RealView® Compilation Tools

Version 3.1

Assembler Guide

ARM®

Copyright © 2002-2007 ARM Limited. All rights reserved.
ARM DUI 0204H

RealView Compilation Tools
Assembler Guide

Copyright © 2002-2007 ARM Limited. All rights reserved.

Release Information

The following changes have been made to this book.

Change History

Date	Issue	Confidentiality	Change
August 2002	A	Non-Confidential	Release 1.2
January 2003	B	Non-Confidential	Release 2.0
September 2003	C	Non-Confidential	Release 2.0.1 for RVDS v2.0
January 2004	D	Non-Confidential	Release 2.1 for RVDS v2.1
December 2004	E	Non-Confidential	Release 2.2 for RVDS v2.2
May 2005	F	Non-Confidential	Release 2.2 for RVDS v2.2 SP1
March 2006	G	Non-Confidential	Release 3.0 for RVDS v3.0
March 2007	H	Non-Confidential	Release 3.1 for RVDS v3.1.

Proprietary Notice

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

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

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

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

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

Confidentiality Status

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

Product Status

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

Web Address

`http://www.arm.com`

Copyright © 2002-2007 ARM Limited. All rights reserved.

Copyright © 2002-2007 ARM Limited. All rights reserved.

Contents
RealView Compilation Tools Assembler Guide

Copyright © 2002-2007 ARM Limited. All rights reserved.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

Chapter 7 Directives Reference

Copyright © 2002-2007 ARM Limited. All rights reserved.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

Preface

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

Copyright © 2002-2007 ARM Limited. All rights reserved.

About this book

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

Intended audience

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

Using this book

This book is organized into the following chapters:

Chapter 1 ***Introduction***

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

Chapter 2 ***Writing ARM Assembly Language***

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

Chapter 3 ***Assembler Reference***

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

Chapter 4 ***ARM and Thumb Instructions***

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

 Copyright © 2002-2007 ARM Limited. All rights reserved.

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.

`mono`space 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 Limited and third parties that provide additional information on developing code for the ARM family of processors.

Copyright © 2002-2007 ARM Limited. All rights reserved.

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

ARM publications

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

- *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 and Utilities Guide* (ARM DUI 0206)
- *RVCT Developer Guide* (ARM DUI 0203)
- *NEON Vectorizing Compiler Guide* (ARMDUI 0350)
- *RealView Development Suite Glossary* (ARM DUI 0324).

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:

- *ARM6-M Architecture Reference Manual* (ARM DDI 0419)
- *ARM7-M Architecture Reference Manual* (ARM DDI 0403)
- *ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition* (ARM DDI 0406)
- *ARM Architecture Reference Manual Thumb®-2 Supplement* (ARM DDI 0308)
- *ARM Architecture Reference Manual Security Extensions Supplement* (ARM DDI 0309)
- *ARM Architecture Reference Manual Thumb-2 Execution Environment Supplement* (ARM DDI 0397)
- *ARM Architecture Reference Manual Advanced SIMD Extension and VFPv3 Supplement* (ARM DDI 0268)
- *ARM Reference Peripheral Specification* (ARM DDI 0062)
- ARM datasheet or technical reference manual for your hardware device.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

Other publications

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

For full information about the Intel® Wireless MMX™ Technology, see *Wireless MMX Technology Developer Guide* (August, 2000), Order Number: 251793-001, available from `http://www.intel.com` .

Copyright © 2002-2007 ARM Limited. All rights reserved.

Feedback

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

Feedback on RealView Compilation Tools

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

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

Feedback on this book

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

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

General suggestions for additions and improvements are also welcome.

Copyright © 2002-2007 ARM Limited. All rights reserved.

Chapter 1
Introduction

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

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

Copyright © 2002-2007 ARM Limited. All rights reserved.

1.1 About the RealView Compilation Tools assemblers

RVCT has:

- 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 *RealView Compilation Tools Developer Guide* for more information on the inline and embedded assemblers.

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

1.1.1 ARM assembly language

The current ARM/Thumb assembler language has superseded earlier versions of both the ARM and Thumb assembler languages.

Code written using the current language can be assembled for ARM, Thumb, or Thumb-2. 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 contains support for Wireless MMX Technology Control and *Single Instruction Multiple Data* (SIMD) Data registers, and includes 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 in RVCT.

1.1.3 NEON technology

ARM NEON™ Technology is an optional component of ARMv7 architecture. 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 supports integer, fixed-point, and single-precision floating-point SIMD operations. These instructions are available in both ARM and Thumb-2.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

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.

Copyright © 2002-2007 ARM Limited. All rights reserved.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

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:

Copyright © 2002-2007 ARM Limited. All rights reserved.

2.1 Introduction

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

This chapter does not provide a detailed description of the ARM, Thumb®-2, Thumb, NEON™, VFP, or 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 RVCT2.1 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 2-51.

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* (RVISS).

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

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

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

 Copyright © 2002-2007 ARM Limited. All rights reserved.

2.2 Overview of the ARM architecture

This section gives a brief overview of the ARM architecture.

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

This section describes:

- *Architecture versions*
- *ARM, Thumb, 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 2-9.

2.2.1 Architecture versions

The information and examples in this book assume that you are using a processor that implements ARMv4 or above. 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.

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

Copyright © 2002-2007 ARM Limited. All rights reserved.

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 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.

ThumbEE introduces a new instruction set state, ThumbEE state. In this state, instructions are executed as defined in the ThumbEE instruction set.

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 7-60 for details.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

2.2.4 Processor mode

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

Note

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

Table 2-1 ARM processor modes

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

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

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

Modes other than User mode are entered to service exceptions, or to access privileged resources (see Chapter 6 *Handling Processor Exceptions* in the *RealView Compilation Tools Developer Guide*).

Copyright © 2002-2007 ARM Limited. All rights reserved.

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, as r0, r1, ... , r13, r14.

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

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

In the exception handling modes, r14 holds the return address for the exception, or a subroutine return address if subroutine calls are executed within an exception. r14 can be used as a general-purpose register if the return address is stored on the stack.

The *Program Counter* (PC)

The Program Counter is accessed as r15 (or pc). It is incremented by one word (four bytes) for each instruction in ARM state, or by 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, r15 (pc) does not contain the address of the currently executing instruction. The address of the currently executing instruction is typically pc–8 for ARM, or pc–4 for Thumb.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

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 2-17 for more information.

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

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 4-131 and *MSR* on page 4-133 for details.

The *Current Program Status Register* (CPSR)

The CPSR holds:

- the APSR flags
- the current processor mode
- interrupt disable flags.

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

On ARMv6T2 and above, Thumb-2 introduces new state bits to the CPSR. These are used by the `IT` instruction to control conditional execution of the IT block (see *IT* on page 4-68).

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 4-131 and *MSR* on page 4-133 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 RealView Compilation Tools Developer Guide for more information.

Copyright © 2002-2007 ARM Limited. All rights reserved.

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 to determine whether the code being branched to is Thumb code or ARM code.

Before the introduction of Thumb-2, the Thumb instruction set was limited to a restricted subset of the functionality of the ARM instruction set. Almost all Thumb instructions were 16-bit. The Thumb-2 instruction set functionality is almost identical to that of the ARM instruction set.

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

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

- *Branch instructions*
- *Data processing instructions*
- *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 instructions

These instructions are used to:

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

Data processing instructions

These instructions operate on the general-purpose registers. They can perform operations such as addition, subtraction, or bitwise logic on the contents of two registers and place the result in a third register. They can also operate on the value in a single register, or on a value in a register and a constant supplied within the instruction (an *immediate value*).

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

 Copyright © 2002-2007 ARM Limited. All rights reserved.

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* on page 2-10
- *Access to the inline barrel shifter* on page 2-10.

Conditional execution

Almost all ARM instructions can be executed conditionally on the value of the ALU status flags in the 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 processors without Thumb-2, the only mechanism for conditional execution is a conditional branch. Most data processing instructions update the ALU flags. You cannot generally specify whether instructions update the state of the ALU flags or not.

Copyright © 2002-2007 ARM Limited. All rights reserved.

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 2-17 for a detailed description.

Register access

In ARM state, all instructions can access r0 to r14, and most can also access r15 (pc). The MRS and MSR instructions can move the contents of a status register to a general-purpose register, where they can be manipulated by normal data processing operations. See *MRS* on page 4-131 and *MSR* on page 4-133 for more information.

Thumb state on Thumb-2 processors provides the same facilities, except that some of the less useful accesses to r13 and r15 are not permitted.

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

Access to the inline barrel shifter

The ARM arithmetic logic unit has a 32-bit barrel shifter that is capable of shift and rotate operations. The second operand to all ARM and Thumb-2 data-processing and single register data-transfer instructions can be shifted, before the data-processing or data-transfer is executed, as part of the instruction. This supports, but is not limited to:

- scaled addressing
- multiplication by a constant
- constructing constants.

See *Loading constants into registers* on page 2-25 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 16-bit Thumb instruction set only allows access to the barrel shifter using separate instructions.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

2.3 Structure of assembly language modules

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

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

armasm can also accept source code written in old Thumb assembly language. In this case, you must inform armasm using the --16 command-line option, or the CODE16 directive in the source code. The old 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 2-14
- *Calling subroutines* on page 2-16.

2.3.1 Layout of assembly language source files

The general form of source lines in assembly language is:

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

Note

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

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

Case rules

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

Copyright © 2002-2007 ARM Limited. All rights reserved.

Line length

To make source files easier to read, a long line of source can be split onto several lines by placing a backslash character (\) at the end of the line. The backslash must not be followed by any other characters (including spaces and tabs). The assembler treats the backslash/end-of-line sequence as white space.

Note

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

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

Labels

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

The assembler calculates the address of a label relative to the origin of the section where the label is defined. A reference to a label within the same section can use the PC plus or minus an offset. This is called *program-relative addressing*.

Addresses of labels in other sections are calculated at link time, when the linker has allocated specific locations in memory for each section.

Local labels

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

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

See *Local labels* on page 3-27 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.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

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.

Copyright © 2002-2007 ARM Limited. All rights reserved.

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 3 *Using the Basic Linker Functionality* in the *RealView Compilation Tools Linker and Utilities Guide* for more information on how the linker places sections.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

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 7-66 for a detailed description of the syntax of the AREA directive.

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

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

The ENTRY directive

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

Application execution

The application code in Example 2-1 on page 2-14 begins executing at the label start, where it loads the decimal values 10 and 3 into registers r0 and r1. These registers are added together and the result placed in r0.

Application termination

After executing the main code, the application terminates by returning control to the debugger. This is done using the ARM semihosting SVC (0x123456 by default), with the following parameters:

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

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.

Copyright © 2002-2007 ARM Limited. All rights reserved.

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
```

 Copyright © 2002-2007 ARM Limited. All rights reserved.

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 instruction only update a subset. If a flag is not updated, the original value is preserved. The descriptions 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 earlier architectures, most data processing instructions update the ALU status flags automatically. There is no option not to update the flags. Other instructions cannot update the flags.

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 a processor with Thumb-2, you can 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.

This section describes:

- *The ALU status flags* on page 2-18
- *Conditional execution* on page 2-18
- *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.

Copyright © 2002-2007 ARM Limited. All rights reserved.

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 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 on page 2-19 shows the condition codes that you can use.

On Thumb processors without Thumb-2, the {*cond*} field is only permitted on certain branch instructions.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

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

Table 2-2 Condition code suffixes

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

Example 2-3 shows an example of conditional execution.

Example 2-3

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


Copyright © 2002-2007 ARM Limited. All rights reserved.

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.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

2.4.4 Example of the use of conditional execution

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

In C the algorithm can be expressed as:

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

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

```
gcd     CMP     r0, r1
        BEQ     end
        BLT     less
        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:

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


Copyright © 2002-2007 ARM Limited. All rights reserved.

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

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

Table 2-3 Conditional branches only

r0: a	r1: b	Instruction	Cycles (ARM7)
1	2	`CMP r0, r1`	1
1	2	`BEQ end`	1 (not executed)
1	2	`BLT less`	3
1	2	`SUB r1, r1, r0`	1
1	2	`B gcd`	3
1	1	`CMP r0, r1`	1
1	1	`BEQ end`	3
			Total = 13

Table 2-4 All instructions conditional

r0: a	r1: b	Instruction	Cycles (ARM7)
1	2	`CMP r0, r1`	1
1	2	`SUBGT r0,r0,r1`	1 (not executed)
1	1	`SUBLT r1,r1,r0`	1
1	1	`BNE gcd`	3
1	1	`CMP r0,r1`	1
1	1	`SUBGT r0,r0,r1`	1 (not executed)
1	1	`SUBLT r1,r1,r0`	1 (not executed)
1	1	`BNE gcd`	1 (not executed)
			Total = 10

Copyright © 2002-2007 ARM Limited. All rights reserved.

16-bit Thumb version of gcd

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

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

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

Thumb-2 version of gcd

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

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

This assembles equally well to ARM or Thumb-2 code. The assembler checks the IT instructions, but omits them on assembly to ARM code. (You can omit the 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 *ARM Architecture Reference Manual* and the technical reference manuals for individual processors for full information.

Copyright © 2002-2007 ARM Limited. All rights reserved.

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 *SMULxy and SMLAxy* on page 4-77 and *SMULWy and SMLAWy* on page 4-79).

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 4-133).

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 4-131).

 Copyright © 2002-2007 ARM Limited. All rights reserved.

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. The range of constants that you can generate using 16-bit Thumb instructions is much smaller.

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

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

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

The following sections describe:

- how to use the `MOV` and `MVN` instructions to load a range of immediate values.
  See *Direct loading with MOV and MVN* on page 2-26.
- 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.

Copyright © 2002-2007 ARM Limited. All rights reserved.

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 2-27 shows the range of 8-bit values that this provides (for data processing operations).

Table 2-6 on page 2-27 shows the range of 16-bit values that this provides (for `MOV` instructions only).

 Copyright © 2002-2007 ARM Limited. All rights reserved.

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

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

Table 2-6 ARM state immediate constants in MOV instructions

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

Notes

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

a The MVN values are not available directly as operands in other, non data processing, instructions.

b These values are available in ARM state only. All the other values in this table are also available in Thumb-2, except where noted.

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

Copyright © 2002-2007 ARM Limited. All rights reserved.

Thumb-2 immediate constants

In Thumb state, in ARMv6T2 and above:

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

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

Table 2-7 on page 2-29 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).

 Copyright © 2002-2007 ARM Limited. All rights reserved.

Table 2-7 Thumb state immediate constants

Binary	Decimal	Step	Hexadecimal	MVN value[a]	Notes
00000000000000000000000abcdefgh	0-255	1	0-0xFF	–1 to –256	-
00000000000000000000000abcdefgh0	0-510	2	0-0x1FE	–2 to –512	-
00000000000000000000000abcdefgh00	0-1020	4	0-0x3FC	–4 to –1024	-
	...	...	...	...	-
0abcdefgh000000000000000000000000	0-255 x 2^{23}	2^{23}	0-0x7F800000	1-256 x -2^{23}	-
abcdefgh000000000000000000000000	0-255 x 2^{24}	2^{24}	0-0xFF000000	1-256 x -2^{24}	-
abcdefghabcdefghabcdefghabcdefgh	(bit pattern)	-	0xXYXYXYXY	0xXYXYXYXY	-
00000000abcdefgh00000000abcdefgh	(bit pattern)	-	0x00XY00XY	0xFFXYFFXY	-
abcdefgh00000000abcdefgh00000000	(bit pattern)	-	0xXY00XY00	0xXYFFXYFF	-
00000000000000000000abcdefghijkl	0-4095	1	0-0xFFF	-	See b in *Notes*

Table 2-8 Thumb state immediate constants in MOV instructions

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

Notes

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

a The MVN values are not available directly as operands in other instructions.

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

c These values are only available in MOV instructions.

Copyright © 2002-2007 ARM Limited. All rights reserved.

2.5.2 Loading with MOV32

In ARMv6T2, both ARM and Thumb-2 instruction sets include:

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

You can use these two instructions to construct any 32-bit constant in a register. Alternatively, you can use the `MOV32` pseudo-instruction. The assembler generates the `MOV`, `MOVT` instruction pair for you. See *MOV32 pseudo-instruction* on page 4-151 for a description of the syntax of the `MOV32` pseudo-instruction.

2.5.3 Loading with LDR Rd, =const

The `LDR Rd,=const` pseudo-instruction can construct any 32-bit numeric constant in a single instruction. You can use this pseudo-instruction to generate constants that are out of range of the `MOV` and `MVN` instructions.

The `LDR` pseudo-instruction generates the most efficient single instruction for a specific constant:

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

  For example:

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

  You must ensure that there is a literal pool within range of the `LDR` instruction generated by the assembler. See *Placing literal pools* on page 2-31 for more information.

See *LDR pseudo-instruction* on page 4-153 for a description of the syntax of the `LDR` pseudo-instruction.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

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 Thumb code when using the 16-bit instruction.

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 (Thumb, 16-bit Thumb-2). See *LTORG* on page 7-18 for a detailed description.

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

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

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

Example 2-4

```
        AREA     Loadcon, CODE, READONLY
        ENTRY                              ; Mark first instruction to execute
start
        BL       func1                     ; Branch to first subroutine
        BL       func2                     ; Branch to second subroutine
stop
        MOV      r0, #0x18                 ; angel_SWIreason_ReportException
```


Copyright © 2002-2007 ARM Limited. All rights reserved.

```
        LDR     r1, =0x20026            ; ADP_Stopped_ApplicationExit
        SVC     #0x123456               ; ARM semihosting (formerly SWI)

func1
        LDR     r0, =42                 ; => MOV R0, #42
        LDR     r1, =0x55555555         ; => LDR R1, [PC, #offset to
                                        ; Literal Pool 1]
        LDR     r2, =0xFFFFFFFF         ; => MVN R2, #0
        BX      lr
        LTORG                           ; Literal Pool 1 contains
                                        ; literal 0x55555555
func2
        LDR     r3, =0x55555555         ; => LDR R3, [PC, #offset to
                                        ; Literal Pool 1]
        ; LDR r4, =0x66666666           ; If this is uncommented it
                                        ; fails, because Literal Pool 2
                                        ; is out of reach
        BX      lr
LargeTable
        SPACE   4200                    ; Starting at the current location,
                                        ; clears a 4200 byte area of memory
                                        ; to zero
        END                             ; Literal Pool 2 is empty
```

2.5.4 Loading floating-point constants

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 5-37 for details of the NEON instructions
- *VMOV* on page 5-96 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 5-77 for details.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

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.

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 processors without Thumb-2.

ADR

The available range depends on the instruction set:

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

See *ADR* on page 4-22 for details.

Copyright © 2002-2007 ARM Limited. All rights reserved.

ADRL

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

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

The available range depends on the instruction set in use:

ARM ±64KB to a byte or halfword-aligned address.
±256KB to a word-aligned address.

32-bit Thumb-2 ±1MB to a byte, halfword, or word-aligned address.

16-bit Thumb ADRL is not available.

See *Loading addresses with LDR Rd, =label* on page 2-36 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 *Code examples* on page 2-2 for instructions on how to assemble, link, and execute the example.

Example 2-5 Implementing a jump table (ARM)

```
        AREA    Jump, CODE, READONLY     ; Name this block of code
        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
stop
        MOV     r0, #0x18                ; angel_SWIreason_ReportException
        LDR     r1, =0x20026             ; ADP_Stopped_ApplicationExit
        SVC     #0x123456                ; ARM semihosting (formerly SWI)
arithfunc                                ; Label the function
        CMP     r0, #num                 ; Treat function code as unsigned
integer
        BXHS    lr                       ; If code is >= num then simply return
        ADR     r3, JumpTable            ; Load address of jump table
        LDR     pc, [r3,r0,LSL#2]        ; Jump to the appropriate routine
```

 Copyright © 2002-2007 ARM Limited. All rights reserved.

```
JumpTable
        DCD     DoAdd
        DCD     DoSub

DoAdd
        ADD     r0, r1, r2              ; Operation 0
        BX      lr                      ; Return
DoSub
        SUB     r0, r1, r2              ; Operation 1
        BX      lr                      ; Return
        END                             ; Mark the end of this file
```

In Example 2-5 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:

argument1=0 Result = argument2 + argument3.

argument1=1 Result = argument2 – argument3.

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

`EQU` Is an assembler directive. It is used to give a value to a symbol. In Example 2-5 on page 2-34 it assigns the value 2 to *num*. When *num* is used elsewhere in the code, the value 2 is substituted. Using `EQU` in this way is similar to using `#define` to define a constant in C.

`DCD` Declares one or more words of store. In 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 `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.

Copyright © 2002-2007 ARM Limited. All rights reserved.

2.6.2 Loading addresses with LDR Rd, =label

The LDR Rd,= pseudo-instruction can load any 32-bit constant into a register (see *Loading with LDR Rd, =const* on page 2-30). It also accepts program-relative expressions such as labels, and labels with offsets.

The assembler converts an LDR r0, =*label* pseudo-instruction by:

- placing the address of *label* in a literal pool (a portion of memory embedded in the code to hold constant values)
- generating a program-relative LDR instruction that reads the address from the literal pool, for example:

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

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

```
        AREA    LDRlabel, CODE,READONLY
        ENTRY                           ; Mark first instruction to execute
start
        BL      func1                   ; Branch to first subroutine
        BL      func2                   ; Branch to second subroutine
stop
        MOV     r0, #0x18               ; angel_SWIreason_ReportException
        LDR     r1, =0x20026            ; ADP_Stopped_ApplicationExit
        SVC     #0x123456               ; ARM semihosting (formerly SWI)
func1
        LDR     r0, =start              ; => LDR R0,[PC, #offset into
                                        ; Literal Pool 1]
```

 Copyright © 2002-2007 ARM Limited. All rights reserved.

```
        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
```


Copyright © 2002-2007 ARM Limited. All rights reserved.

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 7-22 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
```

 Copyright © 2002-2007 ARM Limited. All rights reserved.

2.7 Load and store multiple register instructions

The ARM, Thumb, and Thumb-2 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 2-43.

Copyright © 2002-2007 ARM Limited. All rights reserved.

2.7.1 Load and store multiple instructions available in ARM and Thumb

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

`LDM` Load Multiple registers.

`STM` Store Multiple registers.

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

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

In `LDM` and `STM` instructions:

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

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

In `PUSH` and `POP` instructions:

- The stack pointer (r13) is the base register, and is always updated.
- The address is incremented after each transfer in `POP` instructions, and decremented before each transfer in `PUSH` instructions.
- The list of registers loaded or stored can include:
  - in ARM instructions, any or all of r0-r15
  - in 32-bit Thumb-2 instructions, any or all of r0-r12, and optionally r14 or r15 with some restrictions
  - in 16-bit Thumb-2 and Thumb instructions, any or all of r0-r7, and optionally r14 (`PUSH` only) or r15 (`POP` only).

 Copyright © 2002-2007 ARM Limited. All rights reserved.

2.7.2 Implementing stacks with LDM and STM

The load and store multiple instructions can update the base register. For stack operations, the base register is usually the stack pointer, r13. This means that you can use these instructions to implement push and pop operations for any number of registers in a single instruction.

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

Descending or ascending

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

Full or empty

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

To make it easier for the programmer, stack-oriented suffixes can be used instead of the increment or decrement, and before or after suffixes. See Table 2-9 for a list of stack-oriented suffixes.

Table 2-9 Suffixes for load and store multiple instructions

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

For example:

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

Note

The *Procedure Call Standard for the ARM Architecture* (AAPCS), and ARM and Thumb C and C++ compilers always use a full descending stack.

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

Copyright © 2002-2007 ARM Limited. All rights reserved.

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 4 *Interworking ARM and Thumb* in the RealView Compilation Tools Developer Guide for more information on mixing ARM and Thumb.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

2.7.3 Block copy with LDM and STM

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

Example 2-8 Block copy without LDM and STM

```
            AREA    Word, CODE, READONLY     ; name this block of code
num         EQU     20                       ; set number of words to be copied
            ENTRY                            ; mark the first instruction called
start
            LDR     r0, =src                 ; r0 = pointer to source block
            LDR     r1, =dst                 ; r1 = pointer to destination block
            MOV     r2, #num                 ; r2 = number of words to copy
wordcopy
            LDR     r3, [r0], #4             ; load a word from the source and
            STR     r3, [r1], #4             ; store it to the destination
            SUBS    r2, r2, #1               ; decrement the counter
            BNE     wordcopy                 ; ... copy more
stop
            MOV     r0, #0x18                ; angel_SWIreason_ReportException
            LDR     r1, =0x20026             ; ADP_Stopped_ApplicationExit
            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 2-44 lists the block copy module rewritten to use `LDM` and `STM` for copying.

Copyright © 2002-2007 ARM Limited. All rights reserved.

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 (r13)
blockcopy
            MOVS    r3,r2, LSR #3            ; Number of eight word multiples
            BEQ     copywords                ; Less than eight words to move?
            PUSH    {r4-r11}                 ; Save some working registers
octcopy
            LDM     r0!, {r4-r11}            ; Load 8 words from the source
            STM     r1!, {r4-r11}            ; and put them at the destination
            SUBS    r3, r3, #1               ; Decrement the counter
            BNE     octcopy                  ; ... copy more
            POP     {r4-r11}                 ; Don't need these now - restore
                                             ; 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
```


Copyright © 2002-2007 ARM Limited. All rights reserved.

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:

 Copyright © 2002-2007 ARM Limited. All rights reserved.

2.8.1 Test-and-branch macro example

In ARM code, and Thumb on a processor without Thumb-2, a test-and-branch operation requires two ARM instructions to implement.

You can define a macro definition such as this:

```
        MACRO
$label  TestAndBranch  $dest, $reg, $cc

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

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

This macro can be invoked as follows:

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

After substitution this becomes:

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

2.8.2 Unsigned integer division macro example

Example 2-10 on page 2-47 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.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

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

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

To avoid multiple definitions of labels if `DivMod` is used more than once in the assembler source, the macro uses local labels (90, 91). See *Local labels* on page 2-12 for more information.

Example 2-11 on page 2-48 shows the code that this macro produces if it is invoked as follows:

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


Copyright © 2002-2007 ARM Limited. All rights reserved.

