>
<td>16</td>
<td>Turns on listing for SET, GBL and LCL directives.</td>
</tr>
<tr>
<td>32</td>
<td>Turns off listing for SET, GBL and LCL directives.</td>
</tr>
<tr>
<td>64</td>
<td>Turns on listing of macro expansions.</td>
</tr>
<tr>
<td>128</td>
<td>Turns off listing of macro expansions.</td>
</tr>
<tr>
<td>256</td>
<td>Turns on listing of macro invocations.</td>
</tr>
<tr>
<td>512</td>
<td>Turns off listing of macro invocations.</td>
</tr>
<tr>
<td>1024</td>
<td>Turns on the first pass listing.</td>
</tr>
<tr>
<td>2048</td>
<td>Turns off the first pass listing.</td>
</tr>
<tr>
<td>4096</td>
<td>Turns on listing of conditional directives.</td>
</tr>
<tr>
<td>8192</td>
<td>Turns off listing of conditional directives.</td>
</tr>
<tr>
<td>16384</td>
<td>Turns on listing of MEND directives.</td>
</tr>
<tr>
<td>32768</td>
<td>Turns off listing of MEND directives.</td>
</tr>
</tbody>
</table>
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 Listing output to a file on page 3-17 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
```
7.6.4 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.         |
7.7 Instruction set and syntax selection directives

This section describes the following directives:

- ARM, THUMB, THUMBX, CODE16 and CODE32 on page 7-63.
7.7.1 ARM, THUMB, THUMBX, CODE16 and CODE32

The ARM directive and the CODE32 directive are synonyms. They instruct the assembler to interpret subsequent instructions as ARM instructions, using either the UAL or the pre-Thumb-2 ARM assembler language syntax.

The THUMB directive instructs the assembler to interpret subsequent instructions as Thumb instructions, using the UAL syntax.

The THUMBX directive instructs the assembler to interpret subsequent instructions as Thumb-2EE instructions, using the UAL syntax.

The CODE16 directive instructs the assembler to interpret subsequent instructions as Thumb instructions, using the pre-UAL assembly language syntax.

If necessary, these directives also insert up to three bytes of padding to align to the next word boundary for ARM, or up to one byte of padding to align to the next halfword boundary for Thumb or Thumb-2EE.

Syntax

ARM
THUMB
THUMBX
CODE16
CODE32

Usage

In files that contain code using different instruction sets:

- ARM must precede any ARM code. CODE32 is a synonym for ARM.
- THUMB must precede Thumb code written in UAL syntax.
- THUMBX must precede Thumb-2EE code written in UAL syntax.
- CODE16 must precede Thumb code written in pre-UAL syntax.

These directives do not assemble to any instructions. They also do not change the state. They only instruct the assembler to assemble ARM, Thumb, or Thumb-2EE 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 ToThumb, CODE, READONLY ; Name this block of code
ENTRY                ; Mark first instruction to execute
ARM                  ; Subsequent instructions are ARM

start
  ADR     r0, into_thumb + 1      ; Processor starts in ARM state
  BX      r0                      ; Inline switch to Thumb state
  THUMB                           ; Subsequent instructions are Thumb

into_thumb
  MOVS    r0, #10                 ; New-style Thumb instructions
```
7.8 Miscellaneous directives

This section describes the following directives:

- ALIAS on page 7-66
- ALIGN on page 7-67
- AREA on page 7-70
- ATTR on page 7-74
- END on page 7-76
- ENTRY on page 7-76
- EQU on page 7-77
- EXPORT or GLOBAL on page 7-78
- EXPORTAS on page 7-80
- GET or INCLUDE on page 7-81
- IMPORT and EXTERN on page 7-82
- INCBIN on page 7-84
- KEEP on page 7-85
- NOFP on page 7-86
- REQUIRE on page 7-86
- REQUIRE8 and PRESERVE8 on page 7-87
- ROUT on page 7-88.
7.8.1 ALIAS

The ALIAS directive creates an alias for a symbol.

Syntax

ALIAS name, aliasname

where:

name is the name of the symbol to create an alias for
aliasname is the name of the alias to be created.

Usage

The symbol name must already be defined in the source file before creating an alias for it. Properties of name set by the EXPORT directive (see EXPORT or GLOBAL on page 7-78) will not be inherited by aliasname, so you must use EXPORT on aliasname if you want to make the alias available outside the current source file. Apart from the properties set by the EXPORT directive, name and aliasname are identical. See Data definition directives on page 7-15.