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
```


Copyright © 2002-2007 ARM Limited. All rights reserved.

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 *RealView Compilation Tools Linker and Utilities Guide* for a full description of symbol versioning in RVCT.

Copyright © 2002-2007 ARM Limited. All rights reserved.

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 7-40 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.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

2.11 Assembly language changes

Table 2-10 shows the main differences between the current ARM/Thumb assembler language and the earlier separate ARM and Thumb assembler languages. The old ARM syntax is accepted by the assembler.

Table 2-10 Changes from earlier ARM assembly language

Change	Old ARM syntax	Preferred syntax
The default addressing mode for LDM and STM is IA	`LDMIA, STMIA`	`LDM, STM`
You can use the PUSH and POP mnemonics for full, descending stack operations in ARM as well as Thumb.	`STMFD sp!, {reglist}` `LDMFD sp!, {reglist}`	`PUSH {reglist}` `POP {reglist}`

You can use the LSL, LSR, ASR, ROR, and RRX instruction mnemonics for instructions with rotations and no other operation, in ARM as well as Thumb.	`MOV Rd, Rn, LSL shift`
	`MOV Rd, Rn, LSR shift`
	`MOV Rd, Rn, ASR shift`
	`MOV Rd, Rn, ROR shift`
	`MOV Rd, Rn, RRX`
		`LSL Rd, Rn, shift`
		`LSR Rd, Rn, shift`
		`ASR Rd, Rn, shift`
		`ROR Rd, Rn, shift`
		`RRX Rd, Rn`

Use the *label* form for PC-relative addressing. Do not use the *offset* form in new code.	`LDR Rd, [pc, #offset]`	`LDR Rd, label`
Specify both registers for doubleword memory accesses. You must still obey rules about the register combinations you can use.	`LDRD Rd, addr_mode`	`LDRD Rd, Rd2, addr_mode`
{*cond*}, if used, is always the last element of all instructions.	`ADD{cond}S` `LDR{cond}SB`	`ADDS{cond}` `LDRSB{cond}`

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.	`ADDEQ r1, r2, r3`
	`LDRNE r1, [r2, r3]`
		`ITEQ E`
		`ADDEQ r1, r2, r3`
		`LDRNE r1, [r2, r3]`

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

Table 2-11 Relaxation of requirements

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

You can write source code for Thumb processors using the ARM/Thumb assembler language.

Copyright © 2002-2007 ARM Limited. All rights reserved.

If you are writing Thumb code for a pre-Thumb-2 processor, you must restrict yourself to instructions that are available on the processor. The assembler generates error messages if you attempt to use an instruction that is not available.

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

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

Table 2-12 Differences between old Thumb syntax and current syntax

Change	Old Thumb syntax	New syntax
The default addressing mode for `LDM` and `STM` is `IA`	`LDMIA, STMIA`	`LDM, STM`

You must use the `S` postfix on instructions that update the flags. This change is essential to avoid conflict with 32-bit Thumb-2 instructions.	`ADD r1, r2, r3`
	`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`
`NOP` instructions replace `MOV r8, r8` when available.	`- NOP`	`NOP MOV r8, r8`

 Copyright © 2002-2007 ARM Limited. All rights reserved.

Chapter 3
Assembler Reference

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

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.

Copyright © 2002-2007 ARM Limited. All rights reserved.

3.1 Command syntax

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

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

Invoke the ARM assembler using this command:

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

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

`--16` Instructs the assembler to interpret instructions as Thumb® instructions, using the old Thumb syntax. This is equivalent to a `CODE16` directive at the head of the source file. Use the `--thumb` option to specify Thumb or Thumb-2 instructions using the new syntax.

`--32` Instructs the assembler to interpret instructions as ARM instructions. This is the default.

`--apcs [qualifiers]`

Specifies whether you are using the *Procedure Call Standard for the ARM Architecture* (AAPCS). It can also specify some attributes of code sections. See *AAPCS* on page 3-8 for details.

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

`--bigend` Instructs the assembler to assemble code suitable for a big-endian ARM. The default is `--littleend`.

`--brief_diagnostics`

See *Controlling the output of diagnostic messages* on page 3-14.

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

`--checkreglist`

Instructs the assembler to check `RLIST`, `LDM`, and `STM` register lists to ensure that all registers are provided in increasing register number order. A warning is given if registers are not listed in order.

This option is deprecated and will be removed in a future release. Use `--diag_warning 1206` instead (see *Controlling the output of diagnostic messages* on page 3-14).

`--cpu name` Sets the target CPU. See *CPU names* on page 3-10.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

`--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.

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

See *Controlling the output of diagnostic messages* on page 3-14.

`--dllexport_all`

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

`--dwarf2` Use with `--debug`, to instruct the assembler to generate 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 3-16.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

`--exceptions_unwind`

See *Controlling exception table generation* on page 3-16.

`--fpmode` *model*

Specifies the floating-point conformance, and sets library attributes and floating-point optimizations. See *Floating-point model* on page 3-9.

`--fpu` *name* Selects the target *floating-point unit* (FPU) architecture. See *FPU names* on page 3-10.

`-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 7-74).

`--keep` Instructs the assembler to keep local labels in the symbol table of the object file, for use by the debugger (see *KEEP* on page 7-78).

`--length` See *Listing output to a file* on page 3-13

`--library_type=`*lib*

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

Where *lib* can be one of:

`standardlib` Specifies that the full RVCT runtime libraries are selected at link time. This is the default.

`microlib` Specifies that the C micro-library (microlib) is selected at link time.

Note

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

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 2-54 in the *RealView Compilation Tools Compiler Reference Guide*.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

`--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 3-13 for details.

`-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 3-11.

Note

The `--memaccess` option is deprecated and will be removed in a future release.

`--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 3-16.

`--no_exceptions_unwind`

See *Controlling exception table generation* on page 3-16.

`--no_hide_all`

Enables you to control symbol visibility when building SVr4 shared objects. All exported definitions and references are given dynamic visibility (see *EXPORT or GLOBAL* on page 7-71).

`--no_regs` Instructs the assembler not to predefine register names. See *Predefined register and coprocessor names* on page 3-19 for a list of predefined register names.

This option is deprecated and will be removed in a future release. Use `--regnames=none` instead.

`--no_terse` See *Listing output to a file* on page 3-13

Copyright © 2002-2007 ARM Limited. All rights reserved.

`--no_unaligned_access`

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

`--no_warn` Turns off warning messages.

`-o` *`filename`* Names the output object file. If this option is not specified, the assembler creates an object filename of the form *`inputfilename`*`.o`.

`--predefine "`*`directive`*`"`

Instructs the assembler to pre-execute one of the `SET` directives. See *Pre-executing a SET directive* on page 3-12 for details.

`--[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 *--[no_]reduce_paths* on page 2-76 in the *RealView Compilation Tools Compiler Reference Guide* for more information.

`--regnames=none`

Instructs the assembler not to predefine register names. See *Predefined register and coprocessor names* on page 3-19 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 3-8 for details).

`--regnames=all`

Defines all AAPCS registers regardless of the value of `--apcs` (see *AAPCS* on page 3-8 for details).

 Copyright © 2002-2007 ARM Limited. All rights reserved.

`--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 3-12 for details. Use of this option is deprecated.

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

`--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 3-14).

`--untyped_local_labels`

Forces the assembler not to set the Thumb bit when referencing labels in Thumb code. See *LDR pseudo-instruction* on page 4-153 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 *RealView Compilation Tools Compiler Reference Guide*.

`--width` See *Listing output to a file* on page 3-13

`--xref` See *Listing output to a file* on page 3-13

`inputfile` Specifies the input file for the assembler. Input files must be ARM or Thumb assembly language source files.

3.1.1 Obtaining a list of available options

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

```
armasm --help
```


Copyright © 2002-2007 ARM Limited. All rights reserved.

3.1.2 Specifying command-line options with an environment variable

You can specify command-line options by setting the value of the RVCT31_ASMOPT environment variable. The syntax is identical to the command line syntax. The assembler reads the value of RVCT31_ASMOPT and inserts it at the front of the command string. This means that options specified in RVCT31_ASMOPT can be overridden by arguments on the command-line.

3.1.3 AAPCS

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

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

For more information, see the *Procedure Call Standard for the ARM Architecture* specification, aapcs.pdf, in *install_directory*\Documentation\Specifications\...

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/Thumb interworking. See Chapter 4 *Interworking ARM and Thumb* in the RealView Compilation Tools Developer Guide for information.
/nointerwork	Specifies that the code in *inputfile* is not suitable for ARM/Thumb interworking. This is the default.
/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.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

`/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.

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.

Copyright © 2002-2007 ARM Limited. All rights reserved.

`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 3-14).

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

See *RealView Compilation Tools Linker and Utilities 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:

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

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

Valid values for *name* are:

`none` Selects no floating-point architecture. This makes your assembled object file compatible with any other object file.

`vfpv3` Selects hardware floating-point unit conforming to architecture VFPv3.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

`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 equivalent to using `--fpu vfpv2`.
`softvfp+vfpv3`	Selects a floating-point library with software floating-point linkage that uses VFP instructions. This is equivalent to using `--fpu vfpv3`.

See the *RealView Compilation Tools Linker and Utilities 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 `LDR`s.
`-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 and will be removed in a future release.

Copyright © 2002-2007 ARM Limited. All rights reserved.

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.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

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.

`--[no_]project=[`*filename*`]`

Enables or disables the use of a project template file.

`--reinitialize_workdir`

Enables you to reinitialize the project template working directory.

Copyright © 2002-2007 ARM Limited. All rights reserved.

`--workdir=`*directory*

Enables you to provide a working directory for a project template.

For more information on each of these options, see:

- *--[no_]project=filename* on page 2-74 in the *RealView Compilation Tools Compiler Reference Guide*
- *--reinitialize_workdir* on page 2-77 in the *RealView Compilation Tools Compiler Reference Guide*
- *--workdir=directory* on page 2-94 in the *RealView Compilation Tools Compiler Reference Guide*.

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

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

`--diag_error` *tag*`[, `*tag*`, ...]`

Sets the diagnostic messages that have the specified tag(s) to the error severity (see Table 3-1 on page 3-16).

 Copyright © 2002-2007 ARM Limited. All rights reserved.

`--diag_remark` *tag*`[, ` *tag*`, ...]`

Sets the diagnostic messages that have the specified tag(s) to the remark severity (see Table 3-1 on page 3-16).

`--diag_warning` *tag*`[, ` *tag*`, ...]`

Sets the diagnostic messages that have the specified tag(s) to the warning severity (see Table 3-1 on page 3-16).

`--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 3-37).

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.

Copyright © 2002-2007 ARM Limited. All rights reserved.

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

Table 3-1 Severity of diagnostic messages

Severity	Description
Catastrophic error	Catastrophic errors indicate problems that cause the assembly to stop. These errors include command-line errors, internal errors, and missing include files.
Error	Errors indicate violations in the syntactic or semantic rules of assembly language. Assembly continues, but object code is not generated.
Warning	Warnings indicate unusual conditions in your code that might indicate a problem. Assembly continues, and object code is generated unless any problems with an Error severity are detected.
Remark	Remarks indicate common, but not recommended, use of assembly language. These diagnostics are not issued by default. Assembly continues, and object code is generated unless any problems with an Error severity are detected.

3.1.13 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 7-52 and *FRAME UNWIND OFF* on page 7-52.

Unwind tables

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

 Copyright © 2002-2007 ARM Limited. All rights reserved.

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.

Copyright © 2002-2007 ARM Limited. All rights reserved.

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 3-26 and *Local labels* on page 3-27). 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 3-23).

 Copyright © 2002-2007 ARM Limited. All rights reserved.

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).

Copyright © 2002-2007 ARM Limited. All rights reserved.

3.3.3 Predeclared XScale register names

The following register name is 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`
- `wCID`, `wcid`, and `WCID`
- `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).

Copyright © 2002-2007 ARM Limited. All rights reserved.

3.4 Built-in variables and constants

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

Table 3-2 Built-in variables

{ARCHITECTURE}	Holds the name of the selected ARM architecture.
{AREANAME}	Holds the name of the current AREA.
{ARMASM_VERSION}	Holds an integer that increases with each version of armasm.
|ads$version|	Has the same value as {ARMASM_VERSION}.
{CODESIZE}	Is a synonym for {CONFIG}.
{COMMANDLINE}	Holds the contents of the command line.
{CONFIG}	Has the value 32 if the assembler is assembling ARM code, or 16 if it is assembling Thumb code.
{CPU}	Holds the name of the selected cpu. The default is ARM7TDMI. If an architecture was specified in the command line --cpu option, {CPU} holds the value "Generic ARM".
{ENDIAN}	Has the value big if the assembler is in big-endian mode, or little if it is in little-endian mode.
{FPIC}	Has the value True if /fpic is set. The default is False.
{FPU}	Holds the name of the selected fpu. The default is SoftVFP.
{INPUTFILE}	Holds the name of the current source file.
{INTER}	Has the value True if /inter is set. The default is False.
{LINENUM}	Holds an integer indicating the line number in the current source file.
{OPT}	Value of the currently-set listing option. The OPT directive can be used to save the current listing option, force a change in it, or restore its original value.
{PC} or .	Address of current instruction.
{PCSTOREOFFSET}	Is the offset between the address of the STR pc,[...] or STM Rb,{..., pc} instruction and the value of pc stored out. This varies depending on the CPU or architecture specified.
{ROPI}	Has the value True if /ropi is set. The default is False.
{RWPI}	Has the value True if /rwpi is set. The default is False.
{VAR} or @	Current value of the storage area location counter.

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

Copyright © 2002-2007 ARM Limited. All rights reserved.

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

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

You can use the built-in variable {ARMASM_VERSION} to distinguish between versions of armasm.

The ARM assembler did not have the built-in variable {ARMASM_VERSION} before ADS. 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 RVCT or ADS
ELSE
  ; code for SDT
ENDIF
```

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

Table 3-3 Built-in Boolean constants

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

Copyright © 2002-2007 ARM Limited. All rights reserved.

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 3-24
- *Numeric constants* on page 3-24
- *Assembly time substitution of variables* on page 3-24
- *Labels* on page 3-26
- *Local labels* on page 3-27.

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 3-27).
- Symbols must not use the same name as built-in variable names or predefined symbol names (see *Predefined register and coprocessor names* on page 3-19 and *Built-in variables and constants* on page 3-21).
- 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.

Copyright © 2002-2007 ARM Limited. All rights reserved.

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 3-30).

The possible values of a logical variable are `{TRUE}` or `{FALSE}` (see *Logical expressions* on page 3-33).

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 3-29).

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

Use the `EQU` directive to define constants (see *EQU* on page 7-70). You cannot change the value of a numeric constant after you define it.

See also *Numeric expressions* on page 3-30 and *Numeric literals* on page 3-31.

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.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

Numeric and logical variables can also be substituted. The current value of the variable is converted to a hexadecimal string (or T or F for logical variables) before substitution.

Use a dot to mark the end of the variable name if the following character would be permissible in a symbol name (see *Symbol naming rules* on page 3-23). You must set the contents of the variable before you can use it.

If you require a $ that you do not want to be substituted, use $$. This is converted to a single $.

You can include a variable with a $ prefix in a string. Substitution occurs in the same way as anywhere else.

Substitution does not occur within vertical bars, except that vertical bars within double quotes do not affect substitution.

Examples

```
    ; straightforward substitution
        GBLS    add4ff
        ;
add4ff  SETS    "ADD  r4,r4,#0xFF"    ; set up add4ff
        $add4ff.00                    ; invoke add4ff
        ; this produces
        ADD  r4,r4,#0xFF00

    ; elaborate substitution
            GBLS    s1
            GBLS    s2
            GBLS    fixup
            GBLA    count
            ;
count       SETA    14
s1          SETS    "a$$b$count" ; s1 now has value a$b0000000E
s2          SETS    "abc"
fixup       SETS    "|xy$s2.z|"  ; fixup now has value |xyabcz|
|C$$code|   MOV     r4,#16       ; but the label here is C$$code
```


Copyright © 2002-2007 ARM Limited. All rights reserved.

3.5.5 Labels

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

Program-relative labels

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

- *DCB* on page 7-22
- *DCD and DCDU* on page 7-23
- *DCFD and DCFDU* on page 7-25
- *DCFS and DCFSU* on page 7-26
- *DCI* on page 7-27
- *DCQ and DCQU* on page 7-28
- *DCW and DCWU* on page 7-29.

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 7-19
- *SPACE* on page 7-21
- *DCDO* on page 7-24
- *EQU* on page 7-70.

Absolute addresses

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

Copyright © 2002-2007 ARM Limited. All rights reserved.

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 3-18):

- 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 3-26).

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 7-81). 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.

Copyright © 2002-2007 ARM Limited. All rights reserved.

Syntax

The syntax of a local label is:

n{*routname*}

The syntax of a reference to a local label is:

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

where:

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

 Copyright © 2002-2007 ARM Limited. All rights reserved.

3.6 Expressions, literals, and operators

This section contains the following subsections:

3.6.1 String expressions

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

- *Variables* on page 3-24
- *String literals* on page 3-30
- *Unary operators* on page 3-35
- *String manipulation operators* on page 3-37
- *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
```


Copyright © 2002-2007 ARM Limited. All rights reserved.

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 3-18).

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 3-24
- *Variables* on page 3-24
- *Numeric literals* on page 3-31
- *Binary operators* on page 3-37
- *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 >= use the unsigned interpretation. This means that $0 > -1$ is `{FALSE}`.

Example

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

 Copyright © 2002-2007 ARM Limited. All rights reserved.

3.6.4 Numeric literals

Numeric literals can take any of the following forms:

decimal-digits

0x*hexadecimal-digits*

&*hexadecimal-digits*

n_base-n-digits

'*character*'

where:

decimal-digits	Is a sequence of characters using only the digits 0 to 9.
hexadecimal-digits	Is a sequence of characters using only the digits 0 to 9 and the letters A to F or a to f.
*n_*	Is a single digit between 2 and 9 inclusive, followed by an underscore character.
base-n-digits	Is a sequence of characters using only the digits 0 to (n –1)
character	Is any single character except a single quote. Use \' if you require a single quote. In this case the value of the numeric literal is the numeric code of the character.

You must not use any other characters. The sequence of characters must evaluate to an integer in the range 0 to 2^{32}–1 (except in DCQ and DCQU directives, where the range is 0 to 2^{64}–1).

Examples

```
a       SETA    34906
addr    DCD     0xA10E
        LDR     r4,=&1000000F
        DCD     2_11001010
c3      SETA    8_74007
        DCQ     0x0123456789abcdef
        LDR     r1,='A'             ; pseudo-instruction loading 65 into r1
        ADD     r3,r2,#'\''         ; add 39 to contents of r2, result to r3
```


Copyright © 2002-2007 ARM Limited. All rights reserved.

3.6.5 Floating-point literals

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

```
{-}digitsE{-}digits
{-}{digits}.digits{E{-}digits}
0xhexdigits
&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

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

 Copyright © 2002-2007 ARM Limited. All rights reserved.

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 7-19).

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

Note

The value used in program-relative addresses is:

- the address of the instruction *following* the instruction being executed
- OR `0xFFFFFFFC` (this makes no difference in ARM code)
- plus or minus the numeric constant.

Example

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

3.6.7 Logical expressions

Logical expressions consist of combinations of logical literals (`{TRUE}` or `{FALSE}`), logical variables, Boolean operators, relations, and parentheses (see *Boolean operators* on page 3-41).

Relations consist of combinations of variables, literals, constants, or expressions with appropriate relational operators (see *Relational operators* on page 3-40).

3.6.8 Logical literals

The logical literals are:

- `{TRUE}`
- `{FALSE}`.

Copyright © 2002-2007 ARM Limited. All rights reserved.

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 3-35 and *Binary operators* on page 3-37).

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 3-4 shows the order of precedence of operators in `armasm`, and a comparison with the order in C (see Table 3-5 on page 3-35).

From these tables:

- The highest precedence operators are at the top of the list.
- The highest precedence operators are evaluated first.
- Operators of equal precedence are evaluated from left to right.

Table 3-4 Operator precedence in armasm

armasm precedence	equivalent C operators
unary operators	unary operators
`* / :MOD:`	`* / %`
string manipulation	n/a
`:SHL: :SHR: :ROR: :ROL:`	`<< >>`

Copyright © 2002-2007 ARM Limited. All rights reserved.

Table 3-4 Operator precedence in armasm (continued)

armasm precedence	equivalent C operators
+ - :AND: :OR: :EOR:	+ - & |
= > >= < <= /= <>	== > >= < <= !=
:LAND: :LOR: :LEOR:	&& ||

Table 3-5 Operator precedence in C

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

3.6.10 Unary operators

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

Copyright © 2002-2007 ARM Limited. All rights reserved.

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

Table 3-6 Unary operators that return strings

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

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

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

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

 Copyright © 2002-2007 ARM Limited. All rights reserved.

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

3.6.11 Binary operators

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

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

Note

The order of precedence is not the same as in C, see *Operator precedence in armasm and C* on page 3-34.

Multiplicative operators

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

Table 3-8 shows the multiplicative operators.

Table 3-8 Multiplicative operators

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

String manipulation operators

Table 3-9 on page 3-38 shows the string manipulation operators.

In the slicing operators LEFT and RIGHT:

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

Copyright © 2002-2007 ARM Limited. All rights reserved.

In CC, A and B must both be strings.

Table 3-9 String manipulation operators

Operator	Usage	Explanation
:CC:	A:CC:B	B concatenated onto the end of A
:LEFT:	A:LEFT:B	The left-most B characters of A
:RIGHT:	A:RIGHT:B	The right-most B characters of A

Copyright © 2002-2007 ARM Limited. All rights reserved.

Shift operators

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

Table 3-10 shows the shift operators.

Table 3-10 Shift operators

Operator	Alias	Usage	Explanation
:ROL:		A:ROL:B	Rotate A left by B bits
:ROR:		A:ROR:B	Rotate A right by B bits
:SHL:	<<	A:SHL:B	Shift A left by B bits
:SHR:	>>	A:SHR:B	Shift A right by B bits

Note

SHR is a logical shift and does not propagate the sign bit.

Addition, subtraction, and logical operators

Addition and subtraction operators act on numeric expressions.

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

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

Table 3-11 Addition, subtraction, and logical operators

Operator	Alias	Usage	Explanation
+		A+B	Add A to B
-		A-B	Subtract B from A
:AND:	&&	A:AND:B	Bitwise AND of A and B
:EOR:	^	A:EOR:B	Bitwise Exclusive OR of A and B
:OR:	||	A:OR:B	Bitwise OR of A and B

Copyright © 2002-2007 ARM Limited. All rights reserved.

Relational operators

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

The operands can be one of:

- numeric
- program-relative
- register-relative
- strings.

Strings are sorted using ASCII ordering. String `A` is less than string `B` if it is a leading substring of string `B`, or if the left-most character in which the two strings differ is less in string `A` than in string `B`.

Arithmetic values are unsigned, so the value of `0>-1` is `{FALSE}`.

Table 3-12 Relational operators

Operator	Alias	Usage	Explanation
`=`	`==`	`A=B`	A equal to B
`>`		`A>B`	A greater than B
`>=`		`A>=B`	A greater than or equal to B
`<`		`A<B`	A less than B
`<=`		`A<=B`	A less than or equal to B
`/=`	`<> !=`	`A/=B`	A not equal to B

Copyright © 2002-2007 ARM Limited. All rights reserved.

Boolean operators

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

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

Table 3-13 shows the Boolean operators.

Table 3-13 Boolean operators

Operator	Usage	Explanation
:LAND:	A:LAND:B	Logical AND of A and B
:LEOR:	A:LEOR:B	Logical Exclusive OR of A and B
:LOR:	A:LOR:B	Logical OR of A and B

Copyright © 2002-2007 ARM Limited. All rights reserved.

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 3-14 for details.

This section contains the following subsections:

- *Interlocks*
- *IT block generation*
- *Thumb branch target alignment* on page 3-43.

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
```

 Copyright © 2002-2007 ARM Limited. All rights reserved.

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
```


Copyright © 2002-2007 ARM Limited. All rights reserved.

3.8 Using the C preprocessor

You can use C preprocessor commands in your assembly language source file. If you do this, you must preprocess the file using the C preprocessor, before using `armasm` to assemble it. See the *RealView Compilation Tools Compiler User Guide* for more information.

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

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

Example 3-1 Preprocessing an assembly language source file

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


Copyright © 2002-2007 ARM Limited. All rights reserved.

Chapter 4
ARM and Thumb Instructions

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

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,.

Copyright © 2002-2007 ARM Limited. All rights reserved.

4.1 Instruction summary

Table 4-1 gives an overview of the instructions available in the ARM, Thumb, Thumb-2, 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

Mnemonic	Brief description	Page	Arch. [a]
ADC, ADD	Add with Carry, Add	page 4-44	All
ADR	Load program or register-relative address (short range)	page 4-22	All
ADRL pseudo-instruction	Load program or register-relative address (medium range)	page 4-149	x6M
AND	Logical AND	page 4-50	All
ASR	Arithmetic Shift Right	page 4-66	All
B	Branch	page 4-115	All
BFC, BFI	Bit Field Clear and Insert	page 4-107	T2
BIC	Bit Clear	page 4-50	All
BKPT	Breakpoint	page 4-129	5
BL	Branch with Link	page 4-115	All
BLX	Branch with Link, change instruction set	page 4-115	T
BX	Branch, change instruction set	page 4-115	T
BXJ	Branch, change to Jazelle	page 4-115	J, x7M
CBZ, CBNZ	Compare and Branch if {Non}Zero	page 4-118	T2
CDP	Coprocessor Data Processing operation	page 4-121	x6M
CDP2	Coprocessor Data Processing operation	page 4-121	5, x6M
CHKA	Check array	page 4-146	EE
CLREX	Clear Exclusive	page 4-38	K, x6M

Copyright © 2002-2007 ARM Limited. All rights reserved.

Table 4-1 Location of instructions (continued)

Mnemonic	Brief description	Page	Arch. [a]
CLZ	Count leading zeros	page 4-53	5, x6M
CMN, CMP	Compare Negative, Compare	page 4-54	All
CPS	Change Processor State	page 4-135	6
CPY	Copy	page 4-56	6
DBG	Debug	page 4-141	7
DMB, DSB	Data Memory Barrier, Data Synchronization Barrier	page 4-141	7, 6M
ENTERX, LEAVEX	Change state to or from ThumbEE	page 4-145	EE
EOR	Exclusive OR	page 4-50	All
HB, HBL, HBLP, HBP	Handler Branch, branches to a specified handler	page 4-147	EE
ISB	Instruction Synchronization Barrier	page 4-141	7, 6M
IT	If-Then	page 4-68	T2
LDC	Load Coprocessor	page 4-126	x6M
LDC2	Load Coprocessor	page 4-126	5, x6M
LDM	Load Multiple registers	page 4-26	All
LDR	Load Register instructions	page 4-11	All
LDR pseudo-instruction	Load Register pseudo-instruction	page 4-153	All
LDREX	Load Register Exclusive	page 4-35	6, x6M
LDREXB, LDREXH	Load Register Exclusive Byte, Halfword	page 4-35	K, x6M
LDREXD	Load Register Exclusive Doubleword	page 4-35	K, x7M
LSL, LSR	Logical Shift Left, Logical Shift Right	page 4-66	All
MAR	Move from Registers to 40-bit Accumulator	page 4-143	XScale
MCR	Move from Register to Coprocessor	page 4-122	x6M
MCR2	Move from Register to Coprocessor	page 4-122	5, x6M
MCRR	Move from Registers to Coprocessor	page 4-122	5E, x6M
MCRR2	Move from Registers to Coprocessor	page 4-122	6, x6M

Copyright © 2002-2007 ARM Limited. All rights reserved.

Table 4-1 Location of instructions (continued)

Mnemonic	Brief description	Page	Arch. [a]
MIA, MIAPH, MIAxy	Multiply with Internal 40-bit Accumulate	page 4-91	XScale
MLA	Multiply Accumulate	page 4-73	x6M
MLS	Multiply and Subtract	page 4-73	T2
MOV	Move	page 4-56	All
MOVT	Move Top	page 4-59	T2
MOV32 pseudo-instruction	Move 32-bit constant to register	page 4-151	T2
MRA	Move from 40-bit Accumulator to Registers	page 4-143	XScale
MRC	Move from Coprocessor to Register	page 4-124	All
MRC2	Move from Coprocessor to Register	page 4-124	5, x6M
MRS	Move from PSR to register	page 4-131	All
MSR	Move from register to PSR	page 4-133	All
MUL	Multiply	page 4-73	All
MVN	Move Not	page 4-56	All
NOP	No Operation	page 4-139	All
ORN	Logical OR NOT	page 4-50	T2
ORR	Logical OR	page 4-50	All
PKHBT, PKHTB	Pack Halfwords	page 4-112	6, x7M
PLD	Preload Data	page 4-24	5E, x6M
PLI	Preload Instruction	page 4-24	7
PUSH, POP	PUSH, POP registers	page 4-29	All
QADD, QDADD, QDSUB, QSUB	Saturating Arithmetic	page 4-94	5E, x7M
QADD8, QADD16, QASX, QSUB8, QSUB16, QSAX	Parallel signed Saturating Arithmetic	page 4-99	6, x7M
REV, REV16, REVSH	Reverse byte order	page 4-64	6
RBIT	Reverse Bits	page 4-64	T2

Copyright © 2002-2007 ARM Limited. All rights reserved.

Table 4-1 Location of instructions (continued)

Mnemonic	Brief description	Page	Arch. [a]
RFE	Return From Exception	page 4-31	T2, x7M
ROR	Rotate Right Register	page 4-66	All
RSB	Reverse Subtract	page 4-44	All
RSC	Reverse Subtract with Carry	page 4-44	x6M
SBC	Subtract with Carry	page 4-44	All
SADD8, SADD16, SASX	Parallel signed arithmetic	page 4-99	6, x7M
SBFX, UBFX	Signed, Unsigned Bit Field eXtract	page 4-108	T2
SDIV	Signed divide	page 4-71	7M, 7R
SEL	Select bytes according to APSR GE flags	page 4-62	6, x7M
SEV	Set Event	page 4-139	K, 6M
SETEND	Set Endianness for memory accesses	page 4-138	6, x7M
SHADD8, SHADD16, SHASX, SHSUB8, SHSUB16, SHSAX	Parallel signed Halving arithmetic	page 4-99	6, x7M
SMC	Secure Monitor Call	page 4-137	Z
SMLAD	Dual Signed Multiply Accumulate (32 <= 32 + 16 x 16 + 16 x 16)	page 4-86	6, x7M
SMLAL	Signed Multiply Accumulate (64 <= 64 + 32 x 32)	page 4-75	x6M
SMLALxy	Signed Multiply Accumulate (64 <= 64 + 16 x 16)	page 4-80	5E, x7M
SMLALD	Dual Signed Multiply Accumulate Long (64 <= 64 + 16 x 16 + 16 x 16)	page 4-88	6, x7M
SMLSD	Dual Signed Multiply Subtract Accumulate (32 <= 32 + 16 x 16 – 16 x 16)	page 4-86	6, x7M
SMLSLD	Dual Signed Multiply Subtract Accumulate Long (64 <= 64 + 16 x 16 – 16 x 16)	page 4-88	6, x7M
SMMUL	Signed top word Multiply (32 <= TopWord(32 x 32))	page 4-84	6, x7M

Copyright © 2002-2007 ARM Limited. All rights reserved.

Table 4-1 Location of instructions (continued)

Mnemonic	Brief description	Page	Arch. [a]
SMUAD, SMUSD	Dual Signed Multiply, and Add or Subtract products	page 4-82	6, x7M
SMULL	Signed Multiply (64 <= 32 x 32)	page 4-75	x6M
SMULxy	Signed Multiply (32 <= 16 x 16)	page 4-77	5E, x7M
SMULWy	Signed Multiply (32 <= 32 x 16)	page 4-79	5E, x7M
SRS	Store Return State	page 4-33	T2, x7M
SSAT	Signed Saturate	page 4-96	6, x6M
SSAT16	Signed Saturate, parallel halfwords	page 4-104	6, x7M
SSUB8, SSUB16, SSAX	Parallel signed arithmetic	page 4-99	6, x7M
STC	Store Coprocessor	page 4-126	x6M
STC2	Store Coprocessor	page 4-126	5, x6M
STM	Store Multiple registers	page 4-26	All
STR	Store Register instructions	page 4-11	All
STREX	Store Register Exclusive	page 4-35	6, x6M
STREXB, STREXH	Store Register Exclusive Byte, Halfword	page 4-35	K, x6M
STREXD	Store Register Exclusive Doubleword	page 4-35	K, x7M
SUB	Subtract	page 4-44	All
SUBS pc, LR	Exception return, no stack	page 4-48	T2, x7M
SVC (formerly SWI)	SuperVisor Call	page 4-130	All
SWP, SWPB	Swap registers and memory (ARM only)	page 4-39	All, x7M
SXT	Signed extend	page 4-109	6
SXTA	Signed extend, with Addition	page 4-109	6, x7M
TBB, TBH	Table Branch Byte, Halfword	page 4-119	T2
TEQ, TST	Test Equivalence, Test	page 4-60	All
UADD8, UADD16, UASX	Parallel Unsigned Arithmetic	page 4-99	6, x7M
UDIV	Unsigned divide	page 4-71	7M, 7R

Copyright © 2002-2007 ARM Limited. All rights reserved.

Table 4-1 Location of instructions (continued)

Mnemonic	Brief description	Page	Arch. [a]
UHADD8, UHADD16, UHASX, UHSUB8, UHSUB16, UHSAX	Parallel Unsigned Halving Arithmetic	page 4-99	6, x7M
UMAAL	Unsigned Multiply Accumulate Accumulate Long (64 <= 32 + 32 + 32 x 32)	page 4-90	6, x7M
UMLAL, UMULL	Unsigned Multiply Accumulate, Multiply (64 <= 32 x 32 + 64), (64 <= 32 x 32)	page 4-75	x6M
UQADD8, UQADD16, UQASX, UQSUB8, UQSUB16, UQSAX	Parallel Unsigned Saturating Arithmetic	page 4-99	6, x7M
USAD8	Unsigned Sum of Absolute Differences	page 4-102	6, x7M
USADA8	Accumulate Unsigned Sum of Absolute Differences	page 4-102	6, x7M
USAT	Unsigned Saturate	page 4-96	6, x6M
USAT16	Unsigned Saturate, parallel halfwords	page 4-104	6, x7M
USUB8, USUB16, USAX	Parallel unsigned arithmetic	page 4-99	6, x7M
UXT	Unsigned extend	page 4-109	6
UXTA	Unsigned extend with Addition	page 4-109	6, x7M
V*	See Chapter 5 *NEON and VFP Programming*		
WFE, WFI, YIELD	Wait For Event, Wait For Interrupt, Yield	page 4-139	T2, 6M

Copyright © 2002-2007 ARM Limited. All rights reserved.

a. Entries in the Architecture column have the following meanings:

All	These instructions are available in all versions of the ARM architecture.
5	These instructions are available in the ARMv5T*, ARMv6*, and ARMv7 architectures.
5E	These instructions are available in the ARMv5TE, ARMv6*, and ARMv7 architectures.
6	These instructions are available in the ARMv6* and ARMv7 architectures.
6M	These instructions are available in the ARMv6-M and ARMv7 architectures.
x6M	These instructions are not available in the ARMv6-M profile.
7	These instructions are available in the ARMv7 architectures.
7M	These instructions are available in the ARMv7-M profile.
x7M	These instructions are not available in the ARMv6-M or ARMv7-M profile.
7R	These instructions are available in the ARMv7-R profile.
EE	These instructions are available in ThumbEE variants of the ARM architecture.
J	This instruction is available in the ARMv5TEJ, ARMv6*, and ARMv7 architectures.
K	These instructions are available in the ARMv6K, and ARMv7 architectures.
T	These instructions are available in ARMv4T, ARMv5T*, ARMv6*, and ARMv7 architectures.
T2	These instructions are available in the ARMv6T2 and ARMv7 architectures.
XScale	These instructions are available in XScale versions of the ARM architecture.
Z	This instruction is available if Security Extensions are implemented.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

4.2 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.)

4.2.1 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
```

4.2.2 Different behavior for some instructions

For forward references, `LDR`, `ADR`, and `B` without `.W` alway 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.

Copyright © 2002-2007 ARM Limited. All rights reserved.

4.2.3 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.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

4.3 Memory access instructions

This section contains the following subsections:

Copyright © 2002-2007 ARM Limited. All rights reserved.

- *CLREX* on page 4-38

  Clear Exclusive.

- *SWP and SWPB* on page 4-39

  Swap data between registers and memory.

Note

There is also an `LDR` pseudo-instruction (see *LDR pseudo-instruction* on page 4-153). This pseudo-instruction either assembles to an `LDR` instruction, or to a `MOV` or `MVN` instruction.

4.3.1 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.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

4.3.2 LDR and STR (immediate offset)

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

Syntax

```
op{type}{cond} Rt, [Rn {, #offset}]        ; immediate offset
op{type}{cond} Rt, [Rn, #offset]!          ; pre-indexed
op{type}{cond} Rt, [Rn], #offset           ; post-indexed
opD{cond} Rt, Rt2, [Rn {, #offset}]        ; immediate offset, doubleword
opD{cond} Rt, Rt2, [Rn, #offset]!          ; pre-indexed, doubleword
opD{cond} Rt, Rt2, [Rn], #offset           ; post-indexed, doubleword
```

where:

op can be either:
- LDR Load Register
- STR Store Register.

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 2-17).

Rt is the register to load or store.

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

offset is an offset. If *offset* is omitted, the address is the contents of *Rn*.

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 ranges and architectures* on page 4-14 for details.

Copyright © 2002-2007 ARM Limited. All rights reserved.

Offset ranges and architectures

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

Table 4-2 Offsets and architectures, LDR/STR, word, halfword, and byte

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

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 ARMv5 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.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

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 *Rt*
- *Rt2* must be R(*t* + 1).

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.
```


Copyright © 2002-2007 ARM Limited. All rights reserved.

4.3.3 LDR and STR (register offset)

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

Syntax

```
op{type}{cond} Rt, [Rn, +/-Rm {, shift}]   ; register offset
op{type}{cond} Rt, [Rn, +/-Rm {, shift}]!  ; pre-indexed
op{type}{cond} Rt, [Rn], +/-Rm {, shift}   ; post-indexed
opD{cond} Rt, Rt2, [Rn, +/-Rm {, shift}]   ; register offset, doubleword
opD{cond} Rt, Rt2, [Rn, +/-Rm {, shift}]!  ; pre-indexed, doubleword
opD{cond} Rt, Rt2, [Rn], +/-Rm {, shift}   ; post-indexed, doubleword
```

where:

op can be either:

- LDR Load Register
- STR Store Register.

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 2-17).

Rt is the register to load or store.

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

Rm is a register containing a value to be used as the offset. *Rm* must not be r15. *-Rm* is not allowed in Thumb code.

shift is an optional shift.

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 4-17 for details.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

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)

Instruction	*+/–Rm* [a]	shift	Arch.
ARM, word or byte [b]	*+/–Rm*	`LSL #0-31` `LSR #1-32` `ASR #1-32` `ROR #1-31` `RRX`	All
ARM, signed byte, halfword, or signed halfword	*+/–Rm*	Not available	All
ARM, doubleword	*+/–Rm*	Not available	v5TE +
32-bit Thumb, word, halfword, signed halfword, byte, or signed byte [a]	*+Rm*	`LSL #0-3`	v6T2, v7
32-bit Thumb, doubleword	*+Rm*	Not available	v6T2, v7
16-bit Thumb, all [c]	*+Rm*	Not available	All T
16-bit ThumbEE, word [b]	*+Rm*	`LSL #2` (required)	T-2EE
16-bit ThumbEE, halfword, signed halfword [b]	*+Rm*	`LSL #1` (required)	T-2EE
16-bit ThumbEE , byte , signed byte [b]	*+Rm*	Not available	T-2EE

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 ARMv5 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.

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 *Rt*
- *Rt2* must be `R(t + 1)`.

Copyright © 2002-2007 ARM Limited. All rights reserved.

4.3.4 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

```
op{type}T{cond} Rt, [Rn {, #offset}]       ; immediate offset (Thumb-2 only)
op{type}T{cond} Rt, [Rn] {, #offset}       ; post-indexed (ARM only)
op{type}T{cond} Rt, [Rn], +/-Rm {, shift}  ; post-indexed (register) (ARM only)
```

where:

op can be either:

- `LDR` Load Register
- `STR` Store Register.

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 2-17).

Rt is the register to load or store.

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

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

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

shift is an optional shift.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

Offset ranges and architectures

Table 4-2 on page 4-14 shows the ranges of offsets and availability of these instructions.

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

Instruction	Immediate offset	Post-indexed	*+/–Rm* [a]	shift	Arch.

ARM, word or byte	Not available	–4095 to 4095	*+/–Rm*	`LSL #0-31`
				`LSR #1-32`
				`ASR #1-32`
				`ROR #1-31`
				`RRX`
					All

ARM, signed byte, halfword, or signed halfword	Not available	–255 to 255	*+/–Rm*	Not available	All
32-bit Thumb, word, halfword, signed halfword, byte, or signed byte	0 to 255	Not available	Not available		v6T2, v7

a. You can use –Rm, +Rm, or Rm.

Copyright © 2002-2007 ARM Limited. All rights reserved.

4.3.5 LDR (pc-relative)

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

Note

See also *Pseudo-instructions* on page 4-148.

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 2-17).

.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 3-33 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.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

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

Instruction	Offset range	Architectures
ARM LDR, LDRB, LDRSB, LDRH, LDRSH [a]	+/– 4095	All
ARM LDRD	+/– 255	v5TE +
32-bit Thumb LDR, LDRB, LDRSB, LDRH, LDRSH [a]	+/– 4095	v6T2, v7
32-bit Thumb LDRD	+/– 1020 [b]	v6T2, v7
16-bit Thumb LDR [c]	0-1020 [b]	All T

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 ARMv5 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.

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:

- *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).

Copyright © 2002-2007 ARM Limited. All rights reserved.

4.3.6 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 2-17).

.W is an optional instruction width specifier. See *ADR in Thumb-2* on page 4-23 for details.

Rd is the register to load.

label is a program-relative expression. See *Register-relative and program-relative expressions* on page 3-33 for more information.

label must be within a limited distance of the current instruction. See *Offset range and architectures* on page 4-23 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 4-149).

If *label* is program-relative, it must evaluate to an address in the same assembler area as the ADR instruction, see *AREA* on page 7-66.

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.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

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 4-6 pc-relative offsets

Instruction	Offset range	Architectures
ARM ADR	See *Constants in Operand2* on page 4-42	All
32-bit Thumb ADR	+/– 4095	v6T2, v7
16-bit Thumb ADR [a]	0-1020 [b]	All T

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 ADD instruction.

Copyright © 2002-2007 ARM Limited. All rights reserved.

4.3.7 PLD 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.

Implementation of PLD and PLI is optional. If they are not implemented, they execute as NOPs.

Syntax

PL*type*{*cond*} [*Rn* {, #*offset*}]

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

PL*type*{*cond*} *label*

where:

type is either D (data) or I (instruction).

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

Note

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

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

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

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

shift is an optional shift.

label is a program-relative expression. See *Register-relative and program-relative expressions* on page 3-33 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.

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

 Copyright © 2002-2007 ARM Limited. All rights reserved.

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. ARM PLI is available in ARMv7.

32-bit Thumb PLD is available in ARMv6T2 and above. 32-bit Thumb PLI is available in ARMv7.

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

Copyright © 2002-2007 ARM Limited. All rights reserved.

4.3.8 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 4-29.

Syntax

`op{addr_mode}{cond} Rn{!}, reglist{^}`

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.

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

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 4-28).

See *Restrictions on reglist in 32-bit Thumb-2 instructions* on page 4-27.

`^` 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 `LDM` (or `LDMIA`) and *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.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

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.

16-bit instructions

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

The following restrictions apply to the 16-bit instructions:

- all registers in *`reglist`* must be Lo registers
- *`Rn`* must be a Lo register
- *`addr_mode`* must be omitted (or `IA`), meaning increment address after each transfer
- writeback must be specified.

In addition, the `PUSH` and `POP` instructions can be expressed in this form. Some forms of `PUSH` and `POP` are also 16-bit instructions. See *PUSH and POP* on page 4-29 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 ARMv5 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.

Copyright © 2002-2007 ARM Limited. All rights reserved.

Loading or storing the base register, with writeback

In ARM code or Thumb-1 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 Thumb-1 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
```


Copyright © 2002-2007 ARM Limited. All rights reserved.

4.3.9 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 2-17).

`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 r13 (sp), and the adjusted address written back to the base register. `PUSH` and `POP` are the preferred mnemonic in these cases.

Registers are stored on the stack in numerical order, with the lowest numbered register at the lowest address.

POP, with reglist including the pc

This instruction causes a branch to the address popped off the stack into 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 ARMv5 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.

Copyright © 2002-2007 ARM Limited. All rights reserved.

16-bit instructions

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

The following restriction applies to the 16-bit instructions:

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

Examples

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


Copyright © 2002-2007 ARM Limited. All rights reserved.

4.3.10 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 2-17).

Note

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

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

`!` is an optional suffix. If `!` is present, the final address is written back into *Rn*.

Usage

You can use `RFE` to return from an exception if you previously saved the return state using the `SRS` instruction (see *SRS* on page 4-33).

In Thumb-2EE, if the value in the base register is zero, execution branches to the NullCheck handler at HandlerBase - 4.

Operation

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

Copyright © 2002-2007 ARM Limited. All rights reserved.

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.

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

If Monitor mode is specified by *mode*, the result is unpredictable (see *SMC* on page 4-137).

Architectures

This ARM instruction is available in ARMv6 and above.

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

There is no 16-bit version of this instruction.

Example

```
RFE r13!
```

 Copyright © 2002-2007 ARM Limited. All rights reserved.

4.3.11 SRS

Store Return State.

Syntax

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

where:

addr_mode is any one of the following:

- `IA` Increment address After each transfer
- `IB` Increment address Before each transfer (ARM only)
- `DA` Decrement address After each transfer (ARM only)
- `DB` Decrement address Before each transfer.

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

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

Note

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

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

modenum specifies the number of the mode whose banked r13 is used as the base register, see *Processor mode* on page 2-5.

Operation

`SRS` stores the r14 and the SPSR of the current mode, at the address contained in r13 of the mode specified by *modenum*, and the following word respectively. Optionally updates r13 of the mode specified by *modenum*. This is compatible with the normal use of the `STM` instruction for stack accesses, see *LDM and STM* on page 4-26.

Usage

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

In Thumb-2EE, if the value in the base register is zero, execution branches to the NullCheck handler at HandlerBase - 4.

Copyright © 2002-2007 ARM Limited. All rights reserved.

Notes

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

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

SRS is unpredictable in User and System modes, because they do not have SPSRs.

Architectures

This ARM instruction is available in ARMv6 and above.

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

There is no 16-bit version of this instruction.

Example

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


Copyright © 2002-2007 ARM Limited. All rights reserved.

4.3.12 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 2-17).

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 only allowed 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.

Copyright © 2002-2007 ARM Limited. All rights reserved.

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

offset is not allowed in ARM instructions. The value of *offset* can be any multiple of four in the range 0-1020.

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 LDREX, *Rt* and *Rt2* must not be the same register.

In the ARM instructions:

- *Rt* must be an even numbered register, and not r14
- *Rt2* must be *R(d+1)*.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

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 ARMv7, except that LDREXD and STREXD are not available in the ARMv7-M profile.

There are no 16-bit versions of these instructions.

Examples

```
    MOV r1, #0x1               ; load the 'lock taken' value
try
    LDREX r0, [LockAddr]       ; load the lock value
    CMP r0, #0                 ; is the lock free?
    STREXEQ r0, r1, [LockAddr] ; try and claim the lock
    CMPEQ r0, #0               ; did this succeed?
    BNE try                    ; no - try again
    ....                       ; yes - we have the lock
```


Copyright © 2002-2007 ARM Limited. All rights reserved.

4.3.13 CLREX

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

Syntax

`CLREX{`*cond*`}`

where:

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

Note

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

Usage

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

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

Architectures

This ARM instruction is available in ARMv6K and above.

This 32-bit Thumb-2 instruction is available in T2 variants of ARMv6K and above.

There is no 16-bit Thumb `CLREX` instruction.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

4.3.14 SWP and SWPB

Swap data between registers and memory.

Syntax

`SWP{B}{cond} Rt, Rt2, [Rn]`

where:

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

B is an optional suffix. If B is present, a byte is swapped. Otherwise, a 32-bit word is swapped.

Rt is the destination register.

Rt2 is the source register. *Rt2* can be the same register as *Rt*.

Rn contains the address in memory. *Rn* must be a different register from both *Rt* and *Rt2*.

Usage

You can use SWP and SWPB to implement semaphores:

- Data from memory is loaded into *Rt*.
- The contents of *Rt2* is saved to memory.
- If *Rt2* is the same register as *Rt*, the contents of the register is swapped with the contents of the memory location.

Note

The use of SWP and SWPB is deprecated in ARMv6 and above. See *LDREX and STREX* on page 4-35 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 SWP or SWPB instructions.

Copyright © 2002-2007 ARM Limited. All rights reserved.

4.4 General data processing instructions

This section contains the following subsections:

 Copyright © 2002-2007 ARM Limited. All rights reserved.

4.4.1 Flexible second operand

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

Syntax

Operand2 has two possible forms:

`#constant`

`Rm{, shift}`

where:

constant is an expression evaluating to a numeric constant. The range of constants available in ARM and Thumb-2 is not exactly the same. See *Constants in Operand2* on page 4-42 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. $1 \le n \le 32$.
- `LSL #n` logical shift left *n* bits. $0 \le n \le 31$.
- `LSR #n` logical shift right *n* bits. $1 \le n \le 32$.
- `ROR #n` rotate right *n* bits. $1 \le n \le 31$.
- `RRX` rotate right one bit, with extend.
- *type Rs* 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.

Copyright © 2002-2007 ARM Limited. All rights reserved.

Constants in Operand2

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

In 32-bit Thumb-2 instructions, *constant* can be:

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

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

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

ASR

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

LSR and LSL

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

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

ROR

Rotate right by *n* bits moves the right-hand n bits of the register into the left-hand *n* bits of the result. At the same time, all other bits are moved right by *n* bits (see Figure 4-1 on page 4-43).

 Copyright © 2002-2007 ARM Limited. All rights reserved.

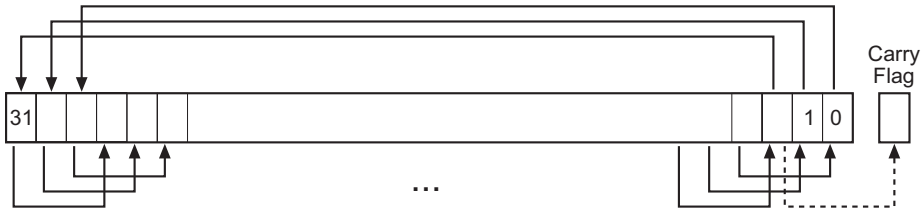

Figure 4-1 ROR

RRX

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

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

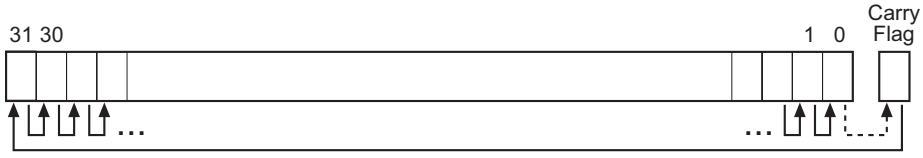

Figure 4-2 RRX

The carry flag

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

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

Instruction substitution

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

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

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

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

Copyright © 2002-2007 ARM Limited. All rights reserved.

4.4.2 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

```
op{S}{cond} {Rd}, Rn, Operand2
op{cond} {Rd}, Rn, #imm12              ; Thumb-2 only
```

where:

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 2-17).

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

Rd is the destination register.

Rn is the register holding the first operand.

Operand2 is a flexible second operand. See *Flexible second operand* on page 4-41 for details of the option.

imm12 is any value in the range 0-4095. Only permitted for ADD and SUB instructions, and only in Thumb-2 code.

Usage

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

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

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

You can use ADC, SBC, and RSC to synthesize multiword arithmetic (see *Multiword arithmetic examples* on page 4-47).

 Copyright © 2002-2007 ARM Limited. All rights reserved.

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

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

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

In certain circumstances, the assembler can substitute one instruction for another. Be aware of this when reading disassembly listings. See *Instruction substitution* on page 4-43 for details.

Use of r15 in Thumb-2 instructions

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

The exception is that you can use r15 for *Rn* in ADD and SUB instructions, with a constant *Operand2* value in the range 0-4095, and no S suffix. These instructions are useful for generating pc-relative addresses. Bit[1] of the pc value reads as 0 in this case, so that the base address for the calculation is always word-aligned.

See also *SUBS pc, LR* on page 4-48.

See also *ADR* on page 4-22.

Use of r15 in ARM instructions

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

If you use r15 as *Rd*:

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

See also *ADR* on page 4-22.

Caution

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

You cannot use r15 for *Rd* or any operand in any data processing instruction that has a register-controlled shift (see *Flexible second operand* on page 4-41).

Copyright © 2002-2007 ARM Limited. All rights reserved.

Condition flags

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

16-bit instructions

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

`ADDS 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, Rm`	ARMv6 and earlier: either *Rd* or *Rm*, or both, must be a Hi register. ARMv6T2 and above: this restriction does not apply.
`ADDS Rd, Rd, #imm`	*imm* range 0-255. *Rd* must be a Lo register.
`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.
`ADD Rd, pc, #imm`	*imm* range 0-1020, word aligned. *Rd* must be a Lo register. Bits[1:0] of the pc are read as 0 in this instruction.
`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.

Copyright © 2002-2007 ARM Limited. All rights reserved.

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

Multiword arithmetic examples

These two instructions add a 64-bit integer contained in `r2` and `r3` to another 64-bit integer contained in `r0` and `r1`, and place the result in `r4` and `r5`.

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

These instructions subtract one 96-bit integer from another:

```
SUBS    r3, r6, r9
SBCS    r4, r7, r10
SBC     r5, r8, r11
```

For clarity, the above examples use consecutive registers for multiword values. There is no requirement to do this. The following, for example, is perfectly valid:

```
SUBS    r6, r6, r9
SBCS    r9, r2, r1
SBC     r2, r8, r11
```


Copyright © 2002-2007 ARM Limited. All rights reserved.

4.4.3 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 4-44.

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 a flexible second operand. See *Flexible second operand* on page 4-41 for details.

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

Usage

You can use `SUBS pc, LR` to return from an exception if there is no return state on the stack.

`SUBS pc, LR` subtracts a value from the link register and loads the pc with the result, then copies the SPSR to the CPSR.

Notes

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

- For a return to ARM, the address written to the pc must be word-aligned.
- For a return to Thumb-2, the address written to the pc must be halfword-aligned.
- For a return to Jazelle, there are no alignment restrictions on the address written to the pc.

The results of breaking these rules are unpredictable. 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.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

Architectures

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

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

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

Copyright © 2002-2007 ARM Limited. All rights reserved.

4.4.4 AND, ORR, EOR, BIC, and ORN

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

Syntax

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

where:

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 2-17).

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

Rd is the destination register.

Rn is the register holding the first operand.

Operand2 is a flexible second operand. See *Flexible second operand* on page 4-41 for details of the options.

Usage

The AND, EOR, and ORR instructions perform bitwise AND, Exclusive OR, and OR operations on the values in *Rn* and *Operand2*.

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

The ORN Thumb-2 instruction performs an OR operation on the bits in *Rn* with the complements of the corresponding bits in the value of *Operand2*.

In certain circumstances, the assembler can substitute BIC for AND, AND for BIC, ORN for ORR, or ORR for ORN. Be aware of this when reading disassembly listings. See *Instruction substitution* on page 4-43 for details.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

Use of r15 in Thumb-2 instructions

You cannot use r15 for *Rd* or any operand in any of these instructions.

Use of r15 in ARM instructions

Note

All these uses of r15 in theseARM instructions are deprecated from ARMv7.

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

If you use r15 as *Rd*:

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

Caution

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

You cannot use r15 for any operand in any data processing instruction that has a register-controlled shift (see *Flexible second operand* on page 4-41).

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 4-41)
- do not affect the V flag.

16-bit instructions

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

`ANDS Rd, Rd, Rm` *Rd* and *Rm* must both be Lo registers.

Copyright © 2002-2007 ARM Limited. All rights reserved.

`EORS` *Rd*, *Rd*, *Rm* *Rd* and *Rm* must both be Lo registers.

`ORRS` *Rd*, *Rd*, *Rm* *Rd* and *Rm* must both be Lo registers.

`BICS` *Rd*, *Rd*, *Rm* *Rd* and *Rm* must both be Lo registers.

In the first three cases, it does not matter if you specify *OP*`S` *Rd*, *Rm*, *Rd*. The instruction is the same.

ARM/Thumb-2 examples

```
AND     r9,r2,#0xFF00
ORREQ   r2,r0,r5
EORS    r0,r0,r3,ROR r6
ANDS    r9, r8, #0x19
EORS    r7, r11, #0x18181818
BIC     r0, r1, #0xab
ORN     r7, r11, r14, ROR #4
ORNS    r7, r11, r14, ASR #32
```

Incorrect example

```
EORS    r0,r15,r3,ROR r6    ; r15 not permitted with register
                            ; controlled shift
```

 Copyright © 2002-2007 ARM Limited. All rights reserved.

4.4.5 CLZ

Count Leading Zeros.

Syntax

`CLZ{cond} Rd, Rm`

where:

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

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

Rm is the operand register. *Rm* must not be r15.

Usage

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

Condition flags

This instruction does not change the flags.

Architectures

This ARM instruction is available in ARMv5 and above.

This 32-bit Thumb instruction is available in ARMv6T2 and ARMv7.

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

Examples

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

Use the `CLZ` Thumb-2 instruction followed by a left shift of *Rm* by the resulting *Rd* value to normalize the value of register *Rm*. Use `MOVS`, rather than `MOV`, to flag the case where *Rm* is zero:

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


Copyright © 2002-2007 ARM Limited. All rights reserved.

4.4.6 CMP and CMN

Compare and Compare Negative.

Syntax

`CMP{cond} Rn, Operand2`

`CMN{cond} Rn, Operand2`

where:

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

Rn is the ARM register holding the first operand.

Operand2 is a flexible second operand. See *Flexible second operand* on page 4-41 for details of the options.

Usage

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

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

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

In certain circumstances, the assembler can substitute `CMN` for `CMP`, or `CMP` for `CMN`. Be aware of this when reading disassembly listings. See *Instruction substitution* on page 4-43 for details.

Use of r15 in ARM instructions

Note

Use of r15 in these ARM instructions is deprecated from ARMv7.

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

You cannot use r15 for any operand in any data processing instruction that has a register-controlled shift (see *Flexible second operand* on page 4-41).

 Copyright © 2002-2007 ARM Limited. All rights reserved.

Use of r15 in Thumb-2 instructions

You cannot use r15 for any operand in these instructions.

Condition flags

These instructions update the N, Z, C and V flags according to the result.

16-bit instructions

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

`CMP Rn, Rm`

`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
CMPGT   r13, r7, LSL #2
```

Incorrect example

```
CMP     r2, r15, ASR r0 ; r15 not permitted with register controlled shift
```


Copyright © 2002-2007 ARM Limited. All rights reserved.

4.4.7 MOV and MVN

Move and Move Not.

Syntax

`MOV{S}{cond} Rd, Operand2`

`MOV{cond} Rd, #imm16`

`MVN{S}{cond} Rd, Operand2`

where:

`S` is an optional suffix. If `S` is specified, the condition code flags are updated on the result of the operation (see *Conditional execution* on page 2-17).