Example

baz
bar PROC
BX lr
ENDP
ALIAS bar,foo ; foo is an alias for bar
EXPORT bar
EXPORT foo ; foo and bar have identical properties
; because foo was created using ALIAS
EXPORT baz ; baz and bar are not identical
; because the size field of baz is not set

Incorrect example

EXPORT bar
IMPORT car
ALIAS bar,foo ; ERROR – bar is not defined yet
ALIAS car,boo ; ERROR – car is external
bar PROC
BX lr
ENDP
7.8.2 ALIGN

The ALIGN directive aligns the current location to a specified boundary by padding with zeros or NOP instructions.

Syntax

ALIGN {expr{,offset{,pad{,padsize}}}{}

where:

- **expr** is a numeric expression evaluating to any power of 2 from $2^0$ to $2^{31}$
- **offset** can be any numeric expression
- **pad** can be any numeric expression
- **padsize** can be 1, 2 or 4.

Operation

The current location is aligned to the next address of the form:

$$offset + n \times expr$$

If **expr** is not specified, ALIGN sets the current location to the next word (four byte) boundary. The unused space between the previous and the new current location are filled with:

- copies of **pad**, if **pad** is specified
- NOP instructions, if all the following conditions are satisfied:
  - **pad** is not specified
  - the ALIGN directive follows ARM or Thumb instructions
  - the current section has the CODEALIGN attribute set on the AREA directive
- zeros otherwise.

**pad** is treated as a byte, halfword, or word, according to the value of **padsize**. If **padsize** is not specified, **pad** defaults to bytes in data sections, halfwords in Thumb code, or words in ARM code.
Usage

Use ALIGN to ensure that your data and code is aligned to appropriate boundaries. This is typically required in the following circumstances:

- The ADR Thumb pseudo-instruction can only load addresses that are word aligned, but a label within Thumb code might not be word aligned. Use ALIGN 4 to ensure four-byte alignment of an address within Thumb code.
- Use ALIGN to take advantage of caches on some ARM processors. For example, the ARM940T has a cache with 16-byte lines. Use ALIGN 16 to align function entries on 16-byte boundaries and maximize the efficiency of the cache.
- 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 7-15) 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 7-70 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 subroutine1
    ; ensures that subroutine1 addresses
    ; the following instruction.
    MOV r5,#0x5
7.8.3 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 2-15 for more information.

Syntax

```
AREA sectionname{,attr}{,attr}...
```

where:

- `sectionname` is the name to give to the section.
  
  You can choose any name for your sections. However, names starting with a non-alphabetic character must be enclosed in bars or a missing section name error is generated. For example, `|1_DataArea|`.
  
  Certain names are conventional. For example, `|.text|` is used for code sections produced by the C compiler, or for code sections otherwise associated with the C library.

- `attr` are one or more comma-delimited section attributes. Valid attributes are:
  
  **ALIGN=expression**
  
  By default, ELF sections are aligned on a four-byte boundary. `expression` can have any integer value from 0 to 31. The section is aligned on a $2^{expression}$-byte boundary. For example, if `expression` is 10, the section is aligned on a 1KB boundary.
  
  *This is not the same as the way that the ALIGN directive is specified.* See ALIGN on page 7-67.

  ________ Note ________
  
  Do not use ALIGN=0 or ALIGN=1 for ARM code sections.
  Do not use ALIGN=0 for Thumb code sections.

  **ASSOC=section**
  
  `section` specifies an associated ELF section.
  `sectionname` must be included in any link that includes `section`

- `CODE` Contains machine instructions. READONLY is the default.
CODEALIGN
Causes the assembler to insert NOP instructions when the ALIGN directive is used after ARM or Thumb instructions within the section, unless the ALIGN directive specifies a different padding.

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 Chapter 3 Using the Basic Linker Functionality in the Linker User Guide.

COMGROUP=symbol_name
Is a common group section. All sections within a common group are common. When the object is linked, other object files may have a GROUP with signature symbol_name. Only one group is kept in the final image.

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.

FINI_ARRAY
Sets the ELF type of the current area to SHT_FINI_ARRAY.

GROUP=symbol_name
Is the signature for the group and must be defined by the source file, or a file included by the source-file. All AREAS with the same symbol_name signature are placed in the same group. Sections within a group are kept or discovered together.