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

`Rd` is the destination register.

`Operand2` is a flexible second operand. See *Flexible second operand* on page 4-41 for details of the options.

`imm16` is any value in the range 0-65535.

Usage

The `MOV` instruction copies the value of *`Operand2`* into *`Rd`*.

The `MVN` instruction takes the value of *`Operand2`*, performs a bitwise logical NOT operation on the value, and places the result into *`Rd`*.

In certain circumstances, the assembler can substitute `MVN` for `MOV`, or `MOV` for `MVN`. Be aware of this when reading disassembly listings. See *Instruction substitution* on page 4-43 for details.

Use of r15 in Thumb-2 MOV and MVN

You cannot use r15 for *`Rd`*, or in *`Operand2`*, in the Thumb-2 `MOV` or `MVN` instructions.

Use of r15 in ARM MOV and MVN

Note

`MOV Rd,Rm` syntax is allowed with Rd or Rn = pc, but not both. All other cases are deprecated in ARMv7 ARM.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

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

If you use r15 as *Rd*:

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

Caution

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

You cannot use r15 for *Rd* or any operand in any data processing instruction that has a register-controlled shift (see *Flexible second operand* on page 4-41).

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 4-41)
- do not affect the V flag.

16-bit instructions

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

`MOVS Rd, imm`	*Rd* must be a Lo register. *imm* range 0-255.
`MOVS Rd, Rm`	*Rd* and *Rm* must both be Lo registers.
`MOV Rd, Rm`	In ARMv5 and earlier, either *Rd* or *Rm*, or both, must be a Hi register. In ARMv6 and above, this restriction does not apply.

Copyright © 2002-2007 ARM Limited. All rights reserved.

Architectures

The *#imm16* form of the ARM instruction is available from ARMv6T2. 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 ARMv7.

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

Example

```
MVNNE   r11, #0xF000000B ; ARM only. This constant is not available in T2.
```

Incorrect example

```
MVN     r15,r3,ASR r0   ; r15 not permitted with register controlled shift
```

 Copyright © 2002-2007 ARM Limited. All rights reserved.

4.4.8 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 2-17).

`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 4-151.

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 ARMv7.

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

Copyright © 2002-2007 ARM Limited. All rights reserved.

4.4.9 TST and TEQ

Test bits and Test Equivalence.

Syntax

`TST{cond} Rn, Operand2`

`TEQ{cond} Rn, Operand2`

where:

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

Rn is the ARM register holding the first operand.

Operand2 is a flexible second operand. See *Flexible second operand* on page 4-41 for details of the options.

Usage

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

The `TST` instruction performs a bitwise AND operation on the value in *Rn* and the value of *Operand2*. This is the same as an `ANDS` instruction, except that the result is discarded.

The `TEQ` instruction performs a bitwise Exclusive OR operation on the value in *Rn* and the value of *Operand2*. This is the same as a `EORS` instruction, except that the result is discarded.

Use the `TEQ` instruction to test if two values are equal, without affecting the V or C flags (as `CMP` does).

`TEQ` is also useful for testing the sign of a value. After the comparison, the N flag is the logical Exclusive OR of the sign bits of the two operands.

Use of r15

You cannot use r15 for *Rn*, or in *Operand2*, in the Thumb-2 `TST` or `TEQ` instructions. The remainder of this section applies to the ARM instructions.

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

You cannot use r15 for any operand in any data processing instruction that has a register-controlled shift (see *Flexible second operand* on page 4-41).

 Copyright © 2002-2007 ARM Limited. All rights reserved.

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 4-41)
- do not affect the V flag.

16-bit instructions

The following form of the TST instruction is available in Thumb code, and is a 16-bit instruction when used in Thumb-2 code:

`TST` *Rn*, *Rm* *Rn* and *Rm* must both be Lo registers.

Examples

```
TST     r0, #0x3F8
TEQEQ   r10, r9
TSTNE   r1, r5, ASR r1
```

Incorrect example

```
TEQ     r15, r1, ROR r0     ; r15 not permitted with register
                            ; controlled shift
```


Copyright © 2002-2007 ARM Limited. All rights reserved.

4.4.10 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 2-17).

Rd is the destination register.

Rn is the register holding the first operand.

Rm is the register holding the second operand.

Operation

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

- if GE[0] is set, `Rd[7:0]` come from `Rn[7:0]`, otherwise from `Rm[7:0]`
- if GE[1] is set, `Rd[15:8]` come from `Rn[15:8]`, otherwise from `Rm[15:8]`
- if GE[2] is set, `Rd[23:16]` come from `Rn[23:16]`, otherwise from `Rm[23:16]`
- if GE[3] is set, `Rd[31:24]` come from `Rn[31:24]`, otherwise from `Rm[31:24]`.

Usage

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 ARMv7, except the ARMv7-M profile.

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

 Copyright © 2002-2007 ARM Limited. All rights reserved.

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
```


Copyright © 2002-2007 ARM Limited. All rights reserved.

4.4.11 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 2-17).

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.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

16-bit instructions

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

`REV` *Rd*, *Rm* — *Rd* and *Rm* must both be Lo registers.

`REV16` *Rd*, *Rm* — *Rd* and *Rm* must both be Lo registers.

`REVSH` *Rd*, *Rm* — *Rd* and *Rm* must both be Lo registers.

Architectures

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

The ARM `RBIT` instruction is available in ARMv6T2 and above.

These 32-bit Thumb instructions are available in ARMv6T2 and ARMv7.

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

Examples

```
REV     r3, r7
REV16   r0, r0
REVSH   r0, r5       ; Reverse Signed Halfword
REVHS   r3, r7       ; Reverse with Higher or Same condition
RBIT    r7, r8
```


Copyright © 2002-2007 ARM Limited. All rights reserved.

4.4.12 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 2-17).

Rd is the destination register.

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

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

sh is a constant shift. The range of values permitted depends on the instruction:

- ASR allowed shifts 1-32
- LSL allowed shifts 0-31
- LSR allowed shifts 1-32
- ROR allowed shifts 1-31.

Usage

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

LSL provides the value of a register multiplied by a power of two. LSR provides the unsigned value of a register divided by a variable power of two. Both instructions insert zeros into the vacated bit positions.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

ROR provides the value of the contents of a register rotated by a value. The bits that are rotated off the right end are inserted into the vacated bit positions on the left.

RRX provides the value of the contents of a register shifted right one bit. The old carry flag is shifted into bit[31]. If the S suffix is present, the old bit[0] is placed in the carry flag.

Condition flags

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

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

16-bit instructions

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

`ASRS Rd, Rm, #sh` *Rd* and *Rm* must both be Lo registers.

`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.

Examples

```
ASR     r7, r8, r9
LSLS    r1, r2, r3
LSR     r4, r5, r6
ROR     r4, r5, r6
```


Copyright © 2002-2007 ARM Limited. All rights reserved.

4.4.13 IT

The IT (If-Then) instruction makes up to four following Thumb instructions (the *IT block*) conditional. The conditions can be all the same, or some of them can be the logical inverse of the others.

TBD Move this instruction into a sensible location. Move much of the detail into section 2.4 (conditional execution). Note: That armasm generates IT instructions automatically, so code does not strictly need to have explicit IT instructions.

Syntax

`IT{x{y{z}}} {cond}`

where:

x	is the condition for the second instruction in the IT block.
y	is the condition for the third instruction in the IT block.
z	is the condition for the fourth instruction in the IT block.
cond	is the condition for the first instruction in the IT block.

Conditions for the second, third and fourth instruction in the IT block can be either:

T	Then. The condition applied to the instruction is *cond*.
E	Else. The condition applied to the instruction is the inverse of *cond*.

Usage

16-bit instructions in the IT block, other than CMP, CMN and TST, do not affect the condition code flags. You can use IT with the AL condition if you want this effect, without the instructions in the IT block being conditional.

The instructions in the IT block must also specify the condition in the {*cond*} part of their syntax. The assembler validates this condition against the condition in the IT instruction.

The assembler accepts IT instructions during ARM assembly, and validates them against the conditions in following instructions, but does not generate any code for them.

Restrictions

The following instructions are not permitted in an IT block:

- IT
- Conditional branches
- CBZ and CBNZ

 Copyright © 2002-2007 ARM Limited. All rights reserved.

- TBB and TBH
- CPS, CPSID and CPSIE
- SETEND.

In addition, an unconditional branch is only allowed in an IT block if it is the last instruction in the block.

Condition flags

This instruction does not change the flags.

Exceptions

Exceptions can occur between an IT instruction and the corresponding IT block, or within an IT block. This exception results in entry to the appropriate exception handler, with suitable return information in r14 and SPSR.

Instructions designed for use for exception returns can be used as normal to return from the exception, and execution of the IT block resumes correctly. This is the only way that a pc-modifying instruction can branch to an instruction in an IT block.

Architectures

This 16-bit Thumb instruction is available in ARMv6T2 and ARMv7.

Example

```
ITTE   NE        ; assemblers can allow IT to 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(CPY)
ITT    AL        ; emit 2 non-flag setting 16-bit instructions
ADDAL  r0,r0,r1  ; 16-bit ADD, not ADDS
SUBAL  r2,r2,#1  ; 16-bit SUB, not SUB
ADD    r0,r0,r1  ; expands into 32-bit ADD
IT     NE
ADD    r0,r0,r1  ; syntax error: no condition code used in IT block
```

Notes

Branches into an IT block

Except as noted under *Exceptions*, no instruction in an IT block can be the target of any branch, either from a branch instruction or from any other instruction that changes the pc.

Copyright © 2002-2007 ARM Limited. All rights reserved.

Branches out of an IT block

Instructions that modify the pc must not be used in an IT block, except as the last instruction in the block.

Branches in an IT block

Branches in IT blocks must be encoded as unconditional branches (the IT condition makes them conditional). Therefore, it is sometimes beneficial to place a conditional branch into an IT block as this increases its effective range:

```
ITT    EQ
MOVEQ  r0,r1
BEQ    dloop
```

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`, `CZB`, and `RFE`.

The assembler does not take account of directives inside IT blocks.

`BKPT` A `BKPT` instruction in an IT block is always executed, even if its condition fails.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

4.4.14 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 2-17).

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.

Copyright © 2002-2007 ARM Limited. All rights reserved.

4.5 Multiply instructions

This section contains the following subsections:

 Copyright © 2002-2007 ARM Limited. All rights reserved.

4.5.1 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 2-17).

`S` is an optional suffix. If `S` is specified, the condition code flags are updated on the result of the operation (see *Conditional execution* on page 2-17).

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*.

Copyright © 2002-2007 ARM Limited. All rights reserved.

Condition flags

If `S` is specified, the `MUL` and `MLA` instructions:

- update the N and Z flags according to the result
- corrupt the C and V flag in ARMv4 and earlier
- do not affect the C or V flag in ARMv5 and above.

Thumb instructions

The following form of the `MUL` instruction is available in Thumb code, and is a 16-bit instruction when used in Thumb-2 code:

`MULS` *`Rd`*, *`Rn`*, *`Rd`* *`Rd`* and *`Rn`* must both be Lo registers.

There are no other Thumb instructions that can set 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 ARMv7.

These 32-bit Thumb instructions are available in ARMv6T2 and ARMv7.

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
```


Copyright © 2002-2007 ARM Limited. All rights reserved.

4.5.2 UMULL, UMLAL, SMULL, and SMLAL

Signed and Unsigned Long Multiply, with optional Accumulate, with 32-bit operands, and 64-bit result and accumulator.

Syntax

Op{S}{*cond*} *RdLo*, *RdHi*, *Rn*, *Rm*

where:

Op is one of UMULL, UMLAL, SMULL, or SMLAL.

S is an optional suffix available in ARM state only. If S is specified, the condition code flags are updated on the result of the operation (see *Conditional execution* on page 2-17).

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

RdLo, *RdHi* are the destination registers. For UMLAL and SMLAL they also hold the accumulating value. *RdLo* and *RdHi* must be different registers

Rn, *Rm* are ARM registers holding the operands.

Do not use r15 for *RdHi*, *RdLo*, *Rn*, or *Rm*.

Usage

The UMULL instruction interprets the values from *Rn* and *Rm* as unsigned integers. It multiplies these integers and places the least significant 32 bits of the result in *RdLo*, and the most significant 32 bits of the result in *RdHi*.

The UMLAL instruction interprets the values from *Rn* and *Rm* as unsigned integers. It multiplies these integers, and adds the 64-bit result to the 64-bit unsigned integer contained in *RdHi* and *RdLo*.

The SMULL instruction interprets the values from *Rn* and *Rm* as two's complement signed integers. It multiplies these integers and places the least significant 32 bits of the result in *RdLo*, and the most significant 32 bits of the result in *RdHi*.

The SMLAL instruction interprets the values from *Rn* and *Rm* as two's complement signed integers. It multiplies these integers, and adds the 64-bit result to the 64-bit signed integer contained in *RdHi* and *RdLo*.

Copyright © 2002-2007 ARM Limited. All rights reserved.

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 ARMv7.

There are no 16-bit Thumb versions of these instructions.

Examples

```
UMULL       r0, r4, r5, r6
UMLALS      r4, r5, r3, r8
```


Copyright © 2002-2007 ARM Limited. All rights reserved.

4.5.3 SMUL*xy* and SMLA*xy*

Signed Multiply and Multiply Accumulate, with 16-bit operands and a 32-bit result and accumulator.

Syntax

`SMUL<x><y>{cond} {Rd}, Rn, Rm`

`SMLA<x><y>{cond} Rd, Rn, Rm, Ra`

where:

`<x>` is either `B` or `T`. `B` means use the bottom half (bits [15:0]) of `Rm`, `T` means use the top half (bits [31:16]) of `Rm`.

`<y>` is either `B` or `T`. `B` means use the bottom half (bits [15:0]) of `Rs`, `T` means use the top half (bits [31:16]) of `Rs`.

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

`Rd` is the destination register.

`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`.

`SMULxy` multiplies the 16-bit signed integers from the selected halves of `Rn` and `Rm`, and places the 32-bit result in `Rd`.

`SMLAxy` multiplies the 16-bit signed integers from the selected halves of `Rn` and `Rm`, adds the 32-bit result to the 32-bit value in `Ra`, and places the result in `Rd`.

Condition flags

These instructions do not affect the N, Z, C, or V flags.

If overflow occurs in the accumulation, `SMLAxy` sets the Q flag. To read the state of the Q flag, use an `MRS` instruction (see *MRS* on page 4-131).

Note

`SMLAxy` never clears the Q flag. To clear the Q flag, use an `MSR` instruction (see *MSR* on page 4-133).

Copyright © 2002-2007 ARM Limited. All rights reserved.

Architectures

These ARM instructions are available in ARMv6 and above, and E variants of ARMv5.

These 32-bit Thumb instructions are available in ARMv6T2 and ARMv7, 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
```


Copyright © 2002-2007 ARM Limited. All rights reserved.

4.5.4 SMULW*y* and SMLAW*y*

Signed Multiply Wide and Signed Multiply-Accumulate Wide, with one 32-bit and one 16-bit operand, providing the top 32-bits of the result.

Syntax

`SMULW<y>{cond} {Rd}, Rn, Rm`

`SMLAW<y>{cond} Rd, Rn, Rm, Ra`

where:

`<y>` is either `B` or `T`. `B` means use the bottom half (bits [15:0]) of `Rs`, `T` means use the top half (bits [31:16]) of `Rs`.

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

`Rd` is the destination register.

`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 change the flags.

Architectures

These ARM instructions are available in ARMv6 and above, and E variants of ARMv5.

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

There are no 16-bit Thumb versions of these instructions.

Copyright © 2002-2007 ARM Limited. All rights reserved.

4.5.5 SMLAL*xy*

Signed Multiply-Accumulate with 16-bit operands and a 64-bit accumulator.

Syntax

`SMLAL<x><y>{cond} RdLo, RdHi, Rn, Rm`

where:

`<x>` is either `B` or `T`. `B` means use the bottom half (bits [15:0]) of `Rm`, `T` means use the top half (bits [31:16]) of `Rm`.

`<y>` is either `B` or `T`. `B` means use the bottom half (bits [15:0]) of `Rs`, `T` means use the top half (bits [31:16]) of `Rs`.

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

`RdHi, RdLo` are the destination registers. They also hold the accumulate value. `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

`SMLALxy` multiplies the signed integer from the selected half of `Rm` by the signed integer from the selected half of `Rn`, and adds the 32-bit result to the 64-bit value in `RdHi` and `RdLo`.

Condition flags

This instruction does not change the flags.

Note

`SMLALxy` cannot raise an exception. If overflow occurs on this instruction, the result wraps round without any warning.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

Architectures

This ARM instruction is available in ARMv6 and above, and E variants of ARMv5.

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

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

Examples

```
SMLALTB     r2, r3, r7, r1
SMLALBTVS   r0, r1, r9, r2
```


Copyright © 2002-2007 ARM Limited. All rights reserved.

4.5.6 SMUAD{X} and SMUSD{X}

Dual 16-bit Signed Multiply with Addition or Subtraction of products, and optional exchange of operand halves.

Syntax

op{X}{*cond*} {*Rd*}, *Rn*, *Rm*

where:

op is one of:

- SMUAD — Dual multiply, add products.
- SMUSD — Dual multiply, subtract products.

X is an optional parameter. If X is present, the most and least significant halfwords of the second operand are exchanged before the multiplications occur.

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

Rd is the destination register.

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.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

Architectures

These ARM instructions are available in ARMv6 and above, and E variants of ARMv5.

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

There are no 16-bit Thumb versions of these instructions.

Examples

```
SMUAD       r2, r3, r2
SMUSDXNE    r0, r1, r2
```


Copyright © 2002-2007 ARM Limited. All rights reserved.

4.5.7 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

```
SMMUL{R}{cond} {Rd}, Rn, Rm
SMMLS{R}{cond} Rd, Rn, Rm, Ra
SMMLS{R}{cond} Rd, Rn, Rm, Ra
```

where:

`R` is an optional parameter. If `R` is present, the result is rounded, otherwise it is truncated.

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

`Rd` is the destination register.

`Rn, Rm` are the registers holding the operands.

`Ra` is a register holding the value to be added or subtracted from.

Do not use r15 for any of *`Rd`*, *`Rn`*, *`Rm`*, or *`Ra`*.

Operation

`SMMUL` multiplies the values from *`Rn`* and *`Rm`*, and stores the most significant 32 bits of the 64-bit result to *`Rd`*.

`SMMLA` multiplies the values from *`Rn`* and *`Rm`*, adds the value in *`Ra`* to the most significant 32 bits of the product, and stores the result in *`Rd`*.

`SMMLS` multiplies the values from *`Rn`* and *`Rm`*, subtracts the product from the value in *`Ra`* shifted left by 32 bits, and stores the most significant 32 bits of the result in *`Rd`*.

If the optional `R` parameter is specified, `0x80000000` is added before extracting the most significant 32 bits. This has the effect of rounding the result.

Condition flags

These instructions do not change the flags.

Copyright © 2002-2007 ARM Limited. All rights reserved.

Architectures

These ARM instructions are available in ARMv6 and above, and E variants of ARMv5.

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

There are no 16-bit Thumb versions of these instructions.

Examples

```
SMMULGE     r6, r4, r3
SMMULR      r2, r2, r2
```


Copyright © 2002-2007 ARM Limited. All rights reserved.

4.5.8 SMLAD and SMLSD

Dual 16-bit Signed Multiply with Addition or Subtraction of products and 32-bit accumulation.

Syntax

op{X}{*cond*} *Rd*, *Rn*, *Rm*, *Ra*

where:

op is one of:

- SMLAD — Dual multiply, accumulate sum of products.
- SMLSD — Dual multiply, accumulate difference of products.

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

X is an optional parameter. If X is present, the most and least significant halfwords of the second operand are exchanged before the multiplications occur.

Rd is the destination register.

Rn, Rm are the registers holding the operands.

Ra is the register holding the accumulate operand.

Do not use r15 for any of *Rd*, *Rn*, *Rm*, or *Ra*.

Operation

SMLAD multiplies the bottom halfword of *Rn* with the bottom halfword of *Rm*, and the top halfword of *Rn* with the top halfword of *Rm*. It then adds both products to the value in *Ra* and stores the sum to *Rd*.

SMLSD multiplies the bottom halfword of *Rn* with the bottom halfword of *Rm*, and the top halfword of *Rn* with the top halfword of *Rm*. It then subtracts the second product from the first, adds the difference to the value in *Ra*, and stores the result to *Rd*.

Condition flags

These instructions do not change the flags.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

Architectures

These ARM instructions are available in ARMv6 and above, and E variants of ARMv5.

These 32-bit Thumb instructions are available in ARMv6T2 and ARMv7, 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
```


Copyright © 2002-2007 ARM Limited. All rights reserved.

4.5.9 SMLALD and SMLSLD

Dual 16-bit Signed Multiply with Addition or Subtraction of products and 64-bit Accumulation.

Syntax

op{X}{*cond*} *RdLo*, *RdHi*, *Rn*, *Rm*

where:

op is one of:

- SMLALD — Dual multiply, accumulate sum of products.
- SMLSLD — Dual multiply, accumulate difference of products.

X is an optional parameter. If X is present, the most and least significant halfwords of the second operand are exchanged before the multiplications occur.

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

RdLo, *RdHi* are the destination registers for the 64-bit result. They also hold the 64-bit accumulate operand. *RdHi* and *RdLo* must be different registers.

Rn, *Rm* are the registers holding the operands.

Do not use r15 for any of *RdLo*, *RdHi*, *Rn*, or *Rm*.

Operation

SMLALD multiplies the bottom halfword of *Rn* with the bottom halfword of *Rm*, and the top halfword of *Rn* with the top halfword of *Rm*. It then adds both products to the value in *RdLo*, *RdHi* and stores the sum to *RdLo*, *RdHi*.

SMLSLD multiplies the bottom halfword of *Rn* with the bottom halfword of *Rm*, and the top halfword of *Rn* with the top halfword of *Rm*. It then subtracts the second product from the first, adds the difference to the value in *RdLo*, *RdHi*, and stores the result to *RdLo*, *RdHi*.

Condition flags

These instructions do not change the flags.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

Architectures

These ARM instructions are available in ARMv6 and above, and E variants of ARMv5.

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

There are no 16-bit Thumb versions of these instructions.

Examples

```
SMLALD      r10, r11, r5, r1
SMLSLD      r3, r0, r5, r1
```


Copyright © 2002-2007 ARM Limited. All rights reserved.

4.5.10 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 2-17).

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, and E variants of ARMv5.

These 32-bit Thumb instructions are available in ARMv6T2 and ARMv7, 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
```

 Copyright © 2002-2007 ARM Limited. All rights reserved.

4.5.11 MIA, MIAPH, and MIAxy

XScale coprocessor 0 instructions.

Multiply with internal accumulate (32-bit by 32-bit, 40-bit accumulate).

Multiply with internal accumulate, packed halfwords (16-bit by 16-bit twice, 40-bit accumulate).

Multiply with internal accumulate (16-bit by 16-bit, 40-bit accumulate).

Syntax

`MIA{`*cond*`} Acc, Rn, Rm`

`MIAPH{`*cond*`} Acc, Rn, Rm`

`MIA<x><y>{`*cond*`} Acc, Rn, Rm`

where:

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

Acc is the internal accumulator. The standard name is `acc`*x*, 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.

Rn, Rm are the ARM registers holding the values to be multiplied.

Do not use r15 for either *Rn* or *Rm*.

<*x*><*y*> is one of: `BB`, `BT`, `TB`, `TT`.

Usage

The `MIA` instruction multiplies the signed integers from *Rn* and *Rm*, and adds the result to the 40-bit value in *Acc*.

The `MIAPH` instruction multiplies the signed integers from the bottom halves of *Rn* and *Rm*, multiplies the signed integers from the upper halves of *Rn* and *Rm*, and adds the two 32-bit results to the 40-bit value in *Acc*.

The `MIAxy` instruction multiplies the signed integer from the selected half of *Rs* by the signed integer from the selected half of *Rm*, and adds the 32-bit result to the 40-bit value in *Acc*. <*x*> == `B` means use the bottom half (bits [15:0]) of *Rn*, <*x*> == `T` means use the top half (bits [31:16]) of *Rn*. <*y*> == `B` means use the bottom half (bits [15:0]) of *Rm*, <*y*> == `T` means use the top half (bits [31:16]) of *Rm*.

Copyright © 2002-2007 ARM Limited. All rights reserved.

Condition flags

These instructions do not change the flags.

Note

These instructions cannot raise an exception. If overflow occurs on these instructions, the result wraps round without any warning.

Architectures

These ARM instructions are only available in XScale processors.

There are no Thumb 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
```


Copyright © 2002-2007 ARM Limited. All rights reserved.

4.6 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.

4.6.1 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 4-133).

The Q flag can also be set by two other instructions (see *SMULxy and SMLAxy* on page 4-77 and *SMULWy and SMLAWy* on page 4-79), but these instructions do not saturate.

Copyright © 2002-2007 ARM Limited. All rights reserved.

4.6.2 QADD, QSUB, QDADD, and QDSUB

Signed Add, Subtract, Double and Add, Double and Subtract, saturating the result to the signed range $-2^{31} \le x \le 2^{31}-1$.

See also *Parallel add and subtract* on page 4-99.

Syntax

`op{cond} {Rd}, Rm, Rn`

where:

op is one of QADD, QSUB, QDADD, or QDSUB.

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

Rd is the destination register.

Rm, Rn are the registers holding the operands.

Do not use r15 for *Rd*, *Rm*, or *Rn*.

Usage

The QADD instruction adds the values in *Rm* and *Rn*.

The QSUB instruction subtracts the value in *Rn* from the value in *Rm*.

The QDADD instruction calculates SAT(*Rm* + SAT(*Rn* * 2)). Saturation can occur on the doubling operation, on the addition, or on both. If saturation occurs on the doubling but not on the addition, the Q flag is set but the final result is unsaturated.

The QDSUB instruction calculates SAT(*Rm* - SAT(*Rn* * 2)). Saturation can occur on the doubling operation, on the subtraction, or on both. If saturation occurs on the doubling but not on the subtraction, the Q flag is set but the final result is unsaturated.

Note

All values are treated as two's complement signed integers by these instructions.

See also *Parallel add and subtract* on page 4-99 for similar parallel instructions, available in ARMv6 and above only.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

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 4-131).

Architectures

These ARM instructions are available in ARMv6 and above, and E variants of ARMv5.

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

There are no 16-bit Thumb versions of these instructions.

Examples

```
QADD    r0, r1, r9
QDSUBLT r9, r0, r1
```


Copyright © 2002-2007 ARM Limited. All rights reserved.

4.6.3 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

op{*cond*} *Rd*, *#sat*, *Rm*{, *shift*}

where:

op is either SSAT or USAT.

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

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

sat specifies the bit position to saturate to, in the range 1 to 32 for SSAT, and 0 to 31 for USAT.

Rm is the register containing the operand. *Rm* must not be r15.

shift is an optional shift. It must be one of the following:

- ASR *#n* where *n* is in the range 1-32 (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 $-2^{sat-1} \le x \le 2^{sat-1} - 1$.

The USAT instruction applies the specified shift, then saturates to the unsigned range $0 \le x \le 2^{sat} - 1$.

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 4-131).

 Copyright © 2002-2007 ARM Limited. All rights reserved.

Architectures

These ARM instructions are available in ARMv6 and above.

These 32-bit Thumb instructions are available in ARMv6T2 and ARMv7.

There are no 16-bit Thumb versions of these instructions.

Examples

```
SSAT    r7, #16, r7, LSL #4
USATNE  r0, #7, r5
```


Copyright © 2002-2007 ARM Limited. All rights reserved.

4.7 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.

Copyright © 2002-2007 ARM Limited. All rights reserved.

4.7.1 Parallel add and subtract

Various byte-wise and halfword-wise additions and subtractions.

Syntax

`<prefix>op{cond} {Rd}, Rn, Rm`

where:

<prefix> 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.

op is one of:

ADD8	Byte-wise Addition
ADD16	Halfword-wise Addition.
SUB8	Byte-wise Subtraction.
SUB16	Halfword-wise Subtraction.
ASX	Exchange halfwords of *Rm*, then Add top halfwords and Subtract bottom halfwords.
SAX	Exchange halfwords of *Rm*, then Subtract top halfwords and Add bottom halfwords.

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

Rd is the destination register. Do not use r15 for *Rd*.

Rm, Rn are the ARM registers holding the operands. Do not use r15 for *Rm* or *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.

Copyright © 2002-2007 ARM Limited. All rights reserved.

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 $-2^{15} \le x \le 2^{15} - 1$ or $-2^7 \le x \le 2^7 - 1$. The Q flag is not affected even if these operations saturate.
- Unsigned saturating arithmetic to one of the unsigned ranges $0 \le x \le 2^{16} - 1$ or $0 \le x \le 2^8 - 1$. The Q flag is not affected even if these operations saturate.
- Signed or unsigned arithmetic, halving the results. This cannot cause overflow.

Condition flags

These instructions do not affect the N, Z, C, V, or Q flags.

The `Q`, `SH`, `UQ` and `UH` prefix variants of these instructions do not change the flags.

The `S` and `U` prefix variants of these instructions set the GE flags in the 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
  - `GE[1]` for bits[15:8] of the result
  - `GE[2]` for bits[23:16] of the result
  - `GE[3]` for bits[31:24] of the result.
- For halfword-wise operations, the GE flags are used in the same way as the C (Carry) flag for normal word-wise `SUB` and `ADD` instructions:
  - `GE[1:0]` for bits[15:0] of the result
  - `GE[3:2]` for bits[31:16] of the result.

You can use these flags to control a following `SEL` instruction, see *SEL* on page 4-62.

Note

For halfword-wise operations, GE[1:0] are set or cleared together, and GE[3:2] are set or cleared together.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

Architectures

These ARM instructions are available in ARMv6 and above.

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

There are no 16-bit Thumb versions of these instructions.

Examples

```
SHADD8      r4, r3, r9
USAXNE      r0, r0, r2
```

Incorrect examples

```
QHADD       r2, r9, r3    ; No such instruction, should be QHADD8 or QHADD16
SAX         r10, r8, r5   ; Must have a prefix.
```


Copyright © 2002-2007 ARM Limited. All rights reserved.

4.7.2 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:

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

Rd is the destination register.

Rn is the register holding the first operand.

Rm is the register holding the second operand.

Ra is the register holding the accumulate operand.

Do not use r15 for *Rd*, *Rn*, *Rn*, or *Ra*.

Operation

The USAD8 instruction finds the four differences between the unsigned values in corresponding bytes of *Rn* and *Rm*. It adds the absolute values of the four differences, and saves the result to *Rd*.

The USADA8 instruction adds the absolute values of the four differences to the value in *Ra*, and saves the result to *Rd*.

Condition flags

These instructions do not alter any flags.

Architectures

These ARM instructions are available in ARMv6 and above.

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

There are no 16-bit Thumb versions of these instructions.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

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
```


Copyright © 2002-2007 ARM Limited. All rights reserved.

4.7.3 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

op{*cond*} *Rd*, #*sat*, *Rn*

where:

op is one of:

- SSAT16 Signed saturation.
- USAT16 Unsigned saturation.

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

Rd is the destination register.

sat specifies the bit position to saturate to, and is in the range 1 to 16 for SSAT16, or 0 to 15 for USAT16.

Rn is the register holding the operand.

Do not use r15 for *Rd* or *Rn*.

Operation

Halfword-wise signed and unsigned saturation to any bit position.

The SSAT16 instruction saturates each signed halfword to the signed range $-2^{sat-1} \le x \le 2^{sat-1} - 1$.

The USAT16 instruction saturates each signed halfword to the unsigned range $0 \le x \le 2^{sat} - 1$.

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 4-131).

 Copyright © 2002-2007 ARM Limited. All rights reserved.

Architectures

These ARM instructions are available in ARMv6 and above.

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

There are no 16-bit Thumb versions of these instructions.

Examples

```
SSAT16  r7, #12, r7
USAT16  r0, #7, r5
```

Incorrect examples

```
SSAT16  r1, #16, r2, LSL #4 ; shifts not permitted with halfword saturations
```


Copyright © 2002-2007 ARM Limited. All rights reserved.

4.8 Packing and unpacking instructions

This section contains the following subsections:

 Copyright © 2002-2007 ARM Limited. All rights reserved.

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

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

where:

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

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

Rn is the source register. *Rn* must not be r15.

lsb is the least significant bit that is to be cleared or copied.

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

BFC

width bits in *Rd* are cleared, starting at *lsb*. Other bits in *Rd* are unchanged.

BFI

width bits in *Rd*, starting at *lsb*, are replaced by *width* bits from *Rn*, starting at bit[0]. Other bits in *Rd* are unchanged.

Condition flags

These instructions do not change the flags.

Architectures

These ARM instructions are available in ARMv6T2 and above.

These 32-bit Thumb instructions are available in ARMv6T2 and ARMv7.

There are no 16-bit Thumb versions of these instructions.

Copyright © 2002-2007 ARM Limited. All rights reserved.

4.8.2 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

`op{cond} Rd, Rn, #lsb, #width`

where:

op is either SBFX or UBFX.

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

Rd is the destination register.

Rn is the source register.

lsb is the bit number of least significant bit in the bitfield, in the range 0 to 31.

width is the width of the bitfield, in the range 1 to (32–*lsb*).

Do not use r15 for *Rd* or *Rn*.

Condition flags

These instructions do not alter any flags.

Architectures

These ARM instructions are available in ARMv6T2 and above.

These 32-bit Thumb instructions are available in ARMv6T2 and ARMv7.

There are no 16-bit Thumb versions of these instructions.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

4.8.3 SXT, SXTA, UXT, and UXTA

Sign extend, Sign extend with Add, Zero extend, and Zero extend with Add.

Syntax

```
SXT<extend>{cond} {Rd}, Rm{, rotation}
SXTA<extend>{cond} {Rd}, Rn, Rm{, rotation}
UXT<extend>{cond} {Rd}, Rm{, rotation}
UXTA<extend>{cond} {Rd}, Rn, Rm{, rotation}
```

where:

<extend> is one of:

- `B16` Extends two 8-bit values to two 16-bit values.
- `B` Extends an 8-bit value to a 32-bit value.
- `H` Extends a 16-bit value to a 32-bit value.

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

Rd is the destination register.

Rn is the register holding the number to add (`SXTA` and `UXTA` only).

Rm is the register holding the value to extend.

rotation is one of:

- `ROR #8` Value from *Rm* is rotated right 8 bits.
- `ROR #16` Value from *Rm* is rotated right 16 bits.
- `ROR #24` Value from *Rm* is rotated right 24 bits.

If *rotation* is omitted, no rotation is performed.

You must not use r15 for *Rd*, *Rn*, or *Rm*.

Copyright © 2002-2007 ARM Limited. All rights reserved.

Operation

These instructions do the following:

1. Rotate the value from *Rm* right by 0, 8, 16 or 24 bits (ARM and Thumb-2 only).
2. Do one of the following to the value obtained:
   - Extract bits[7:0], sign or zero extend to 32 bits. If the instruction is extend and add, add the value from *Rn*.
   - Extract bits[15:0], sign or zero extend to 32 bits. If the instruction is extend and add, add the value from *Rn*.
   - Extract bits[23:16] and bits[7:0] and sign or zero extend them to 16 bits. If the instruction is extend and add, add them to bits[31:16] and bits[15:0] respectively of *Rn* to form bits[31:16] and bits[15:0] of the result.

Condition flags

These instructions do not change the flags.

16-bit instructions

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

`SXTB` *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.

`SXT` and `UXT` Thumb instructions are available in ARMv6T2 and ARMv7.

`SXTA` and `UXTA` Thumb instructions are available in ARMv6T2 and ARMv7, except the ARMv7-M profile.

These 16-bit Thumb instructions are available in ARMv6 and above.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

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.
```


Copyright © 2002-2007 ARM Limited. All rights reserved.

4.8.4 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 2-17).

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 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.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

Architectures

These ARM instructions are available in ARMv6 and above.

These 32-bit Thumb instructions are available in ARMv6T2 and ARMv7, 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
```


Copyright © 2002-2007 ARM Limited. All rights reserved.

4.9 Branch instructions

This section contains the following subsections:

 Copyright © 2002-2007 ARM Limited. All rights reserved.

4.9.1 B, BL, BX, BLX, and BXJ

Branch, Branch with Link, Branch and exchange instruction set, Branch with Link and exchange instruction set, Branch and change to Jazelle state.

Syntax

op{*cond*}{.W} *label*

op{*cond*} *Rm*

where:

op is one of:

B	Branch.
BL	Branch with link.
BX	Branch and exchange instruction set.
BLX	Branch with link, and exchange instruction set.
BXJ	Branch, and change to Jazelle execution.

cond is an optional condition code (see *Conditional execution* on page 2-17). *cond* is not available on all forms of this instruction, see *Instruction availability and branch ranges* on page 4-116.

.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.

label is a program-relative expression. See *Register-relative and program-relative expressions* on page 3-33 for more information.

Rm is a register containing an address to branch to.

Operation

All these instructions cause a branch to *label*, or to the address contained in *Rm*. In addition:

- The BL and BLX instructions copy the address of the next instruction into r14 (lr, the link register).
- The BX and BLX instructions can change the processor state from ARM to Thumb, or from Thumb to ARM.

  BLX *label* always changes the state.

  BX *Rm* and BLX *Rm* derive the target state from bit[0] of *Rm*:

  — if bit[0] of *Rm* is 0, the processor changes to, or remains in, ARM state

 Copyright © 2002-2007 ARM Limited. All rights reserved.

— 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 4-7 Branch instruction availability and range

Instruction	ARM		16-bit Thumb		32-bit Thumb
B label	±32MB	(All)	±2KB	(All T)	±16MB[a]
B{cond} label	±32MB	(All)	–252 to +258	(All T)	±1MB[a]
B Rm	Use BX Rm		Use BX Rm		Use 16-bit BX Rm
B{cond} Rm	Use BX{cond} Rm		-		-
BL label	±32MB	(All)	±4MB[b]	(All T)	±16MB
BL{cond} label	±32MB	(All)	-		-
BL Rm	Use BLX Rm		Use BLX Rm		Use 16-bit BLX Rm
BL{cond} Rm	Use BLX{cond} Rm		-		-
BX Rm	Available	(4T, 5)	Available	(All T)	Use 16-bit
BX{cond} Rm	Available	(4T, 5)	-		-
BLX label	±32MB	(5)	±4MB[c]	(5T)	±16MB [d]
BLX Rm	Available	(5)	Available	(5T)	Use 16-bit
BLX{cond} Rm	Available	(5)	-		-
BXJ Rm	Available	(5J, 6)	-		Available [d]
BXJ{cond} Rm	Available	(5J, 6)	-		-

a. Use .W to instruct the assembler to use this 32-bit instruction.
b. This is an instruction pair.
c. This is an instruction pair.
d. Not available in ARMv7-M.

Copyright © 2002-2007 ARM Limited. All rights reserved.

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 *RealView Compilation Tools Linker and Utilities Guide*). The added code is called a *veneer*.

B in Thumb-2

You can use the .W width specifier to force B to generate a 32-bit instruction in Thumb-2 code.

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

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

```
B       loopA
BLE     ng+8
BL      subC
BLLT    rtX
BEQ     {pc}+4  ; #0x8004
```


Copyright © 2002-2007 ARM Limited. All rights reserved.

4.9.2 CBZ and CBNZ

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

Syntax

`CBZ Rn, label`

`CBNZ Rn, label`

where:

Rn is the register holding the operand.

label is the branch destination.

Usage

You can use the `CBZ` or `CBNZ` instructions to save one instruction in compare with zero and branch code sequences.

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

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

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

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

Restrictions

The branch destination must be within 4 to 130 bytes after the instruction.

These instructions must not be used inside an IT block.

Condition flags

These instructions do not change the flags.

Architectures

These 16-bit Thumb instructions are available in ARMv6T2 and ARMv7.

There are no ARM or 32-bit Thumb versions of these instructions.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

4.9.3 TBB and TBH

Table Branch Byte and Table Branch Halfword.

Syntax

`TBB [Rn, Rm]`

`TBH [Rn, Rm, LSL #1]`

where:

Rn is the base register. This contains the address of the table of branch lengths.

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.

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 ARMv7.

There are no ARM, or 16-bit Thumb, versions of these instructions.

Copyright © 2002-2007 ARM Limited. All rights reserved.

4.10 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 4-128).

It contains the following sections:

- *CDP and CDP2* on page 4-121

  Coprocessor Data oPerations.

- *MCR, MCR2, MCRR, and MCRR2* on page 4-122

  Move to Coprocessor from ARM Register or Registers, possibly with coprocessor operations.

- *MRC, MRC2, MRRC and MRRC2* on page 4-124

  Move to ARM Register or Registers from Coprocessor, possibly with coprocessor operations.

- *LDC, LDC2, STC, and STC2* on page 4-126

  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.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

4.10.1 CDP and CDP2

Coprocessor data operations.

Syntax

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

where:

op	is either CDP or CDP2.
cond	is an optional condition code (see *Conditional execution* on page 2-17). In ARM code, *cond* is not allowed for CDP2.
coproc	is the name of the coprocessor the instruction is for. The standard name is p*n*, where *n* is an integer in the range 0 to 15.
opcode1	is a coprocessor-specific opcode.
CRd, *CRn*, *CRm*	are coprocessor registers.
opcode2	is an optional coprocessor-specific opcode.

Usage

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

Architectures

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

The CDP2 ARM instruction is available in ARMv5 and above.

These 32-bit Thumb instructions are available in ARMv6T2 and ARMv7.

There are no 16-bit Thumb versions of these instructions.

Copyright © 2002-2007 ARM Limited. All rights reserved.

4.10.2 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

`op{cond} coproc, #opcode1, Rt, CRn, CRm{, #opcode2}`

`op{cond} coproc, #opcode1, Rt, Rt2, CRm`

where:

op is one of MCR, MCR2, MCRR, or MCRR2.

cond is an optional condition code (see *Conditional execution* on page 2-17). In ARM code, *cond* is not allowed for MCR2 or MCRR2.

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

opcode1 is a coprocessor-specific opcode.

Rt, *Rt2* are ARM source registers. *Rt2* is only available in MCRR and MCRR2. Do not use r15 for *Rt* or *Rt2* in MCRR or MCRR2.

CRn, *CRm* are coprocessor registers. *CRn* is only available in MCR and MCR2.

opcode2 is an optional coprocessor-specific opcode available in MCR and MCR2.

Usage

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

 Copyright © 2002-2007 ARM Limited. All rights reserved.

Architectures

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

The MCR2 ARM instruction is available in ARMv5 and above.

The MCRR ARM instruction is available in ARMv6 and above, and E variants of ARMv5.

The MCRR2 ARM instruction is available in ARMv6 and above.

These 32-bit Thumb instructions are available in ARMv6T2 and ARMv7.

There are no 16-bit Thumb versions of these instructions.

Copyright © 2002-2007 ARM Limited. All rights reserved.

4.10.3 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

`op{cond} coproc, #opcode1, Rt, CRn, CRm{, #opcode2}`

`op{cond} coproc, #opcode1, Rt, Rt2, CRm`

where:

op is MRC, MRC2, MRRC, or MRRC2.

cond is an optional condition code (see *Conditional execution* on page 2-17). In ARM code, *cond* is not allowed for MRC2 or MRRC2.

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

opcode1 is a coprocessor-specific opcode.

Rt, *Rt2* are ARM source registers. Do not use r15. *Rt2* is only available in MRRC and MRRC2.

In MRC and MRC2, *Rt* can be APSR_nzcv.

CRn, *CRm* are coprocessor registers. *CRn* is only available in MRC and MRC2.

opcode2 is an optional coprocessor-specific opcode available in MRC and MRC2.

Usage

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

 Copyright © 2002-2007 ARM Limited. All rights reserved.

Architectures

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

The MRC2 ARM instruction is available in ARMv5 and above.

The MRRC ARM instruction is available in ARMv6 and above, and E variants of ARMv5.

The MRRC2 ARM instruction is available in ARMv6 and above.

These 32-bit Thumb instructions are available in ARMv6T2 and ARMv7.

There are no 16-bit Thumb versions of these instructions.

Copyright © 2002-2007 ARM Limited. All rights reserved.

4.10.4 LDC, LDC2, STC, and STC2

Transfer Data between memory and Coprocessor.

Syntax

`op{L}{cond} coproc, CRd, [Rn]`

`op{L}{cond} coproc, CRd, [Rn, #{-}offset]{!}`

`op{L}{cond} coproc, CRd, [Rn], #{-}offset`

`op{L}{cond} coproc, CRd, label`

where:

op is one of LDC, LDC2, STC, or STC2.

cond is an optional condition code (see *Conditional execution* on page 2-17).
In ARM code, *cond* is not allowed for LDC2 or STC2.

L is an optional suffix specifying a long transfer.

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

CRd is the coprocessor register to load or store.

Rn is the register on which the memory address is based. If r15 is specified, the value used is the address of the current instruction plus eight.

- is an optional minus sign. If - is present, the offset is subtracted from *Rn*. Otherwise, the offset is added to *Rn*.

offset is an expression evaluating to a multiple of 4, in the range 0 to 1020.

! is an optional suffix. If ! is present, the address including the offset is written back into *Rn*.

label is a word-aligned program-relative expression. See *Register-relative and program-relative expressions* on page 3-33 for more information.
label must be within 1020 bytes of the current instruction.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

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 ARMv5 and above.

These 32-bit Thumb instructions are available in ARMv6T2 and ARMv7.

There are no 16-bit Thumb versions of these instructions.

Notes

Use of pc-relative addressing in the STC and STC2 instructions is deprecated in ARMv6T2 and above.

Copyright © 2002-2007 ARM Limited. All rights reserved.

4.11 Miscellaneous instructions

This section contains the following subsections:

 Copyright © 2002-2007 ARM Limited. All rights reserved.

4.11.1 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 ARMv5 and above.

This 16-bit Thumb instruction is available in T variants of ARMv5 and above.

There is no 32-bit Thumb version of this instruction.

Copyright © 2002-2007 ARM Limited. All rights reserved.

4.11.2 SVC

SuperVisor Call.

Syntax

`SVC{cond} #immed`

where:

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

immed is an expression evaluating to an integer in the range:

- 0 to 2^{24}–1 (a 24-bit value) in an ARM instruction
- 0-255 (an 8-bit value) in a 16-bit Thumb instruction.

Usage

The `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 RealView Compilation Tools Developer Guide).

immed is ignored by the processor. However, it can be retrieved by the exception handler to determine what service is being requested.

Note

As part of the development of the ARM assembly language, the `SWI` instruction has been renamed to `SVC`. 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 and T2 variants of ARMv6 and above.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

4.11.3 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 2-17).

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.
- `SPSR` on any processor except ARMv7-M, in privileged modes only.
- *Mpsr* on ARMv7-M processors only.

Mpsr can be any of: `IPSR`, `EPSR`, `IEPSR`, `IAPSR`, `EAPSR`, `PSR`, `MSP`, `PSP`, `DSP`, `PRIMASK`, `BASEPRI`, `BASEPRI_MAX`, 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.

Copyright © 2002-2007 ARM Limited. All rights reserved.

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 ARMv7.

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

 Copyright © 2002-2007 ARM Limited. All rights reserved.

4.11.4 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)

`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 2-17).

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).

Copyright © 2002-2007 ARM Limited. All rights reserved.

Syntax (ARMv7-M)

`MSR{cond} psr, Rm`

where:

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

Rm is the source register.

psr can be any of: `APSR`, `IPSR`, `EPSR`, `IEPSR`, `IAPSR`, `EAPSR`, `PSR`, `MSP`, `PSP`, `DSP`, `PRIMASK`, `BASEPRI`, `BASEPRI_MAX`, or `CONTROL`. ARMv7-M only.

Usage

See *MRS* on page 4-131.

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.

These 32-bit Thumb instructions are available in ARMv6T2 and ARMv7.

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

 Copyright © 2002-2007 ARM Limited. All rights reserved.

4.11.5 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

CPS*effect iflags*{, *#mode*}

CPS *#mode*

where:

effect is one of:

- IE — Interrupt or abort enable.
- ID — Interrupt or abort disable.

iflags is a sequence of one or more of:

- a — Enables or disables imprecise aborts.
- i — Enables or disables IRQ interrupts.
- f — Enables or disables FIQ interrupts.

mode specifies the number of the mode to change to.

Condition flags

This instruction does not change the condition flags.

16-bit instructions

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

CPSIE *iflags* — You cannot specify a mode change in a 16-bit Thumb instruction.

CPSID *iflags* — You cannot specify a mode change in a 16-bit Thumb instruction.

Copyright © 2002-2007 ARM Limited. All rights reserved.

Architectures

This ARM instruction is available in ARMv6 and above.

These 32-bit Thumb instructions are available in ARMv6T2 and ARMv7.

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
```

 Copyright © 2002-2007 ARM Limited. All rights reserved.

4.11.6 SMC

Secure Monitor Call.

For details see the *ARM Architecture Reference Manual Security Extensions Supplement*.

Syntax

`SMC{cond} #immed_16`

where:

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

immed_16 is a 16-bit immediate value. This is ignored by the ARM processor, but can be used by the 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 ARMv7, if they have the Security Extensions.

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

Copyright © 2002-2007 ARM Limited. All rights reserved.

4.11.7 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 ARMv7-M profile.

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
```

 Copyright © 2002-2007 ARM Limited. All rights reserved.

4.11.8 NOP, SEV, WFE, WFI, and YIELD

No Operation, Set Event, Wait For Event, Wait for Interrupt, and Yield.

Syntax

`NOP{`*cond*`}`

`SEV{`*cond*`}`

`WFE{`*cond*`}`

`WFI{`*cond*`}`

`YIELD{`*cond*`}`

where:

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

Usage

These are hint instructions. It is optional whether they are implemented or not. If any one of them is not implemented, it behaves as a `NOP`.

NOP

`NOP` does nothing. If `NOP` is not implemented as a specific instruction on your target architecture, the assembler generates an alternative instruction that does nothing, such as `MOV r0, r0` (ARM) or `MOV r8, r8` (Thumb).

`NOP` is not necessarily a time-consuming `NOP`. The processor might remove it from the pipeline before it reaches the execution stage.

You can use `NOP` for padding, for example to place the following instruction on a 64-bit boundary.

SEV

`SEV` causes an event to be signaled to all cores within a multiprocessor system. If `SEV` is implemented, `WFE` must also be implemented.

WFE

If the Event Register is not set, `WFE` suspends execution until one of the following events occurs:

- an IRQ interrupt, unless masked by the CPSR I-bit
- an FIQ interrupt, unless masked by the CPSR F-bit

Copyright © 2002-2007 ARM Limited. All rights reserved.

- an Imprecise Data abort, unless masked by the CPSR A-bit
- a Debug Entry request, if Debug is enabled
- an Event signaled by another processor using the SEV instruction.

If the Event Register is set, WFE clears it and returns immediately.

If WFE is implemented, SEV must also be implemented.

WFI

WFI suspends execution until one of the following events occurs:

- an IRQ interrupt, regardless of the CPSR I-bit
- an FIQ interrupt, regardless of the CPSR F-bit
- an Imprecise Data abort, unless masked by the CPSR A-bit
- a Debug Entry request, regardless of whether Debug is enabled.

YIELD

YIELD indicates to the hardware that the current thread is performing a task, for example a spinlock, that can be swapped out. Hardware can use this hint to suspend and resume threads in a multithreading system.

Architectures

These ARM instructions are available in ARMv6T2 and ARMv7.

These 32-bit Thumb instructions are available in ARMv6T2 and ARMv7.

These 16-bit Thumb instructions are available in ARMv6T2 and above, and K variants of ARMv6.

NOP is available on all other ARM and Thumb architectures as a pseudo-instruction.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

4.11.9 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 2-17).

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.

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.

Copyright © 2002-2007 ARM Limited. All rights reserved.

- All Cache, Branch predictor and TLB maintenance operations before this instruction complete.

Allowed values are:

`SY`	Full system DSB operation. This is the default, and can be omitted.
`UN`	DSB operation only out to the point of unification.
`ST`	DSB operation that waits only for stores to complete.
`UNST`	DSB operation that waits only for stores to complete and only out to the point of unification.

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 DMB operation. This is the default, and can be omitted.

Architectures

These ARM and 32-bit Thumb instructions are available in ARMv7.

There are no 16-bit Thumb versions of these instructions.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

4.11.10 MAR and MRA

XScale coprocessor 0 instructions.

Transfer between two general-purpose registers and a 40-bit internal accumulator.

Syntax

`MAR{cond} Acc, RdLo, RdHi`

`MRA{cond} RdLo, RdHi, Acc`

where:

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

Acc is the internal accumulator. The standard name is `acc`*x*, 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 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
MRAGT   r4, r8, acc0
```


Copyright © 2002-2007 ARM Limited. All rights reserved.

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 4-145

  Switch between Thumb state and ThumbEE state.

- *CHKA* on page 4-146

  Check array.

- *HB, HBL, HBLP, and HBP* on page 4-147

  Handler Branch, branches to a specified handler.

Copyright © 2002-2007 ARM Limited. All rights reserved.

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*.

Copyright © 2002-2007 ARM Limited. All rights reserved.

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. Can be any of r0-r12, SP, or LR.

Rm contains the array index. Can be any of r0-r12, or LR.

Architectures

This instruction is not available in ARM state.

This 16-bit ThumbEE instruction is only available in ARMv7, with Thumb-2EE support.

Copyright © 2002-2007 ARM Limited. All rights reserved.

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.

Copyright © 2002-2007 ARM Limited. All rights reserved.

4.13 Pseudo-instructions

The ARM assembler supports a number of pseudo-instructions that are translated into the appropriate combination of ARM, Thumb, or Thumb-2 instructions at assembly time.

The pseudo-instructions are described in the following sections:

- *ADRL pseudo-instruction* on page 4-149

  Load a program-relative or register-relative address into a register (medium range, position independent)

- *MOV32 pseudo-instruction* on page 4-151

  Load a register with a 32-bit constant value or an address (unlimited range, but not position independent). Available for ARMv6T2 and above only.

- *LDR pseudo-instruction* on page 4-153

  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 4-156

  Generate an architecturally undefined instruction. Available for all ARM architectures.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

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 processors before ARMv6T2.

Syntax

ADRL{*cond*} *Rd*, *label*

where:

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

Rd is the register to load.

label is a program-relative or register-relative expression. See *Register-relative and program-relative expressions* on page 3-33 for more information.

Usage

ADRL always assembles to two 32-bit instructions. Even if the address can be reached in a single instruction, a second, redundant instruction is produced.

If the assembler cannot construct the address in two instructions, it generates an error message and the assembly fails. See *LDR pseudo-instruction* on page 4-153 for information on loading a wider range of addresses (see also *Loading constants into registers* on page 2-25).

ADRL produces position-independent code, because the address is program-relative or register-relative.

If *label* is program-relative, it must evaluate to an address in the same assembler area as the ADRL pseudo-instruction, see *AREA* on page 7-66.

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.

Copyright © 2002-2007 ARM Limited. All rights reserved.

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. In ARM and 32-bit Thumb, more distant addresses can be in range if the alignment is 16-bytes or more relative to this point.

Copyright © 2002-2007 ARM Limited. All rights reserved.

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 address space.

If `MOV32` is used to load an address, the resulting code is not position-independent.

Syntax

`MOV32{`*`cond`*`} `*`Rd`*`, `*`expr`*

where:

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

`Rd` is the register to be loaded. *`Rd`* must not be sp or pc.

`expr` can be any one of the following:

`symbol`	A label in this or another program area.
`constant`	Any 32-bit constant.
`symbol + constant`	A label plus a 32-bit constant.

Copyright © 2002-2007 ARM Limited. All rights reserved.

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 ARMv7 in both ARM and Thumb.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

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 4-11 for information on the LDR *instruction*.

Also, see *Loading with LDR Rd, =const* on page 2-30, for information on loading constants with the LDR pseudo-instruction.

Syntax

```
LDR{cond}{.w} Rt,=[expr | label-expr]
```

where:

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

.W is an optional instruction width specifier.

Rt is the register to be loaded.

expr evaluates to a numeric constant:

- The assembler generates a MOV or MVN instruction, if the value of *expr* is within range.
- If the value of *expr* is *not* within range of a MOV or MVN instruction, the assembler places the constant in a literal pool and generates a program-relative LDR instruction that reads the constant from the literal pool.

label-expr is a program-relative or external expression. The assembler places the value of *label-expr* in a literal pool and generates a program-relative LDR instruction that loads the value from the literal pool.

If *label-expr* is an external expression, or is not contained in the current section, the assembler places a linker relocation directive in the object file. The linker generates the address at link time.

Copyright © 2002-2007 ARM Limited. All rights reserved.