INIT_ARRAY
Sets the ELF type of the current area to SHT_INIT_ARRAY.
LINKORDER=section
Specifies a relative location for the current section in the image. It ensures that the order of all the sections with the LINKORDER attribute, with respect to each other, is the same as the order of the corresponding named sections in the image.

MERGE=n
Indicates that the linker can merge the current section with other sections with the MERGE=n attribute. \( n \) is the size of the elements in the section, for example \( n \) is 1 for characters. You must not assume that the section will be merged because the attribute does not force the linker to merge the sections.

NOALLOC
Indicates that no memory on the target system is allocated to this area.

NOINIT
Indicates that the data section is uninitialized, or initialized to zero. It contains only space reservation directives SPACE or DCB, DCD, DCDU, DCQ, DCQU, DCW, or DCWU with initialized values of zero. You can decide at link time whether an area is uninitialized or zero initialized. See Chapter 3 Using the Basic Linker Functionality in the Linker User Guide.

PREINIT_ARRAY
Sets the ELF type of the current area to SHT_PREINIT_ARRAY.

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.

SECFLAGS=n
Adding one or more ELF flags, denoted by \( n \), to the current section.

SECTYPE=n
Sets the ELF type of the current section to \( n \).

STRINGS
Add the SHF_STRINGS flag to the current section. To use the STRINGS attribute, you must also use the MERGE=1 attribute. The contents of the section must be strings that are null-terminated using the DCB directive.
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 3-31 and ROUT on page 7-88).

There must be at least one AREA directive for an assembly.

Note

The assembler emits R_ARM_TARGET1 relocations for the DCD and DCDU directives (see DCD and DCDU on page 7-22) if the directive uses PC-relative expressions and is in any of the PREINIT_ARRAY, FINI_ARRAY, or INIT_ARRAY ELF sections. You can override the relocation using the RELOC directive (see RELOC on page 7-8) after each DCD or DCDU directive. If this relocation is used, read-write sections might become read-only sections at link time if the platform ABI permits this.

Example

The following example defines a read-only code section named Example.

```
AREA Example, CODE, READONLY ; An example code section.
    ; code
```
7.8.4 ATTR

The ATTR set directives set values for the ABI build attributes.
The ATTR scope directives specify the scope for which the set value applies to.

Syntax

ATTR FILESCOPE

ATTR SCOPE name

ATTR settype, tagid, value

where:

name is a section name or symbol name.

settype can be any of:

• SETVALUE
• SETSTRING
• SETCOMPATIBLEWITHVALUE
• SETCOMPATIBLEWITHSTRING

tagid is an attribute tag name (or its numerical value) defined in the ABI for the ARM Architecture.

value depends on settype:

• is a 32-bit integer value when settype is SETVALUE or SETCOMPATIBLEWITHVALUE
• is a null-terminated string when settype is SETSTRING or SETCOMPATIBLEWITHSTRING

Usage

The ATTR set directives following the ATTR FILESCOPE directive apply to the entire object file. The ATTR set directives following the ATTR SCOPE name directive apply only to the named section or symbol.

For tags that expect an integer, you must use SETVALUE or SETCOMPATIBLEWITHVALUE. For tags that expect a string, you must use SETSTRING or SETCOMPATIBLEWITHSTRING. For the list of tag names, see Addenda to, and Errata in, the ABI for the ARM Architecture.

Use SETCOMPATIBLEWITHVALUE and SETCOMPATIBLEWITHSTRING to set tag values which the object file is also compatible with.
Examples

ATTR SETSTRING Tag_CPU_raw_name, "Cortex-A8"
ATTR SETVALUE Tag_VFP_arch, 3 ; VFPv3 instructions were permitted.
ATTR SETVALUE 10, 3 ; 10 is the numerical value of Tag_VFP_arch.
7.8.5 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 7-81 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.

7.8.6 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
7.8.7 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 7-85 and EXPORT or GLOBAL on page 7-78 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
7.8.8 EXPORT or GLOBAL

The EXPORT directive declares a symbol that can be used by the linker to resolve symbol references in separate object and library files. GLOBAL is a synonym for EXPORT.

Syntax

EXPORT {{WEAK}}
EXPORT symbol {{type}}
EXPORT symbol [attr{,type}]
EXPORT symbol [WEAK{,attr}{,type}]

where:

symbol is the symbol name to export. The symbol name is case-sensitive. If symbol is omitted, all symbols are exported.