If *label-expr* is a local label (see *Local labels* on page 3-27), 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 RVCT2.2, 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 (Thumb, 16-bit Thumb-2). You are responsible for ensuring that there is a literal pool within range. See *LTORG* on page 7-18 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.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

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 4-151 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
```


Copyright © 2002-2007 ARM Limited. All rights reserved.

4.13.4 UND pseudo-instruction

Generate an architecturally undefined instruction. An attempt to execute an undefined instuction 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 2-17). *cond* is not allowed on this pseudo-instruction in ARM code, or in 16-bit Thumb code.

`.W` is an optional instruction width specifier.

expr evaluates to a numeric constant in the range:

- 0-65535 in ARM code
- 0-4095 in 32-bit Thumb code
- 0-255 in 16-bit Thumb code.

If *expr* is omitted, the value 0 is used.

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`.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

Chapter 5
NEON and VFP Programming

This chapter provides reference information about programming NEON™ and the VFP coprocessor in assembly language. It contains the following sections:

Copyright © 2002-2007 ARM Limited. All rights reserved.

See Table 5-1, Table 5-2 on page 5-5, and Table 5-3 on page 5-6 to locate descriptions of individual instructions.

Table 5-1 Location of NEON instructions

Mnemonic	Brief description	Page
VABA, VABD	Absolute difference, Absolute difference and Accumulate	page 5-51
VABS	Absolute value	page 5-52
VACGE, VACGT	Absolute Compare Greater than or Equal, Greater Than	page 5-30
VACLE, VACLT	Absolute Compare Less than or Equal, Less than (pseudo-instructionS)	page 5-80
VADD	Add	page 5-53
VADDHN	Add, select High half	page 5-54
VAND	Bitwise AND	page 5-26
VAND	Bitwise AND (pseudo-instruction)	page 5-79
VBIC	Bitwise Bit Clear (register)	page 5-26
VBIC	Bitwise Bit Clear (immediate)	page 5-27
VBIF, VBIT, VBSL	Bitwise Insert if False, Insert if True, Select	page 5-28
VCEQ, VCLE, VCLT	Compare Equal, Less than or Equal, Compare Less Than	page 5-31
VCLE, VCLT	Compare Less than or Equal, Compare Less Than (pseudo-instruction)	page 5-81
VCLS, VCLZ, VCNT	Count Leading Sign bits, Count Leading Zeros, and Count set bits	page 5-59
VCVT	Convert fixed-point or integer to floating point, floating-point to integer or fixed-point	page 5-34
VDUP	Duplicate scalar to all lanes of vector	page 5-35
VEXT	Extract	page 5-36
VCGE, VCGT	Compare Greater than or Equal, Greater Than	page 5-31
VEOR	Bitwise Exclusive OR	page 5-26
VHADD, VHSUB	Halving Add, Halving Subtract	page 5-55
VMAX, VMIN	Maximum, Minimum	page 5-58
VLD	Vector Load	page 5-68
VMLA, VMLS	Multiply Accumulate, Multiply Subtract (vector)	page 5-64

Copyright © 2002-2007 ARM Limited. All rights reserved.

Table 5-1 Location of NEON instructions (continued)

Mnemonic	Brief description	Page
VMLA, VMLS	Multiply Accumulate, Multiply Subtract (by scalar)	page 5-65
VMOV	Move (immediate)	page 5-37
VMOV	Move (register)	page 5-29
VMOVL, VMOV{U}N	Move Long, Move Narrow (register)	page 5-38
VMUL	Multiply (vector)	page 5-64
VMUL	Multiply (by scalar)	page 5-65
VMVN	Move Negative (immediate)	page 5-37
VNEG	Negate	page 5-52
VORN	Bitwise OR NOT	page 5-26
VORN	Bitwise OR NOT (pseudo-instruction)	page 5-79
VORR	Bitwise OR (register)	page 5-26
VORR	Bitwise OR (immediate)	page 5-27
VPADD, VPADAL	Pairwise Add, Pairwise Add and Accumulate	page 5-56
VPMAX, VPMIN	Pairwise Maximum, Pairwise Minimum	page 5-58
VQABS	Absolute value, saturate	page 5-52
VQADD	Add, saturate	page 5-53
VQDMLAL, VQDMLSL	Saturating Doubling Multiply Accumulate, and Multiply Subtract	page 5-66
VQMOV{U}N	Saturating Move (register)	page 5-38
VQDMUL	Saturating Doubling Multiply	page 5-66
VQDMULH	Saturating Doubling Multiply returning High half	page 5-67
VQNEG	Negate, saturate	page 5-52
VQRDMULH	Saturating Doubling Multiply returning High half	page 5-67
VQRSHL	Shift Left, Round, saturate (by signed variable)	page 5-46
VQRSHR	Shift Right, Round, saturate (by immediate)	page 5-48
VQSHL	Shift Left, saturate (by immediate)	page 5-45

Copyright © 2002-2007 ARM Limited. All rights reserved.

Table 5-1 Location of NEON instructions (continued)

Mnemonic	Brief description	Page
VQSHL	Shift Left, saturate (by signed variable)	page 5-46
VQSHR	Shift Right, saturate (by immediate)	page 5-48
VQSUB	Subtract, saturate	page 5-53
VRADDH	Add, select High half, Round	page 5-54
VRECPE, VRECPS	Reciprocal Estimate, Reciprocal Step	page 5-60
VREV	Reverse elements	page 5-39
VRHADD	Halving Add, Round	page 5-55
VRSHR, VRSRA	Shift Right and Round, Shift Right, Round, and Accumulate (by immediate)	page 5-47
VRSUBH	Subtract, select High half, Round	page 5-54
VRSQRTE, VRSQRTS	Reciprocal Square Root Estimate, Reciprocal Square Root Step	page 5-61
VSHL	Shift Left (by immediate)	page 5-45
VSHR	Shift Right (by immediate)	page 5-47
VSLI	Shift Left and Insert	page 5-49
VSRA	Shift Right, Accumulate (by immediate)	page 5-47
VSRI	Shift Right and Insert	page 5-49
VST	Vector Store	page 5-68
VSUB	Subtract	page 5-53
VSUBH	Subtract, select High half	page 5-54
VSWP	Swap vectors	page 5-40
VTBL, VTBX	Vector table look-up	page 5-41
VTST	Test bits	page 5-32
VTRN	Vector transpose	page 5-42
VUZP, VZIP	Vector interleave and de-interleave	page 5-43

Copyright © 2002-2007 ARM Limited. All rights reserved.

Table 5-2 Location of shared NEON and VFP instructions

Mnemonic	Brief description	Page	Op.	Arch.
VLDM	Load multiple	page 5-20	-	All
VLDR	Load (see also *VLDR pseudo-instruction* on page 5-77)	page 5-19	Scalar	All
VMOV	Transfer from one ARM® register to half of double-precision	page 5-22	Scalar	All
	Transfer from two ARM registers to double-precision	page 5-21	Scalar	VFPv2
	Transfer from half of double-precision to ARM register	page 5-22	Scalar	All
	Transfer from double-precision to two ARM registers	page 5-21	Scalar	VFPv2
	Transfer from single-precision to ARM register	page 5-23	Scalar	All
	Transfer from ARM register to single-precision	page 5-23	Scalar	All
VMRS	Transfer from NEON and VFP system register to ARM register	page 5-24	-	All
VMSR	Transfer from ARM register to NEON and VFP system register	page 5-24	-	All
VSTM	Store multiple	page 5-20	-	All
VSTR	Store	page 5-19	Scalar	All

Copyright © 2002-2007 ARM Limited. All rights reserved.

Table 5-3 Location of VFP instructions

Mnemonic	Brief description	Page	Op.	Arch.
VABS	Absolute value	page 5-89	Vector	All
VADD	Add	page 5-90	Vector	All
VCMP	Compare	page 5-92	Scalar	All
VCVT	Convert betwen single-precision and double-precision	page 5-93	Scalar	All
	Convert between floating-point and integer	page 5-94	Scalar	All
	Convert between floating-point and fixed-point	page 5-95	Scalar	VFPv3
VDIV	Divide	page 5-90	Vector	All
VMLA	Multiply accumulate	page 5-91	Vector	All
VMLS	Multiply subtract	page 5-91	Vector	All
VMOV	Insert floating-point constant in single-precision or double-precision register (see also Table 5-2 on page 5-5)	page 5-96	Scalar	VFPv3
VMUL	Multiply	page 5-91	Vector	All
VNEG	Negate	page 5-89	Vector	All
VNMLA	Negated multiply accumulate	page 5-91	Vector	All
VNMLS	Negated multiply subtract	page 5-91	Vector	All
VNMUL	Negated multiply	page 5-91	Vector	All
VSQRT	Square Root	page 5-89	Vector	All
VSUB	Subtract	page 5-90	Vector	All

Copyright © 2002-2007 ARM Limited. All rights reserved.

5.1 The extension register bank

NEON and VFPv3 use the same extension register bank. This is distinct from the ARM register bank. It is a superset of the VFPv2 register bank.

The extension register bank can be referenced using three explicitly aliased views, as described in the following sections.

Figure 5-1 on page 5-8 shows the three views of the extension register bank, and the way the word, doubleword, and quadword registers overlap.

5.1.1 NEON views of the register bank

NEON views the extension register bank as:

- Sixteen 128-bit quadword registers, `Q0-Q15`.
- Thirty-two 64-bit doubleword registers, `D0-D31`. This view is also available in VFPv3.
- 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*.

5.1.2 VFPv3 views of the extension register bank

In VFPv3, you can view the extension register bank as:

- Thirty-two 64-bit doubleword registers, `D0-D31`. This view is also available in NEON.
- 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.

Copyright © 2002-2007 ARM Limited. All rights reserved.

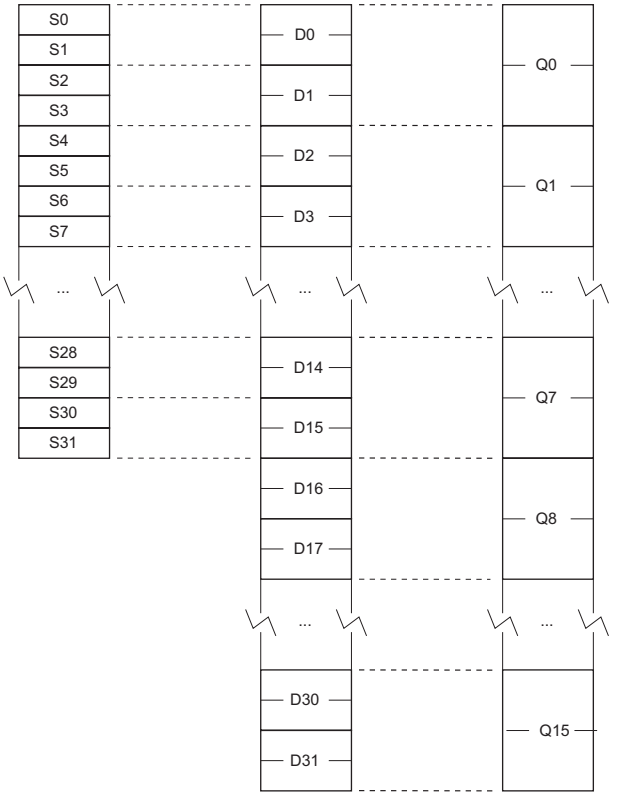

Figure 5-1 Extension register bank

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 `Q6` by referring to `D12`, and the most significant half of the elements by referring to `D13`.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

5.2 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 4-68 for details.

The only VFP instruction that can be used to update the status flags is `FCMP`. 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 5-82).

Note

To use these flags to control conditional instructions, including conditional VFP instructions, you must first copy them into the APSR using a `VMSR` instruction (see *VMRS and VMSR* on page 5-24).

Following an `FCMP` 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 5-10.

Copyright © 2002-2007 ARM Limited. All rights reserved.

Table 5-4 Condition codes

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

Note

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

Copyright © 2002-2007 ARM Limited. All rights reserved.

5.3 General information

Certain information common to many instructions is presented here, to avoid duplication. This section contains the following subsections:

- *Floating-point exceptions*
- *Architecture versions*
- *NEON and VFP data types* on page 5-12
- *Normal, Long, Wide, Narrow, and saturating instructions in NEON* on page 5-13
- *NEON Scalars* on page 5-15
- *Extended notation* on page 5-15
- *Polynomial arithmetic over {0,1}* on page 5-16
- *The VFP coprocessor* on page 5-17
- *VFP registers* on page 5-17.

5.3.1 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 exceptions.

5.3.2 Architecture versions

All NEON instructions are available on all systems supporting NEON. Except for the instructions that are shared between NEON and VFP, NEON instructions are only available on systems supporting NEON.

In the descriptions of each of the VFP and shared instructions, there is a subsection specifying the architectures that support the instruction.

ARMv7-M does not support either NEON or VFP.

Copyright © 2002-2007 ARM Limited. All rights reserved.

5.3.3 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 shows the data types available in VFP instructions.

Table 5-5 NEON data types

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

Table 5-6 VFP data types

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

See *Polynomial arithmetic over {0,1}* on page 5-16 for further information about operations on polynomials over {0,1}.

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

Note

Most instructions have a restricted range of permitted datatypes. See the instruction pages for details.

Copyright © 2002-2007 ARM Limited. All rights reserved.

5.3.4 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.

Copyright © 2002-2007 ARM Limited. All rights reserved.

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

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

Copyright © 2002-2007 ARM Limited. All rights reserved.

5.3.5 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.

5.3.6 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 7-14 for details.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

5.3.7 Polynomial arithmetic over {0,1}

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

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

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

Copyright © 2002-2007 ARM Limited. All rights reserved.

5.3.8 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 *RealView Compilation Tools Libraries and Floating Point Support 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 5-97 for further information.

5.3.9 VFP registers

The VFP coprocessor has 32 single-precision registers, s0 to s31. Each register can contain either a single-precision floating-point value, or a 32-bit integer.

These 32 registers are also treated as 16 double-precision registers, d0 to d15. `dn` occupies the same hardware as `s(2n)` and `s(2n+1)`.

VFPv3 extends the VFP register set by adding 16 further double-precision registers, d16 to d31. These do not overlap with any single-precision VFP registers.

Note

If your processor has both NEON and VFP, all the NEON registers overlap with the VFP registers, see *The extension register bank* on page 5-7.

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.

Copyright © 2002-2007 ARM Limited. All rights reserved.

5.4 Instructions shared by NEON and VFP

This section contains the following subsections:

 Copyright © 2002-2007 ARM Limited. All rights reserved.

5.4.1 VLDR and VSTR

Extension register load and store.

Syntax

`VLDR{cond}{.size} Fd, [Rn{, #offset}]`

`VSTR{cond}{.size} Fd, [Rn{, #offset}]`

`VLDR{cond}{.size} Fd, label`

`VSTR{cond}{.size} Fd, label`

where:

cond is an optional condition code (see *Condition codes* on page 5-9).

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 *Dd*. For a VFP instruction, it can be either *Dd* or *Sd*.

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 3-33 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 5-77).

Copyright © 2002-2007 ARM Limited. All rights reserved.

5.4.2 VLDM, VSTM, VPOP, and VPUSH

Extension register load multiple, store multiple, pop from stack, push onto stack.

Syntax

```
VLDMmode{cond} Rn,{!} Registers
VSTMmode{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.

cond is an optional condition code (see *Condition codes* on page 5-9).

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`.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

5.4.3 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

`VMOV{cond} Dm, Rd, Rn`

`VMOV{cond} Rd, Rn, Dm`

`VMOV{cond} {Sm, Sm1}, Rd, Rn`

`VMOV{cond} Rd, Rn, {Sm, Sm1}`

where:

cond	is an optional condition code (see *Condition codes* on page 5-9).
Dm	is a 64-bit extension register.
Sm	is a VFP 32-bit register.
Sm1	is the next VFP 32-bit register after *Sm*.
Rd, *Rn*	are the ARM registers. Do not use r15.

Usage

`VMOV Dm, Rd, Rn` transfers the contents of *Rd* into the low half of *Dm*, and the contents of *Rn* into the high half of *Dm*.

`VMOV Rd, Rn, Dm` transfers the contents of the low half of *Dm* into *Rd*, and the contents of the high half of *Dm* into *Rn*.

`VMOV Rd, Rn, {Sm, Sm1}` transfers the contents of *Sm* into *Rd*, and the contents of *Sm1* into *Rn*.

`VMOV {Sm, Sm1}, Rd, Rn` transfers the contents of *Rd* into *Sm*, and the contents of *Rn* into *Sm1*.

Architectures

The 64-bit instructions are available in NEON, and VFPv2 and above.

The 2 x 32-bit instructions are available in VFPv2 and above.

Copyright © 2002-2007 ARM Limited. All rights reserved.

5.4.4 VMOV (between an ARM register and a NEON scalar)

Transfer contents between an ARM register and a NEON scalar.

Syntax

`VMOV{cond}{.size} Dn[x], Rd`

`VMOV{cond}{.datatype} Rd, Dn[x]`

where:

cond is an optional condition code (see *Condition codes* on page 5-9).

size the data size. Can be `8`, `16`, or `32`. If omitted, *size* is `32`.

datatype the data type. Can `U8`, `S8`, `U16`, `S16`, or `32`. If omitted, *datatype* is `32`.

Dn[*x*] is the NEON scalar (see *NEON Scalars* on page 5-15).

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

Usage

`VMOV Rd, Dn[x]` transfers the contents of *Dn*[x] into the least signficant byte, halfword, or word of *Rd*.

`VMOV Dn[x], Rd` transfers the contents of the least signficant byte, halfword, or word of *Rd* into *Sn*.

Copyright © 2002-2007 ARM Limited. All rights reserved.

5.4.5 VMOV (between one ARM register and single precision VFP)

Transfer contents between a single-precision floating-point register and an ARM register.

Syntax

`VMOV{cond} Rd, Sn`

`VMOV{cond} Sn, Rd`

where:

cond	is an optional condition code (see *Condition codes* on page 5-9).
Sn	is the VFP single-precision register.
Rd	is the ARM register. *Rd* must not be r15.

Usage

`VMOV Rd, Sn` transfers the contents of *Sn* into *Rd*.

`VMOV Sn, Rd` transfers the contents of *Rd* into *Sn*.

Copyright © 2002-2007 ARM Limited. All rights reserved.

5.4.6 VMRS and VMSR

Transfer contents between an ARM register and a NEON and VFP system register.

Syntax

`VMRS{cond} Rd, extsysreg`

`VMSR{cond} extsysreg, Rd`

where:

cond is an optional condition code (see *Condition codes* on page 5-9).

extsysreg is the NEON and VFP system register, usually FPSCR, FPSID, or FPEXC (see *NEON and VFP system registers* on page 5-82).

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

It can be APSR_nzcv, if *extsysreg* is FPSCR. In this case, the floating-point status flags are transferred into the corresponding flags in the ARM APSR.

Usage

The VMRS instruction transfers the contents of *extsysreg* into *Rd*.

The VMSR instruction transfers the contents of *Rd* into *extsysreg*.

Note

These instructions stall the ARM until all current NEON or VFP operations complete.

Examples

```
VMRS    r2,FPCID
VMRS    APSR_nzcv, FPSCR     ; transfer FP status register to ARM APSR
VMSR    FPSCR, r4
```


Copyright © 2002-2007 ARM Limited. All rights reserved.

5.5 NEON logical and compare operations

This section contains the following subsections:

Copyright © 2002-2007 ARM Limited. All rights reserved.

5.5.1 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

V*op*{*cond*}.{*datatype*} {*Qd*}, *Qn*, *Qm*

V*op*{*cond*}.{*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 5-9).

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 5-29 for details.

Copyright © 2002-2007 ARM Limited. All rights reserved.

5.5.2 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 5-79.

Syntax

V*op*{*cond*}.*datatype Qd*, #*imm*

V*op*{*cond*}.*datatype Dd*, #*imm*

where:

op must be either BIC or ORR.

cond is an optional condition code (see *Condition codes* on page 5-9).

datatype must be either I16 or I32.

Qd or *Dd* is the NEON register for the source and result.

imm is the immediate constant.

Immediate constants

If *datatype* is I16, the immediate constant must correspond to one of the following forms:

- 0x00XY
- 0xXY00.

If *datatype* is I32, the immediate constant must correspond to one of the following forms:

- 0x000000XY
- 0x0000XY00
- 0x00XY0000
- 0xXY000000.

Copyright © 2002-2007 ARM Limited. All rights reserved.

5.5.3 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

V*op*{*cond*}{.*datatype*} {*Qd*}, *Qn*, *Qm*

V*op*{*cond*}{.*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 5-9).

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.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

5.5.4 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, Qm`

`VMVN{`*cond*`}{.`*datatype*`} Qd, Qm`

`VMVN{`*cond*`}{.`*datatype*`} Dd, Qm`

where:

cond is an optional condition code (see *Condition codes* on page 5-9).

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.

Copyright © 2002-2007 ARM Limited. All rights reserved.

5.5.5 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 5-80.

Syntax

`VAC`*`op`*`{`*`cond`*`}.F32 {`*`Qd`*`}, `*`Qn`*`, `*`Qm`*

`VAC`*`op`*`{`*`cond`*`}.F32 {`*`Dd`*`}, `*`Dn`*`, `*`Dm`*

where:

op must be one of:

- `GE` Absolute Greater than or Equal
- `GT` Absolute Greater Than.

cond is an optional condition code (see *Condition codes* on page 5-9).

Qd, Qn, Qm specifies the destination register, the first operand register, and the second operand register, for a quadword operation.

Dd, Dn, Dm specifies the destination register, the first operand register, and the second operand register, for a doubleword operation.

The result datatype is `I32`.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

5.5.6 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 5-81.

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 5-9).

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.

Copyright © 2002-2007 ARM Limited. All rights reserved.

5.5.7 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 5-9).

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.

Copyright © 2002-2007 ARM Limited. All rights reserved.

5.6 NEON general data processing instructions

This section contains the following subsections:

Copyright © 2002-2007 ARM Limited. All rights reserved.

5.6.1 VCVT

VCVT (Vector Convert) converts each element in a vector in one of the following ways, and places the results in a second 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 5-9).

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.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

5.6.2 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 5-9).

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.

Copyright © 2002-2007 ARM Limited. All rights reserved.

5.6.3 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.

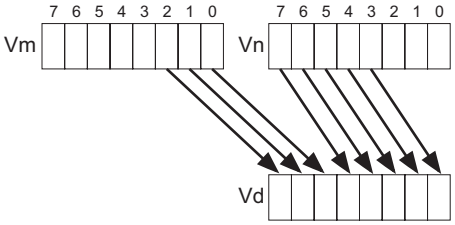

Figure 5-2 Operation of doubleword VEXT for imm = 3

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 5-9).

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.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

5.6.4 VMOV, VMVN (immediate)

Vector Move and Vector Move Negative (immediate) generate an immediate constant into the destination register.

Syntax

`Vop{cond}.datatype Qd, #imm`

`Vop{cond}.datatype Dd, #imm`

where:

op must be either `MOV` or `MVN`.

cond is an optional condition code (see *Condition codes* on page 5-9).

datatype must be one of `I8`, `I16`, `I32`, `I64`, or `F32`.

Qd or *Dd* is the NEON register for the result.

imm is a constant of the type specified by *datatype*. This is replicated to fill the destination register.

Table 5-8 Available constants

datatype	VMOV	VMVN
I8	0xXY	-
I16	0x00XY, 0xXY00	0xFFXY, 0xXYFF
I32	0x000000XY, 0x0000XY00, 0x00XY0000, 0xXY000000	0xFFFFFFXY, 0xFFFFXYFF, 0xFFXYFFFF, 0xXYFFFFFF
	0x0000XYFF, 0x00XYFFFF	0xFFFFXY00, 0xFFXY0000
I64	byte masks, 0xGGHHJJKKLLMMNNPP [a]	-
F32	floating-point numbers [b]	-

a. Each of 0xGG, 0xHH, 0xJJ, 0xKK, 0xLL, 0xMM, 0xNN, and 0xPP must be either 0x00 or 0xFF.

b. Any number that can be expressed as +/–n * 2^{-r}, where n and r are integers, 16 <= n <= 31, 0 <= r <= 7.

Copyright © 2002-2007 ARM Limited. All rights reserved.

5.6.5 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 5-9).

datatype — must be one of:

S8, S16, S32	for VMOVL
U8, U16, U62	for VMOVL
I16, I32, I64	for VMOVN
S16, S32, S64	for VQMOVN or VQMOVUN
U16, U32, U64	for VQMOVN.

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.

Copyright © 2002-2007 ARM Limited. All rights reserved.

5.6.6 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

VREV*n*{*cond*}.*size Qd*, *Qm*

VREV*n*{*cond*}.*size Dd*, *Dm*

where:

n must be one of 16, 32, or 64.

cond is an optional condition code (see *Condition codes* on page 5-9).

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.

Copyright © 2002-2007 ARM Limited. All rights reserved.

5.6.7 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 5-9).

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.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

5.6.8 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

V*op*{*cond*}.8 *Dd*, *list*, *Dm*

where:

op must be either TBL or TBX.

cond is an optional condition code (see *Condition codes* on page 5-9).

Dd specifies the destination vector.

list Specifies the vectors containing the table. It must be one of:

- {*Dn*}
- {*Dn*,*D(n+1)*}
- {*Dn*,*D(n+1)*,*D(n+2)*}
- {*Dn*,*D(n+1)*,*D(n+2)*,*D(n+3)*}.

All the registers in *list* must be in the range D0-D31.

Dm specifies the index vector.

Copyright © 2002-2007 ARM Limited. All rights reserved.

5.6.9 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.

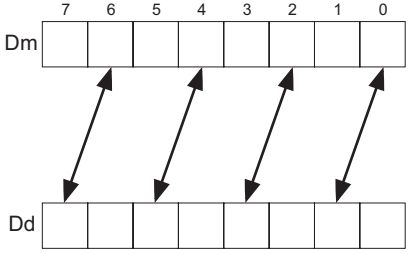

Figure 5-3 Operation of doubleword VTRN.8

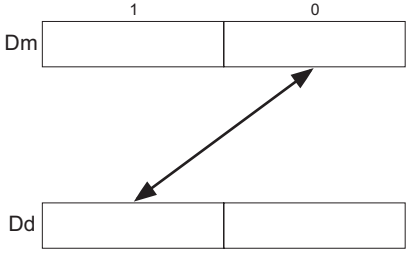

Figure 5-4 Operation of doubleword VTRN.32

Syntax

VTRN{*cond*}.*size Qd*, *Qm*

VTRN{*cond*}.*size Dd*, *Dm*

where:

cond is an optional condition code (see *Condition codes* on page 5-9).

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.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

5.6.10 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 5-68 for an example of de-interleaving. Interleaving is the inverse process.

Syntax

V*op*{*cond*}.*size Qd*, *Qm*

V*op*{*cond*}.*size Dd*, *Dm*

where:

op	must be either UZP or ZIP.
cond	is an optional condition code (see *Condition codes* on page 5-9).
size	must be one of 8, 16, or 32.
Qd, Qm	specifies the vectors, for a quadword operation.
Dd, Dm	specifies the vectors, for a doubleword operation.

Note

The following are all the same instruction:

- VZIP.32 *Dd*, *Dm*
- VUZP.32 *Dd*, *Dm*
- VTRN.32 *Dd*, *Dm*

The instruction is disassembled as VTRN.32 *Dd*, *Dm*. See also *VTRN* on page 5-42.

Copyright © 2002-2007 ARM Limited. All rights reserved.

5.7 NEON shift instructions

This section contains the following subsections:

Copyright © 2002-2007 ARM Limited. All rights reserved.

5.7.1 VSHL, VQSHL, VQSHLU, and VSHLL (by immediate)

Vector Shift Left, Vector Saturating Shift Left, and Vector Shift Left Long.

These 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, bits shifted out of the left of each element are lost..

For VQSHL and VQSHLU, the sticky QC flag (FPSCR bit[27]) is set if saturation occurs.

For VSHLL, values are sign or zero extended.

Syntax

V{Q}SHL{U}{*cond*}.*datatype* {*Qd*}, *Qm*, *#imm*

V{Q}SHL{U}{*cond*}.*datatype* {*Dd*}, *Dm*, *#imm*

VSHLL{*cond*}.*datatype Qd*, *Dm*, *#imm*

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 5-9).

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 VMOVL.

Copyright © 2002-2007 ARM Limited. All rights reserved.

5.7.2 V{Q}{R}SHL (by signed variable)

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

V{Q}{R}SHL{*cond*}.*datatype* {*Qd*}, *Qn*, *Qm*

V{Q}{R}SHL{*cond*}.*datatype* {*Dd*}, *Dn*, *Dm*

where:

Q — if present, indicates that If any of the results overflow, they are saturated.

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 5-9).

datatype — must be one of S8, S16, S32, S64, U8, U16, U32, or U64.

Qd, *Qn*, *Qm* — are the destination vector, the first operand vector, and the second operand vector, for a quadword operation.

Dd, *Dn*, *Dm* — are the destination vector, the first operand vector, and the second operand vector, for a doubleword operation.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

5.7.3 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

V{R}SHR{*cond*}.*datatype* {*Qd*}, *Qm*, *#imm*

V{R}SHR{*cond*}.*datatype* {*Dd*}, *Dm*, *#imm*

V{R}SRA{*cond*}.*datatype* {*Qd*}, *Qm*, *#imm*

V{R}SRA{*cond*}.*datatype* {*Dd*}, *Dm*, *#imm*

V{R}SHRN{*cond*}.*datatype Dd*, *Qm*, *#imm*

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 5-9).

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 (size(*datatype*) – 1).

Copyright © 2002-2007 ARM Limited. All rights reserved.

5.7.4 VQ{R}SHR{U}N (by immediate)

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.

Syntax

VQ{R}SHR{U}N{*cond*}.*datatype* *Dd*, *Qm*, #*imm*

where:

R — if present, indicates that the results are rounded. Otherwise, the results are truncated.

U — if present, indicates that the results are unsigned, although the operands are signed. Otherwise, the results are the same type as the operands.

cond — is an optional condition code (see *Condition codes* on page 5-9).

datatype — must be one of:

S16, S32, S64	for VQ{R}SHRN or VQ{R}SHRUN
U16, U32, U64	for VQ{R}SHRN only.

Dd, *Qm* — are the destination vector and the operand vector.

imm — is the immediate constant specifying the size of the shift, in the range 0 to (size(*datatype*) – 1).

Copyright © 2002-2007 ARM Limited. All rights reserved.

5.7.5 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

V*op*{*cond*}.*size* {*Qd*}, *Qm*, #*imm*

V*op*{*cond*}.*size* {*Dd*}, *Dm*, #*imm*

where:

op must be either SLI or SRI.

cond is an optional condition code (see *Condition codes* on page 5-9).

size must be one of 8, 16, 32, or 64.

Qd, *Qm* are the destination vector and the operand vector, for a quadword operation.

Dd, *Dm* are the destination vector and the operand vector, for a doubleword operation.

imm is the immediate constant specifying the size of the shift, in the range:

- 0 to (*size* – 1) for VSLI
- 1 to *size* for VSRI.

Copyright © 2002-2007 ARM Limited. All rights reserved.

5.8 NEON general arithmetic instructions

This section contains the following subsections:

 Copyright © 2002-2007 ARM Limited. All rights reserved.

5.8.1 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

V*op*{*cond*}.*datatype* {*Qd*}, *Qn*, *Qm*

V*op*{*cond*}.*datatype* {*Dd*}, *Dn*, *Dm*

V*op*L{*cond*}.*datatype Qd*, *Dn*, *Dm*

where:

op must be either ABA or ABD.

cond is an optional condition code (see *Condition codes* on page 5-9).

datatype must be one of:

- S8, S16, S32, U8, U16, or U32 for VABA, VABAL, or VABDL
- S8, S16, S32, U8, U16, U32 or F32 for VABD.

Qd, *Qn*, *Qm* are the destination vector, the first operand vector, and the second operand vector, for a quadword operation.

Dd, *Dn*, *Dm* are the destination vector, the first operand vector, and the second operand vector, for a doubleword operation.

Qd, *Dn*, *Dm* are the destination vector, the first operand vector, and the second operand vector, for a long operation.

Copyright © 2002-2007 ARM Limited. All rights reserved.

5.8.2 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 5-9).	
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.	

Copyright © 2002-2007 ARM Limited. All rights reserved.

5.8.3 V{Q}ADD, VADDL, VADDW, V{Q}SUB, VSUBL, and VSUBW

VADD (Vector Add) adds corresponding elements in two vectors, and places the results in the destination vector.

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.

Syntax

V{Q}*op*{*cond*}.*datatype* {*Qd*}, *Qn*, *Qm*

V{Q}*op*{*cond*}.*datatype* {*Dd*}, *Dn*, *Dm*

V*op*L{*cond*}.*datatype Qd*, *Dn*, *Dm*

V*op*W{*cond*}.*datatype* {*Qd*}, *Qn*, *Dm*

where:

Q — if present, indicates that if any of the results overflow, they are saturated.

op — must be either ADD or SUB.

cond — is an optional condition code (see *Condition codes* on page 5-9).

datatype — must be one of:

I8, I16, I32, I64, F32	for VADD or VSUB
S8, S16, S32	for VQADD, VQSUB, VADDL, VADDW, VSUBL, or VSUBW
U8, U16, U32	for VQADD, VQSUB, VADDL, VADDW, VSUBL, or VSUBW
S64, U64	for VQADD or VQSUB.

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.

Qd, *Qn*, *Dm* — are the destination vector, the first operand vector, and the second operand vector, for a wide operation.

Copyright © 2002-2007 ARM Limited. All rights reserved.

5.8.4 V{R}ADDHN and V{R}SUBHN

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.

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

V{R}*op*HN{*cond*}.*datatype Dd*, *Qn*, *Qm*

where:

R — if present, indicates that each result is rounded. Otherwise, each result is truncated.

op — must be either ADD or SUB.

cond — is an optional condition code (see *Condition codes* on page 5-9).

datatype — must be one of I16, I32, or I64.

Dd, *Qn*, *Qm* — are the destination vector, the first operand vector, and the second operand vector.

Copyright © 2002-2007 ARM Limited. All rights reserved.

5.8.5 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

V{R}HADD{*cond*}.*datatype* {*Qd*}, *Qn*, *Qm*

V{R}HADD{*cond*}.*datatype* {*Dd*}, *Dn*, *Dm*

VHSUB{*cond*}.*datatype* {*Qd*}, *Qn*, *Qm*

VHSUB{*cond*}.*datatype* {*Dd*}, *Dn*, *Dm*

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 5-9).

datatype — must be one of S8, S16, S32, U8, U16, or U32.

Qd, *Qn*, *Qm* — are the destination vector, the first operand vector, and the second operand vector, for a quadword operation.

Dd, *Dn*, *Dm* — are the destination vector, the first operand vector, and the second operand vector, for a doubleword operation.

Copyright © 2002-2007 ARM Limited. All rights reserved.

5.8.6 VPADD{L}, VPADAL

VPADD (Vector Pairwise Add) adds adjacent pairs of elements of two vectors, and places the results in the destination vector.

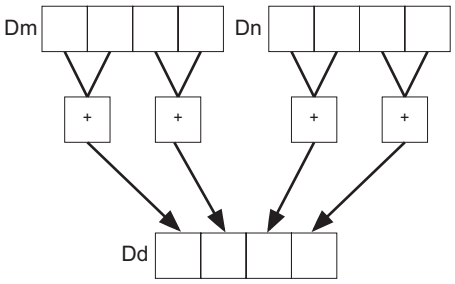

Figure 5-5 Example of operation of VPADD (in this case, for data type I16)

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.

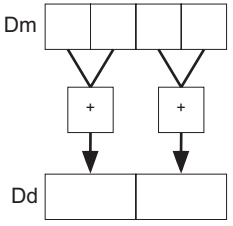

Figure 5-6 Example of operation of doubleword VPADDL (in this case, for data type S16)

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.

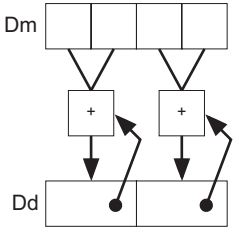

Figure 5-7 Example of operation of VPADAL (in this case for data type S16)

Copyright © 2002-2007 ARM Limited. All rights reserved.

Syntax

VPADD{*cond*}.*datatype* {*Dd*}, *Dn*, *Dm*

VP*op*L{*cond*}.*datatype Qd*, *Qm*

VP*op*L{*cond*}.*datatype Dd*, *Dm*

where:

op must be either ADD or ADA.

cond is an optional condition code (see *Condition codes* on page 5-9).

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.

Copyright © 2002-2007 ARM Limited. All rights reserved.

5.8.7 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 5-56 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 5-9).

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.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

5.8.8 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

V*op*{*cond*}.*datatype Qd*, *Qm*

V*op*{*cond*}.*datatype Dd*, *Dm*

where:

op must be one of CLS, CLZ, or CNT.

cond is an optional condition code (see *Condition codes* on page 5-9).

datatype must be one of:

- S8, S16, or S32 for CLS.
- I8, I16, or I32 for CLZ.
- I8 for CNT.

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.

Copyright © 2002-2007 ARM Limited. All rights reserved.

5.8.9 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

V*op*{*cond*}.*datatype Qd*, *Qm*

V*op*{*cond*}.*datatype Dd*, *Dm*

where:

op must be either RECPE or RSQRTE.

cond is an optional condition code (see *Condition codes* on page 5-9).

datatype must be either U32 or F32.

Qd, *Qm* are the destination vector and the operand vector, for a quadword operation.

Dd, *Dm* are the destination vector and the operand vector, for a doubleword operation.

Results for out-of-range inputs

Table 5-9 shows the results where input values are out of range.

Table 5-9 Results for out-of-range inputs

	Operand element (VRECPE)	Operand element (VRSQRTE)	Result element
Integer	<= 0x7FFFFFFF	<= 0x3FFFFFFF	0xFFFFFFFF
Floating-point	NaN	NaN, Negative Normal, Negative Infinity	Default NaN
	Negative 0, Negative Denormal	Negative 0, Negative Denormal	Negative Infinity [a]
	Positive 0, Positive Denormal	Positive 0, Positive Denormal	Positive Infinity [a]
	Positive infinity	Positive infinity	Positive 0
	Negative infinity		Negative 0

a. The Division by Zero exception bit in the FPSCR (FPSCR[1]) is set

Copyright © 2002-2007 ARM Limited. All rights reserved.

5.8.10 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

V*op*{*cond*}.F32 {*Qd*}, *Qn*, *Qm*

V*op*{*cond*}.F32 {*Dd*}, *Dn*, *Dm*

where:

op must be either RECPS or RSQRTS.

cond is an optional condition code (see *Condition codes* on page 5-9).

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 5-10 shows the results where input values are out of range.

Table 5-10 Results for out-of-range inputs

1st operand element	2nd operand element	Result element (VRECPS)	Result element (VRSQRTS)
NaN	-	Default NaN	Default NaN
-	NaN	Default NaN	Default NaN
+/– 0.0 or denormal	+/– infinity	2.0	1.5
+/– infinity	+/– 0.0 or denormal	2.0	1.5

Copyright © 2002-2007 ARM Limited. All rights reserved.

Usage

The Newton-Raphson iteration:

$x_{n+1} = x_n(2-dx_n)$

converges to $(1/d)$ if x_0 is the result of 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 VRSQRTE applied to d.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

5.9 NEON multiply instructions

This section contains the following subsections:

Copyright © 2002-2007 ARM Limited. All rights reserved.

5.9.1 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

V*op*{*cond*}.*datatype* {*Qd*}, *Qn*, *Qm*

V*op*{*cond*}.*datatype* {*Dd*}, *Dn*, *Dm*

V*op*L{*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 5-9).

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 5-16 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.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

5.9.2 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 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

V*op*{*cond*}.*datatype* {*Qd*}, *Qn*, *Dm*[*x*]

V*op*{*cond*}.*datatype* {*Dd*}, *Dn*, *Dm*[*x*]

V*op*L{*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 5-9).

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.

Copyright © 2002-2007 ARM Limited. All rights reserved.

5.9.3 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*L{*cond*}.*datatype Qd*, *Dn*, *Dm*

VQD*op*L{*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 5-9).

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.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

5.9.4 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 5-9).

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.

Copyright © 2002-2007 ARM Limited. All rights reserved.

5.10 NEON load / store element and structure instructions

This section contains the following subsections:

- *Interleaving*.
- *Alignment restrictions in load / store element and structure instructions* on page 5-69.
- *VLDn and VSTn (single n-element structure to one lane)* on page 5-70.

  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 5-72.
- *VLDn and VSTn (multiple n-element structures)* on page 5-74.

5.10.1 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.

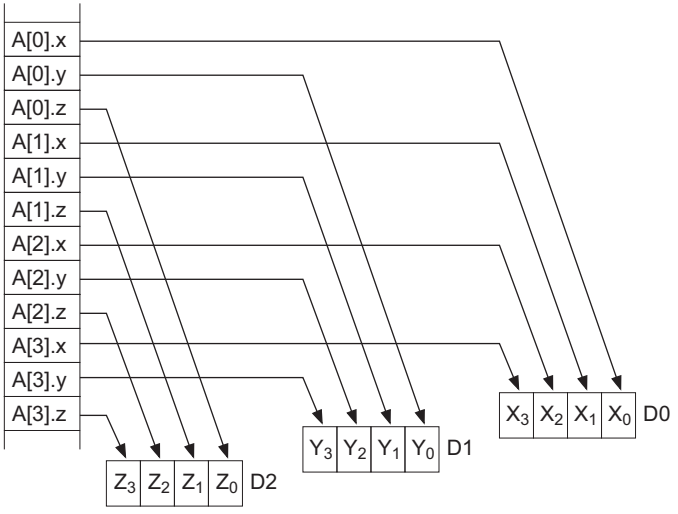

Figure 5-8 De-interleaving an array of 3-element structures

Copyright © 2002-2007 ARM Limited. All rights reserved.

5.10.2 Alignment restrictions in load / 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.

Copyright © 2002-2007 ARM Limited. All rights reserved.

5.10.3 VLD*n* and VST*n* (single *n*-element structure to one lane)

Vector Load single *n*-element structure to one lane loads one *n*-element structure from memory into one or more NEON registers. Elements of the register that are not loaded are unaltered.

Syntax

`V`*op*{*cond*}`.`*datatype* *list*`, [`*Rn*{`@`*align*}`]{!}`

`V`*op*{*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 5-9).

datatype see Table 5-11 on page 5-71.

list specifies the NEON register list. See Table 5-11 on page 5-71 for options.

Rn is the ARM register containing the base address. *Rn* cannot be R15.

align specifies an optional alignment. See Table 5-11 on page 5-71 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/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.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

Table 5-11 Permitted combinations of parameters

n	*datatype*	list [a]	*align* [b]	alignment
1	8	{Dd[x]}	-	Standard only
	16	{Dd[x]}	@16	2-byte
	32	{Dd[x]}	@32	4-byte
2	8	{Dd[x], D(d+1)[x]}	@16	2-byte
	16	{Dd[x], D(d+1)[x]}	@32	4-byte
		{Dd[x], D(d+2)[x]}	@32	4-byte
	32	{Dd[x], D(d+1)[x]}	@64	8-byte
		{Dd[x], D(d+2)[x]}	@64	8-byte
3	8, 16, or 32	{Dd[x], D(d+1)[x], D(d+2)[x]}	-	Standard only
		{Dd[x], D(d+2)[x], D(d+4)[x]}	-	Standard only
4	8	{Dd[x], D(d+1)[x], D(d+2)[x], D(d+3)[x]}	@32	4-byte
		{Dd[x], D(d+2)[x], D(d+4)[x], D(d+6)[x]}	@32	4-byte
	16	{Dd[x], D(d+1)[x], D(d+2)[x], D(d+3)[x]}	@64	8-byte
		{Dd[x], D(d+2)[x], D(d+4)[x], D(d+6)[x]}	@64	8-byte
	32	{Dd[x], D(d+1)[x], D(d+2)[x], D(d+3)[x]}	@64 or @128	8-byte or 16-byte
		{Dd[x], D(d+2)[x], D(d+4)[x], D(d+6)[x]}	@64 or @128	8-byte or 16-byte

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 / store element and structure instructions* on page 5-69.

Copyright © 2002-2007 ARM Limited. All rights reserved.

5.10.4 VLD*n* (single *n*-element structure to all lanes)

Vector Load single *n*-element structure to all lanes loads multiple copies of one *n*-element structure from memory into one or more NEON registers.

Syntax

`VLDn{cond}.datatype list, [Rn{@align}]{!}`

`VLDn{cond}.datatype list, [Rn{@align}], Rm`

where:

n must be one of 1, 2, 3, or 4.

cond is an optional condition code (see *Condition codes* on page 5-9).

datatype see Table 5-12 on page 5-73 .

list specifies the NEON register list. See Table 5-12 on page 5-73 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 5-73 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/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.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

Table 5-12 Permitted combinations of parameters

n	*datatype*	list [a]	*align* [b]	alignment
1	8	{Dd[]}	-	Standard only
		{Dd[],D(d+1)[]}	-	Standard only
	16	{Dd[]}	@16	2-byte
		{Dd[],D(d+1)[]}	@16	2-byte
	32	{Dd[]}	@32	4-byte
		{Dd[],D(d+1)[]}	@32	4-byte
2	8	{Dd[], D(d+1)[]}	@8	byte
		{Dd[], D(d+2)[]}	@8	byte
	16	{Dd[], D(d+1)[]}	@16	2-byte
		{Dd[], D(d+2)[]}	@16	2-byte
	32	{Dd[], D(d+1)[]}	@32	4-byte
		{Dd[], D(d+2)[]}	@32	4-byte
3	8, 16, or 32	{Dd[], D(d+1)[], D(d+2)[]}	-	Standard only
		{Dd[], D(d+2)[], D(d+4)[]}	-	Standard only
4	8	{Dd[], D(d+1)[], D(d+2)[], D(d+3)[]}	@32	4-byte
		{Dd[], D(d+2)[], D(d+4)[], D(d+6)[]}	@32	4-byte
	16	{Dd[], D(d+1)[], D(d+2)[], D(d+3)[]}	@64	8-byte
		{Dd[], D(d+2)[], D(d+4)[], D(d+6)[]}	@64	8-byte
	32	{Dd[], D(d+1)[], D(d+2)[], D(d+3)[]}	@64 or @128	8-byte or 16-byte
		{Dd[], D(d+2)[], D(d+4)[], D(d+6)[]}	@64 or @128	8-byte or 16-byte

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 / store element and structure instructions* on page 5-69.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

5.10.5 VLD*n* and VST*n* (multiple *n*-element structures)

Vector Load multiple *n*-element structures 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 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

`V`*op*{*cond*}`.`*datatype list*`, [`*Rn*{`@`*align*}`]{!}`

`V`*op*{*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 5-9).

datatype see Table 5-13 on page 5-75 for options.

list specifies the NEON register list. See Table 5-13 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 5-13 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/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.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

Table 5-13 Permitted combinations of parameters

n	*datatype*	list [a]	*align* [b]	*alignment*
1	8, 16, 32, or 64	{Dd}	@64	8-byte
		{Dd, D(d+1)}	@64 or @128	8-byte or 16-byte
		{Dd, D(d+1), D(d+2)}	@64	8-byte
		{Dd, D(d+1), D(d+2), D(d+3)}	@64, @128, or @256	8-byte, 16-byte, or 32-byte
2	8, 16, or 32	{Dd, D(d+1)}	@64, @128	8-byte or 16-byte
		{Dd, D(d+2)}	@64, @128	8-byte or 16-byte
		{Dd, D(d+1), D(d+2), D(d+3)}	@64, @128, or @256	8-byte, 16-byte, or 32-byte
3	8, 16, or 32	{Dd, D(d+1), D(d+2)}	@64	8-byte
		{Dd, D(d+2), D(d+4)}	@64	8-byte
4	8, 16, or 32	{Dd, D(d+1), D(d+2), D(d+3)}	@64, @128, or @256	8-byte, 16-byte, or 32-byte
		{Dd, D(d+2), D(d+4), D(d+6)}	@64, @128, or @256	8-byte, 16-byte, or 32-byte

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 / store element and structure instructions* on page 5-69.

Copyright © 2002-2007 ARM Limited. All rights reserved.

5.11 NEON and VFP pseudo-instructions

This section contains the following subsections:

 Copyright © 2002-2007 ARM Limited. All rights reserved.

5.11.1 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 5-19 for information on the VLDR *instruction*.

Syntax

VLDR{*cond*}.*datatype Dd*,=*constant*

VLDR{*cond*}.*datatype Sd*,=*constant*

where:

datatype must be one of:

I*n*	NEON only
S*n*	NEON only
U*n*	NEON only
F32	NEON or VFP
F64	VFP only

n must be one of 8, 16, 32, or 64.

cond is an optional condition code (see *Condition codes* on page 5-9).

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 double-word literal pool entry containing the constant and loads the constant using a VLDR instruction.

Copyright © 2002-2007 ARM Limited. All rights reserved.

5.11.2 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 5-9).

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 5-37 and *VBIC and VORR (immediate)* on page 5-27 for details.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

5.11.3 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

`Vop{cond}.datatype Qd, #imm`

`Vop{cond}.datatype Dd, #imm`

where:

op	must be either VAND or VORN.
cond	is an optional condition code (see *Condition codes* on page 5-9).
datatype	must be either I16, or I32.
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 correspond to one of the following forms:

- 0xFFXY
- 0xXYFF.

If *datatype* is I32, the immediate constant must correspond to one of the following forms:

- 0xFFFFFFXY
- 0xFFFFXYFF
- 0xFFXYFFFF
- 0xXYFFFFFF.

Copyright © 2002-2007 ARM Limited. All rights reserved.

5.11.4 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

`VAC`*op*`{`*cond*`}.`*datatype* `{`*Qd*`}, `*Qn*`, `*Qm*

`VAC`*op*`{`*cond*`}.`*datatype* `{`*Dd*`}, `*Dn*`, `*Dm*

where:

op must be one of:

- `LE` Absolute Less than or Equal
- `LT` Absolute Less Than.

cond is an optional condition code (see *Condition codes* on page 5-9).

datatype must be `F32`.

Qd or *Dd* is the NEON register for the result.
The result datatype is `I32`.

Qn or *Dn* is the NEON register holding the first operand.

Qm or *Dm* is the NEON register holding the second operand.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

5.11.5 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.

Note

On disassembly, these pseudo-instructions are disassembled to the corresponding `VCGE` and `VCGT` instructions, with the operands reversed.

Syntax

`VC`*op*`{`*cond*`}.`*datatype* `{`*Qd*`}, `*Qn*`, `*Qm*

`VC`*op*`{`*cond*`}.`*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 5-9).

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.

Copyright © 2002-2007 ARM Limited. All rights reserved.

5.12 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 5-84
- *FPSID, the floating-point system ID register* on page 5-84
- *Modifying individual bits of a NEON and VFP system register* on page 5-85.

A particular implementation of NEON or VFP can have additional registers (see the technical reference manual for the VFP coprocessor you are using).

5.12.1 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 5-9).

bit[27] Is the QC, cumulative saturation flag. This is set if saturation occurs in NEON or VFP saturating instructions.

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 5-86).

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 *Round to Nearest* (RN) mode.

0b01 *Round towards Plus infinity* (RP) mode.

0b10 *Round towards Minus infinity* (RM) mode.

0b11 *Round towards Zero* (RZ) mode.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

bits[21:20] STRIDE is the distance between successive values in a vector (see *Vectors* on page 5-98). Stride is controlled as follows:

0b00 STRIDE = 1

0b11 STRIDE = 2.

bits[18:16] LEN is the number of registers used by each vector (see *Vectors* on page 5-98). It is 1 + the value of bits[18:16]:

0b000 LEN = 1

...

0b111 LEN = 8.

bits[12:8] Are the exception trap enable bits:

IXE inexact exception enable

UFE underflow exception enable

OFE overflow exception enable

DZE division by zero exception enable

IOE invalid operation exception enable.

This manual does not cover the use of floating-point exception trapping. For information see the technical reference manual for the VFP coprocessor you are using.

bits[4:0] Are the cumulative exception bits:

IXC inexact exception

UFC underflow exception

OFC overflow exception

DZC division by zero exception

IOC invalid operation exception.

Cumulative exception bits are set when the corresponding exception occurs. They remain set until you clear them by writing directly to the FPSCR.

all other bits Are unused in the basic 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 5-85).

Note

The use of vector mode is deprecated in new code. Set LEN and STRIDE to 1.

Copyright © 2002-2007 ARM Limited. All rights reserved.

5.12.2 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 5-85).

5.12.3 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.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

5.12.4 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:

```
VMRS    r10,FPSCR           ; copy FPSCR into r10
BIC     r10,r10,#0x00370000 ; clears STRIDE and LEN
ORR     r10,r10,#0x00030000 ; sets STRIDE = 1, LEN = 4
VMSR    FPSCR,r10           ; copy r10 back into FPSCR
```

See *VMRS and VMSR* on page 5-24.

Copyright © 2002-2007 ARM Limited. All rights reserved.

5.13 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.

5.13.1 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.

5.13.2 The effects of using flush-to-zero mode

With certain exceptions (see *Operations not affected by flush-to-zero mode* on page 5-87), 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.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

- 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.

5.13.3 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 5-29, *VABS, VNEG, and VSQRT* on page 5-89, and *V{Q}ABS and V{Q}NEG* on page 5-52).
- Duplicate (see *VDUP* on page 5-35).
- Swap (see *VSWP* on page 5-40).
- Load and store (see *VLDR and VSTR* on page 5-19).
- Load multiple and store multiple (see *VLDM, VSTM, VPOP, and VPUSH* on page 5-20).
- Transfer between extension registers and ARM general-purpose registers (see *VMOV (between two ARM registers and an extension register)* on page 5-21, *VMOV (between an ARM register and a NEON scalar)* on page 5-22, and *VMOV (between one ARM register and single precision VFP)* on page 5-23).

Copyright © 2002-2007 ARM Limited. All rights reserved.

5.14 VFP instructions

This section contains the following subsections:

 Copyright © 2002-2007 ARM Limited. All rights reserved.

5.14.1 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 5-99).

Syntax

`V`*op*`{`*cond*`}.F32` *Sd*, *Sm*

`V`*op*`{`*cond*`}.F64` *Dd*, *Dm*

where:

op is one of ABS, NEG, or SQRT.

cond is an optional condition code (see *Condition codes* on page 5-9).

Sd, *Sm* are the single-precision registers for the result and operand.

Dd, *Dm* are the double-precision registers for the result and operand.

Usage

The VABS instruction takes the contents of *Sm* or *Dm*, clears the sign bit, and places the result in *Sd* or *Dd*. This gives the absolute value.

The VNEG instruction takes the contents of *Sm* or *Dm*, changes the sign bit, and places the result in *Sd* or *Dd*. This gives the negation of the value.

The 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 VABS and 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

VABS and VNEG instructions cannot produce any exceptions.

VSQRT instructions can produce Invalid Operation or Inexact exceptions.

Copyright © 2002-2007 ARM Limited. All rights reserved.

5.14.2 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 5-99).

Syntax

`V`*`op`*`{`*`cond`*`}.F32 {`*`Sd`*`}, `*`Sn`*`, `*`Sm`*

`V`*`op`*`{`*`cond`*`}.F64 {`*`Dd`*`}, `*`Dn`*`, `*`Dm`*

where:

op is one of `ADD`, `SUB`, or `DIV`.

cond is an optional condition code (see *Condition codes* on page 5-9).

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 `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 regiser.

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 regiser.

Floating-point exceptions

`VADD` and `VSUB` instructions can produce Invalid Operation, Overflow, or Inexact exceptions.

`FDIV` operations can produce Division by Zero, Invalid Operation, Overflow, Underflow, or Inexact exceptions.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

5.14.3 VMUL, VMLA, VMLS, VNMUL, 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 5-99).

Syntax TBD optional destinations?

`V{N}op{cond}.F32 Sd, Sn, Sm`

`V{N}op{cond}.F64 Dd, Dn, Dm`

where:

`N` negates the final result.

op is one of `MUL`, `MLA`, or `MLS`.

cond is an optional condition code (see *Condition codes* on page 5-9).

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 regiser.

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 regiser.

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.

Copyright © 2002-2007 ARM Limited. All rights reserved.

5.14.4 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 5-9).

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 opeand register, and sets the VFP condition flags on the result (see *Condition codes* on page 5-9).

Floating-point exceptions

VCMP instructions can produce Invalid Operation exceptions.

Copyright © 2002-2007 ARM Limited. All rights reserved.

5.14.5 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 5-9).
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.
SD	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 exceptions.

Copyright © 2002-2007 ARM Limited. All rights reserved.

5.14.6 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 5-9).

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.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

5.14.7 VCVT (between floating-point and fixed-point)

Convert between floating-point and fixed-point numbers.

VCVT is always scalar.

Syntax

VCVT{*cond*}.*type*.F64 *Dd, Dd*, *#fbits*

VCVT{*cond*}.*type*.F32 *Sd, Sd*, *#fbits*

VCVT{*cond*}.F64.*type Dd, Dd*, *#fbits*

VCVT{*cond*}.F32.*type Sd, Sd*, *#fbits*

where:

cond is an optional condition code (see *Condition codes* on page 5-9).

type can be any one of:

- S16 16-bit signed fixed-point number
- U16 16-bit unsigned fixed-point number
- S32 32-bit signed fixed-point number
- U32 32-bit unsigned fixed-point number.

Sd is a single-precision register for the operand and result.