WEAK symbol is only imported into other sources if no other source exports an alternative symbol. If [WEAK] is used without symbol, all exported symbols are weak.

attr can be any one of:

DYNAMIC sets the ELF symbol visibility to STV_DEFAULT.
PROTECTED sets the ELF symbol visibility to STV_PROTECTED.
HIDDEN sets the ELF symbol visibility to STV_HIDDEN.
INTERNAL sets the ELF symbol visibility to STV_INTERNAL.

type specifies the symbol type:

DATA symbol is treated as data when the source is assembled and linked.
CODE symbol is treated as code when the source is assembled and linked.

ELFTYPE=n symbol is treated as a particular ELF symbol, as specified by the value of n, where n can be any number from 0 to 15.

If unspecified, the assembler determines the most appropriate type.
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.

See also IMPORT and EXTERN on page 7-82.

See the ELF for the ARM Architecture ABI documentation on http://infocenter.arm.com for more information on symbol visibility.

Example

    AREA Example, CODE, READONLY
    EXPORT DoAdd ; Export the function name
                  ; to be used by external
                  ; modules.
    DoAdd   ADD     r0,r0,r1

Symbol visibility can be overridden for duplicate exports. In the following example, the last EXPORT takes precedence for both binding and visibility:

    EXPORT SymA[WEAK] ; Export as weak-hidden
    EXPORT SymA[DYNAMIC] ; SymA becomes non-weak dynamic.
7.8.9 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 7-78.

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 ; the symbol 'two' will appear in the object
EXPORT one ; file's symbol table with the value 2.
7.8.10 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 7-31).

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 7-84).

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
```
7.8.11 IMPORT and EXTERN

These directives provide the assembler with a name that is not defined in the current assembly.

Syntax

directive symbol {{type}}
directive symbol [attr{,type}]
directive symbol [WEAK{,attr}{,type}]

where:

directive can be either:
   IMPORT imports the symbol unconditionally.
   EXTERN imports the symbol only if it is referred to in the current assembly.

symbol is a symbol name defined in a separately assembled source file, object file, or library. The symbol name is case-sensitive.

WEAK prevents the linker generating an error message if the symbol is not defined elsewhere. It also prevents the linker searching libraries that are not already included.

attr can be any one of:
   DYNAMIC sets the ELF symbol visibility to STV_DEFAULT.
   PROTECTED sets the ELF symbol visibility to STV_PROTECTED.
   HIDDEN sets the ELF symbol visibility to STV_HIDDEN.
   INTERNAL sets the ELF symbol visibility to STV_INTERNAL.

type specifies the symbol type:
   DATA symbol is treated as data when the source is assembled and linked.
   CODE symbol is treated as code when the source is assembled and linked.
   ELFTYPE=n symbol is treated as a particular ELF symbol, as specified by the value of n, where n can be any number from 0 to 15.

If unspecified, the linker determines the most appropriate type.
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.

See the ELF for the ARM Architecture ABI documentation on http://infocenter.arm.com for more information on symbol visibility.

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.
```
7.8.12 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
```
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 7-18).

**Example**

```
label   ADC     r2,r3,r4
KEEP    label       ; makes label available to debuggers
ADD     r2,r2,r5
```
7.8.14 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.

7.8.15 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.
7.8.16 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 checks that any code that requires eight-byte alignment of the stack is only called, directly or indirectly, by code that preserves eight-byte alignment of the stack.

**Syntax**

REQUIRE8 {bool}
PRESERVE8 {bool}

where:

**bool** is an optional Boolean constant, either {TRUE} or {FALSE}.

**Usage**

Where required, if your code preserves eight-byte alignment of the stack, use PRESERVE8 to set the PRES8 build attribute on your file. If your code does not preserve eight-byte alignment of the stack, use PRESERVE8 {FALSE} to ensure that the PRES8 build attribute is not set.

**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

7.8.17 ROUT

The ROUT directive marks the boundaries of the scope of local labels (see Local labels on page 3-31).

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 7-70).

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

; code
routineA ROUT ; ROUT is not necessarily a routine
; code
3routineA ; code ; this label is checked
; code
BEQ %4routineA ; this reference is checked
; code
BGE %3 ; refers to 3 above, but not checked
; code
4routineA ; code ; this label is checked
; code
otherstuff ROUT ; start of next scope