Dd is a double-precision register for the operand and result.

fbits is the number of fraction bits in the fixed-point number, in the range 0-16 if *type* is S16 or U16, or in the range 1-32 if *type* is S32 or U32.

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.

Copyright © 2002-2007 ARM Limited. All rights reserved.

5.14.8 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

`VMOV{cond}.F32 Sd, #imm`

`VMOV{cond}.F64 Dd, #imm`

`VMOV{cond}.F32 Sd, Sm`

`VMOV{cond}.F64 Dd, Dm`

where:

cond is an optional condition code (see *Condition codes* on page 5-9).

Sd is the single-precision destination register.

Dd is the double-precision destination register.

imm is the floating-point constant.

Sm is the single-precision source register.

Dm is the double-precision source register.

Constant values

Any number that can be expressed as +/–$n * 2^{-r}$, where n and r are integers, 16 <= n <= 31, 0 <= r <= 7.

Architectures

This instruction is available in VFPv3.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

5.15 VFP vector mode

Most arithmetic instructions can be used on these vectors, enabling *Single Instruction Multiple Data* (SIMD) parallelism. In addition, the floating-point load and store instructions have multiple register forms, enabling vectors to be transferred to and from memory.

For more details of the VFP coprocessor, see the *ARM Architecture Reference Manual*.

Note

The use of VFP vector mode is deprecated in new code.

5.15.1 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.

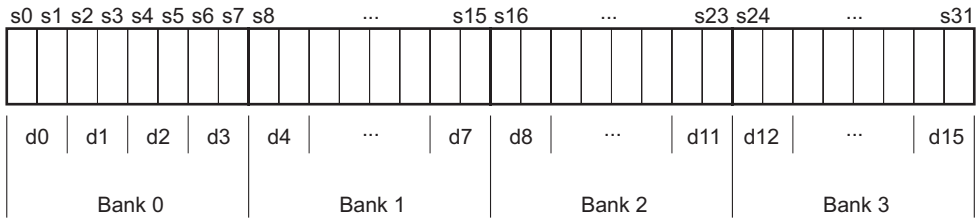

Figure 5-9 VFPv2 register banks

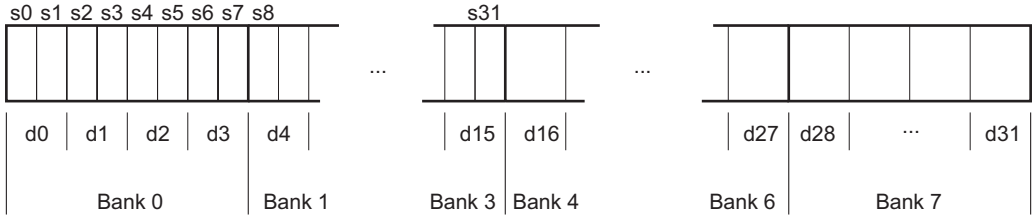

Figure 5-10 VFPv3 register banks

Copyright © 2002-2007 ARM Limited. All rights reserved.

5.15.2 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 5-82).

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 5-82). 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.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

5.15.3 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 5-82).

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.

Copyright © 2002-2007 ARM Limited. All rights reserved.

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 5-97).

Given instructions of the following general forms:

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

the behavior is as follows:

- If *Fd* is in the first 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

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

Vector operations

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

Mixed scalar and vector operations

For single-operand instructions, *Op* acts on the single value in *Fm*. LEN copies of the result are placed in the vector starting at *Fd*.

For multiple-operand instructions, *Op* acts on the single value in *Fm*, together with the values in the vector starting at *Fn*. The results are placed in the vector starting at *Fd*.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

5.15.4 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 in new code, and vector notation is not supported in Unified Assembler Language. To use vector notation, you must use the old VFP mnemonics. See *Pre-UAL VFP mnemonics* on page 5-102 for details. You can mix old 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 5-105
- *VFPASSERT VECTOR* on page 5-106.

If you use `VFPASSERT` directives, you must specify vector details in all VFP data processing instructions written using old mnemonics. The vector notation is described in *Vector notation* on page 5-104. If you do not use `VFPASSERT` directives you must not use this notation.

Copyright © 2002-2007 ARM Limited. All rights reserved.

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 FABSS.

Where UAL mnemonics use .F64 to specify double-precision data, pre-UAL mnemonics use D appended to the instruction mnemonic. For example, VCMPE.64 was FCMPED.

Table 5-14 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 5-14 Pre-UAL VFP mnemonics

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

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 5-103 for details.

Copyright © 2002-2007 ARM Limited. All rights reserved.

Immediate values in FCONST

Table 5-15 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 represenation of the binary number `abcdefgh`, where:

`a` is 0 for positive numbers, or 1 for negative numbers

`bcd` is shown in the column headings

`efgh` is shown in the row headings.

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

Table 5-15 Floating-point constant values

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

Copyright © 2002-2007 ARM Limited. All rights reserved.

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 7-14).

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

Copyright © 2002-2007 ARM Limited. All rights reserved.

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 5-104
- *VFPASSERT VECTOR* on page 5-106.

Note

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

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

Example

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


Copyright © 2002-2007 ARM Limited. All rights reserved.

VFPASSERT VECTOR

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

Syntax

```
VFPASSERT VECTOR[<[n[:s]]>]
```

where:

n is the vector length, 1-8.

s is the vector stride, 1-2.

Usage

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

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

If a function expects the VFP to be in vector mode on entry, place a VFPASSERT VECTOR directive immediately before the first instruction. Such a function would not be AAPCS conformant. See the *Procedure Call Standard for the ARM Architecture* specification, aapcs.pdf, in *install_directory*\Documentation\Specifications\... for more information.

See:

- *Vector notation* on page 5-104
- *VFPASSERT SCALAR* on page 5-105.

Note

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

Example

```
VMRS    r10,FPSCR          ; UAL mnemonic - could be FMRX instead.
BIC     r10,r10,#0x00370000
ORR     r10,r10,#0x00020000    ; set length = 3, stride = 1
VMSR    FPSCR,r10

VFPASSERT VECTOR           ; assert vector mode, unspecified length & stride
faddd  d4, d4, d0          ; ERROR, scalar in vector mode
fadds  s16<3>, s0, s8<3>   ; okay
```


Copyright © 2002-2007 ARM Limited. All rights reserved.

```
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.)
```


Copyright © 2002-2007 ARM Limited. All rights reserved.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

Chapter 6
Wireless MMX Technology Instructions

This chapter describes support for Wireless MMX™ Technology instructions in the ARM® assembler, `armasm`. It contains the following sections:

Copyright © 2002-2007 ARM Limited. All rights reserved.

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 RVCT. 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.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

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 6-1 Status and Control registers

Type	Wireless MMX Technology registers	CP1 registers
Coprocessor ID	`wCID`	`c0`
Control	`wCon`	`c1`
Saturation SIMD flag	`wCSSF`	`c2`
Arithmetic SIMD flag	`wCASF`	`c3`
Reserved	-	`c4` - `c7`
General-purpose	`wCGR0` - `wCGR3`	`c8` - `c11`
Reserved	-	`c12` - `c15`

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.

Copyright © 2002-2007 ARM Limited. All rights reserved.

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:

```
speed WCN  wcgr0 ; defines speed as a symbol for control reg 0
```

WRN Defines a name for a specified SIMD Data register, for example:

```
rate WRN  wr6 ; 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 3-19 or the register names described in *Registers* on page 6-3.

6.2.3 Frame directives

Wireless MMX Technology registers can be used with `FRAME` directives in the usual way to add debug information into your object files (see *Frame directives* on page 7-40 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 7-44).
- Wireless MMX Technology registers cannot be used as address offsets (see *FRAME ADDRESS* on page 7-42 and *FRAME RETURN ADDRESS* on page 7-48).

 Copyright © 2002-2007 ARM Limited. All rights reserved.

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

```
op<type>{cond} wRd, [Rn {, #{-}offset}]{!}
op<type>{cond} wRd, [Rn], #{-}offset
opW{cond} wRd, label
opD{cond} wRd, label
opD wRd, [Rn], {-}Rm {, LSL #imm4}]{!}      ; MMX2 only
opD wRd, [Rn], {-}Rm {, LSL #imm4}          ; MMX2 only
opW wCd, [Rn {, #{-}offset}]{!}
opW wCd, [Rn], #{-}offset
```

where:

op can be either:

- WLDR — Load Wireless MMX Register
- WSTR — Store Wireless MMX Register.

<*type*> can be any one of:

- B — Byte
- H — Halfword
- W — Word
- D — Doubleword.

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

wRd is the Wireless MMX register to load or save.

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

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

! is an optional suffix. If ! is present, the instruction is a pre-indexed instruction.

label is a program-relative expression. See *Register-relative and program-relative expressions* on page 3-33 for more information.

label must be within +/- 1020 bytes of the current instruction.

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

imm4 contains the number of bits to shift *Rm* left, in the range 0-15.

Copyright © 2002-2007 ARM Limited. All rights reserved.

Loading constants into SIMD registers

The assembler also supports the `WLDRW` literal load pseudo-instruction, for example:

```
WLDRW wr0, =0x114
```

Be aware that:

- The assembler cannot load byte and halfword literals. These produce a downgradable error. If downgraded, the instruction is converted to a `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 `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 `TMIA` instructions have a `MIA` mnemonic that overlaps with the XScale `MIA` instructions. Be aware that:
  - `MIA acc0, Rm, Rs` is accepted in XScale, but faulted in Wireless MMX Technology.
  - `MIA wR0, Rm, Rs` and `TMIA wR0, Rm, Rs` are accepted in Wireless MMX Technology.
  - `TMIA acc0, Rm, Rs` is faulted in XScale (XScale has no `TMIA` instruction).

For more details on the XScale instructions, see *Miscellaneous instructions* on page 4-128.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

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 6-2 Wireless MMX Technology instructions

Mnemonic	Example
TANDC	TANDCB r15
TBCST	TBCSTB wr15, r1
TEXTRC	TEXTRCB r15, #0
TEXTRM	TEXTRMUBCS r3, wr7, #7
TINSR	TINSRB wr6, r11, #0
TMIA, TMIAPH, TMIAxy	TMIANE wr1, r2, r3 TMIAPH wr4, r5, r6 TMIABB wr4, r5, r6 MIAPHNE wr4, r5, r6
TMOVMSK	TMOVMSKBNE r14, wr15
TORC	TORCB r15
WACC	WACCBGE wr1, wr2
WADD	WADDBGE wr1, wr2, wr13
WALIGNI, WALIGNR	WALIGNI wr7, wr6, wr5,#3 WALIGNR0 wr4, wr8, wr12
WAND, WANDN	WAND wr1, wr2, wr3 WANDN wr5, wr5, wr9
WAVG2	WAVG2B wr3, wr6, wr9 WAVG2BR wr4, wr7, wr10
WCMPEQ	WCMPEQB wr0, wr4, wr2
WCMPGT	WCMPGTUB wr0, wr4, wr2
WLDR	WLDRB wr1, [r2, #0]
WMAC	WMACU wr3, wr4, wr5

Copyright © 2002-2007 ARM Limited. All rights reserved.

Table 6-2 Wireless MMX Technology instructions (continued)

Mnemonic	Example
WMADD	WMADDU wr3, wr4, wr5
WMAX, WMIN	WMAXUB wr0, wr4, wr2 WMINSB wr0, wr4, wr2
WMUL	WMULUL wr4, wr2, wr3
WOR	WOR wr3, wr1, wr4
WPACK	WPACKHUS wr2, wr7, wr1
WROR	WRORH wr3, wr1, wr4
WSAD	WSADB wr3, wr5, wr8
WSHUFH	WSHUFH wr8, wr15, #17
WSLL, WSRL	WSLLH wr3, wr1, wr4 WSRLHG wr3, wr1, wcgr0
WSRA	WSRAH wr3, wr1, wr4 WSRAHG wr3, wr1, wcgr0
WSTR	WSTRB wr1, [r2, #0] WSTRW wc1, [r2, #0]
WSUB	WSUBBGE wr1, wr2, wr13
WUNPCKEH, WUNPCKEL	WUNPCKEHUB wr0, wr4 WUNPCKELSB wr0, wr4
WUNPCKIH, WUNPCKIL	WUNPCKIHB wr0, wr4, wr2 WUNPCKILH wr1, wr5, wr3
WXOR	WXOR wr3, wr1, wr4

Copyright © 2002-2007 ARM Limited. All rights reserved.

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 6-3 Wireless MMX Technology pseudo-instructions

Mnemonic	Brief description	Example
TMCR	Moves the contents of source register, *Rn*, to Control register, *wCn*. Maps onto the ARM MCR coprocessor instruction (page 4-122).	`TMCR wc1, r10`
TMCRR	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 4-122).	`TMCRR wr4, r5, r6`
TMRC	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 4-124).	`TMRC r1, wc2`
TMRRC	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 4-124).	`TMRRC r1, r0, wr2`
WMOV	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).	`WMOV wr1, wr8`
WZERO	Zeros destination register, *wRd*. This instruction is a form of WANDN (see Table 6-2 on page 6-7).	`WZERO wr1`

Copyright © 2002-2007 ARM Limited. All rights reserved.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

Chapter 7
Directives Reference

This chapter describes the directives that are provided by the ARM® assembler, `armasm`. It contains the following sections:

Note

None of these directives is available in the inline assemblers in the ARM C and C++ compilers.

Copyright © 2002-2007 ARM Limited. All rights reserved.

7.1 Alphabetical list of directives

Table 7-1 shows a complete list of the directives. Use it to locate individual directives described in the rest of this chapter.

Table 7-1 Location of directives

Directive	Page	Directive	Page	Directive	Page
ALIGN	page 7-63	EXPORT *or* GLOBAL	page 7-71	LTORG	page 7-18
ARM *and* CODE32	page 7-61	EXPORTAS	page 7-73	MACRO *and* MEND	page 7-32
AREA	page 7-66	EXTERN	page 7-75	MAP	page 7-19
ASSERT	page 7-55	FIELD	page 7-20	MEND *see* MACRO	page 7-32
CN	page 7-12	FRAME ADDRESS	page 7-42	MEXIT	page 7-35
CODE16	page 7-61	FRAME POP	page 7-43	NOFP	page 7-79
COMMON	page 7-30	FRAME PUSH	page 7-44	OPT	page 7-57
CP	page 7-13	FRAME REGISTER	page 7-46	PRESERVE8 *see* REQUIRE8	page 7-80
DATA	page 7-30	FRAME RESTORE	page 7-47	PROC *see* FUNCTION	page 7-53
DCB	page 7-22	FRAME SAVE	page 7-49	QN	page 7-14
DCD *and* DCDU	page 7-23	FRAME STATE REMEMBER	page 7-50	RELOC	page 7-8
DCDO	page 7-24	FRAME STATE RESTORE	page 7-51	REQUIRE	page 7-79
DCFD *and* DCFDU	page 7-25	FRAME UNWIND ON *or* OFF	page 7-52	REQUIRE8 *and* PRESERVE8	page 7-80
DCFS *and* DCFSU	page 7-26	FUNCTION *or* PROC	page 7-53	RLIST	page 7-11
DCI	page 7-27	GBLA, GBLL, *and* GBLS	page 7-4	RN	page 7-10
DCQ *and* DCQU	page 7-28	GET *or* INCLUDE	page 7-74	ROUT	page 7-81
DCW *and* DCWU	page 7-29	GLOBAL *see* EXPORT	page 7-71	SETA, SETL, *and* SETS	page 7-7
DN	page 7-14	IF, ELSE, ENDIF, *and* ELIF	page 7-36	SN	page 7-14
ELIF, ELSE *see* IF	page 7-36	IMPORT	page 7-75	SPACE	page 7-21
END	page 7-69	INCBIN	page 7-77	SUBT	page 7-59
ENDFUNC *or* ENDP	page 7-54	INCLUDE *see* GET	page 7-74	THUMB	page 7-61
ENDIF *see* IF	page 7-36	INFO	page 7-56	THUMBX	page 7-61
ENTRY	page 7-69	KEEP	page 7-78	TTL	page 7-59
EQU	page 7-70	LCLA, LCLL, *and* LCLS	page 7-6	WHILE *and* WEND	page 7-39

Copyright © 2002-2007 ARM Limited. All rights reserved.

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 7-10
  Define a name for a specified register.
- *RLIST* on page 7-11
  Define a name for a set of general-purpose registers.
- *CN* on page 7-12
  Define a coprocessor register name.
- *CP* on page 7-13
  Define a coprocessor name.
- *QN, DN, and SN* on page 7-14
  Define a double-precision or single-precision VFP register name.

Copyright © 2002-2007 ARM Limited. All rights reserved.

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.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

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

```
            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

```
armasm --predefine "objectsize SETA 0xFF" sourcefile -o objectfile
```


Copyright © 2002-2007 ARM Limited. All rights reserved.

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                           ; Declare a macro
$label  message $a                      ; Macro prototype line
        LCLS    err                     ; Declare local string
                                        ; variable err.
err     SETS    "error no: "            ; Set value of err
$label  ; code
        INFO    0, "err":CC::STR:$a     ; Use string
        MEND
```

 Copyright © 2002-2007 ARM Limited. All rights reserved.

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

variable <*setx*> *expr*

where:

<*setx*> is one of SETA, SETL, or SETS.

variable is the name of a variable declared by a GBLA, GBLL, GBLS, LCLA, LCLL, or LCLS directive.

expr is an expression that is:

- numeric, for SETA (see *Numeric expressions* on page 3-30)
- logical, for SETL (see *Logical expressions* on page 3-33)
- string, for SETS (see *String expressions* on page 3-29).

Usage

You must declare *variable* using a global or local declaration directive before using one of these directives. See *GBLA, GBLL, and GBLS* on page 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

```
                GBLA    VersionNumber
VersionNumber   SETA    21

                GBLL    Debug
Debug           SETL    {TRUE}

                GBLS    VersionString
VersionString   SETS    "Version 1.0"
```


Copyright © 2002-2007 ARM Limited. All rights reserved.

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.

Copyright © 2002-2007 ARM Limited. All rights reserved.

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
```


Copyright © 2002-2007 ARM Limited. All rights reserved.

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 3-19.

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
```

 Copyright © 2002-2007 ARM Limited. All rights reserved.

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 3-19.

list-of-registers
is a comma-delimited list of register names and/or register ranges. The register list must be enclosed in braces.

Usage

Use RLIST to give a name to a set of registers to be transferred by the LDM or STM instructions.

LDM and STM always put the lowest physical register numbers at the lowest address in memory, regardless of the order they are supplied to the LDM or STM instruction. If you have defined your own symbolic register names it can be less apparent that a register list is not in increasing register order.

Use the --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}
```


Copyright © 2002-2007 ARM Limited. All rights reserved.

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 3-19.

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
```

 Copyright © 2002-2007 ARM Limited. All rights reserved.

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

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 3-19.

expr evaluates to a coprocessor number from 0 to 15.

Usage

Use CP to allocate convenient names to coprocessors, to help you to remember what you use each one for.

Note

Avoid conflicting uses of the same coprocessor under different names.

The names p0 to p15 are predefined for coprocessors 0 to 15.

Example

```
dmu    CP  6       ; defines dmu as a symbol for
                   ; coprocessor 6
```


Copyright © 2002-2007 ARM Limited. All rights reserved.

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 3-19.

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.

type is any datatype described in *NEON and VFP data types* on page 5-12.

[*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 5-15 for more information, and *Extended notation example* on page 7-15 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 5-101).

 Copyright © 2002-2007 ARM Limited. All rights reserved.

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]
```


Copyright © 2002-2007 ARM Limited. All rights reserved.

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 7-18

  Set an origin for a literal pool.

- *MAP* on page 7-19

  Set the origin of a storage map.

- *FIELD* on page 7-20

  Define a field within a storage map.

- *SPACE* on page 7-21

  Allocate a zeroed block of memory.

- *DCB* on page 7-22

  Allocate bytes of memory, and specify the initial contents.

- *DCD and DCDU* on page 7-23

  Allocate words of memory, and specify the initial contents.

- *DCDO* on page 7-24

  Allocate words of memory, and specify the initial contents as offsets from the static base register.

- *DCFD and DCFDU* on page 7-25

  Allocate doublewords of memory, and specify the initial contents as double-precision floating-point numbers.

- *DCFS and DCFSU* on page 7-26

  Allocate words of memory, and specify the initial contents as single-precision floating-point numbers.

- *DCI* on page 7-27

  Allocate words of memory, and specify the initial contents. Mark the location as code not data.

- *DCQ and DCQU* on page 7-28

  Allocate doublewords of memory, and specify the initial contents as 64-bit integers.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

- *DCW and DCWU* on page 7-29
  Allocate halfwords of memory, and specify the initial contents.
- *COMMON* on page 7-30
  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.

Copyright © 2002-2007 ARM Limited. All rights reserved.

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, FLDD, and FLDS pseudo-instructions. See *LDR pseudo-instruction* on page 4-153. Use LTORG to ensure that a literal pool is assembled within range. Large programs can require several literal pools.

Place LTORG directives after unconditional branches or subroutine return instructions so that the processor does not attempt to execute the constants as instructions.

The assembler word-aligns data in literal pools.

Example

```
        AREA    Example, CODE, READONLY
start   BL      func1

func1                           ; function body
        ; code
        LDR     r1,=0x55555555  ; => LDR R1, [pc, #offset to Literal Pool 1]
        ; code
        MOV     pc,lr           ; end function
        LTORG                   ; Literal Pool 1 contains literal &55555555.

data    SPACE   4200            ; Clears 4200 bytes of memory,
                                ; starting at current location.
        END                     ; Default literal pool is empty.
```

 Copyright © 2002-2007 ARM Limited. All rights reserved.

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 7-20 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
```

 Copyright © 2002-2007 ARM Limited. All rights reserved.

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:

label is an optional label. If specified, *label* is assigned the value of the storage location counter, {VAR}. The storage location counter is then incremented by the value of *expr*.

expr is an expression that evaluates to the number of bytes to increment the storage counter.

Usage

If a storage map is set by a MAP directive that specifies a *base-register*, the base register is implicit in all labels defined by following FIELD directives, until the next MAP directive. These register-relative labels can be quoted in load and store instructions (see *MAP* on page 7-19).

Example

The following example shows how register-relative labels are defined using the MAP and FIELD directives.

```
    MAP     0,r9        ; set {VAR} to the address stored in r9
    FIELD   4           ; increment {VAR} by 4 bytes
Lab FIELD   4           ; set Lab to the address [r9 + 4]
                        ; and then increment {VAR} by 4 bytes
    LDR     r0,Lab      ; equivalent to LDR r0,[r9,#4]
```

 Copyright © 2002-2007 ARM Limited. All rights reserved.

7.3.4 SPACE

The SPACE directive reserves a zeroed block of memory. % is a synonym for SPACE.

Syntax

{*label*} SPACE *expr*

where:

expr evaluates to the number of zeroed bytes to reserve (see *Numeric expressions* on page 3-30).

Usage

Use the ALIGN directive to align any code following a SPACE directive. See *ALIGN* on page 7-63 for more information.

See also:

- *DCB* on page 7-22
- *DCD and DCDU* on page 7-23
- *DCDO* on page 7-24
- *DCW and DCWU* on page 7-29.

Example

```
        AREA    MyData, DATA, READWRITE
data1   SPACE   255     ; defines 255 bytes of zeroed store
```


Copyright © 2002-2007 ARM Limited. All rights reserved.

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 3-30).
- 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 7-63 for more information.

See also:

- *DCD and DCDU* on page 7-23
- *DCQ and DCQU* on page 7-28
- *DCW and DCWU* on page 7-29
- *SPACE* on page 7-21.

Example

Unlike C strings, ARM assembler strings are not null-terminated. You can construct a null-terminated C string using DCB as follows:

```
C_string   DCB  "C_string",0
```

 Copyright © 2002-2007 ARM Limited. All rights reserved.

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 3-30).
- 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 7-22
- *DCW and DCWU* on page 7-29
- *DCQ and DCQU* on page 7-28
- *SPACE* on page 7-21.

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
```


Copyright © 2002-2007 ARM Limited. All rights reserved.

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

{*label*} DCDO *expr*{,*expr*}...

where:

expr is a register-relative expression or label. The base register must be sb.

Usage

Use DCDO to allocate space in memory for static base register relative relocatable addresses.

Example

```
IMPORT  externsym
DCDO    externsym   ; 32-bit word relocated by offset of
                    ; externsym from base of SB section.
```

 Copyright © 2002-2007 ARM Limited. All rights reserved.

7.3.8 DCFD and DCFDU

The DCFD directive allocates memory for word-aligned double-precision floating-point numbers, and defines the initial runtime contents of the memory. Double-precision numbers occupy two words and must be word aligned to be used in arithmetic operations.

DCDFU 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 3-32).

Usage

The assembler inserts up to three bytes of padding before the first defined number, if necessary, to achieve four-byte alignment.

Use DCFDU if you do not require alignment.

The word order used when converting *fpliteral* to internal form is controlled by the floating-point architecture selected. You cannot use DCFD or DCFDU if you select the --fpu none option.

The range for double-precision numbers is:

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

See also *DCFS and DCFSU* on page 7-26.

Examples

```
DCFD    1E308,-4E-100
DCFDU   10000,-.1,3.1E26
```


Copyright © 2002-2007 ARM Limited. All rights reserved.

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.

DCDSU is the same, except that the memory alignment is arbitrary.

Syntax

{*label*} DCFS{U} *fpliteral*{,*fpliteral*}...

where:

fpliteral is a single-precision floating-point literal (see *Floating-point literals* on page 3-32).

Usage

DCFS inserts up to three bytes of padding before the first defined number, if necessary to achieve four-byte alignment.

Use DCFSU if you do not require alignment.

The range for single-precision values is:

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

See also *DCFD and DCFDU* on page 7-25.

Example

```
DCFS    1E3,-4E-9
DCFSU   1.0,-.1,3.1E6
```

 Copyright © 2002-2007 ARM Limited. All rights reserved.

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 3-30).

.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 7-23 and *DCW and DCWU* on page 7-29.

Example macro

```
MACRO           ; this macro translates newinstr Rd,Rm
                ; to the appropriate machine code
newinst     $Rd,$Rm
DCI         0xe16f0f10 :OR: ($Rd:SHL:12) :OR: $Rm
MEND
```

Thumb-2 example

```
DCI.W   0xf3af8000   ; inserts 32-bit NOP, 2-byte aligned.
```


Copyright © 2002-2007 ARM Limited. All rights reserved.

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

{*label*} DCQ{U} {-}*literal*{,{-}*literal*}...

where:

literal is a 64-bit numeric literal (see *Numeric literals* on page 3-31).

The range of numbers permitted is 0 to 2^{64}–1.

In addition to the characters normally permitted in a numeric literal, you can prefix *literal* with a minus sign. In this case, the range of numbers permitted is -2^{63} to –1.

The result of specifying -*n* is the same as the result of specifying 2^{64}–*n*.

Usage

DCQ inserts up to three bytes of padding before the first defined eight-byte block, if necessary, to achieve four-byte alignment.

Use DCQU if you do not require alignment.

See also:

- *DCB* on page 7-22
- *DCD and DCDU* on page 7-23
- *DCW and DCWU* on page 7-29
- *SPACE* on page 7-21.

Example

```
        AREA    MiscData, DATA, READWRITE
data    DCQ     -225,2_101      ; 2_101 means binary 101.
        DCQU    number+4        ; number must already be defined.
```

 Copyright © 2002-2007 ARM Limited. All rights reserved.

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 3-30).

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 7-22
- *DCD and DCDU* on page 7-23
- *DCQ and DCQU* on page 7-28
- *SPACE* on page 7-21.

Example

```
data    DCW     -225,2*number   ; number must already be defined

        DCWU    number+4
```


Copyright © 2002-2007 ARM Limited. All rights reserved.

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

COMMON *symbol*{,*size*{,*alignment*}}

where:

symbol is the symbol name. The symbol name is case-sensitive.

size is the number of bytes to reserve.

alignment is the alignment.

Usage

The linker allocates the required space as zero initialized memory during the link stage.

Example

```
COMMON    xyz,255,4   ; defines 255 bytes of ZI store, word-aligned
```

7.3.14 DATA

The DATA directive is no longer required. It is ignored by the assembler.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

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 7-35
- *IF, ELSE, ENDIF, and ELIF* on page 7-36
- *WHILE and WEND* on page 7-39.

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.

Copyright © 2002-2007 ARM Limited. All rights reserved.

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 2-45 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 7-35 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 3-24). 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.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

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                   ; start macro definition
$label          xmac    $p1,$p2
                ; code
$label.loop1    ; code
                ; code
                BGE     $label.loop1
$label.loop2    ; code
                BL      $p1
                BGT     $label.loop2
                ; code
                ADR     $p2
                ; code
                MEND                    ; end macro definition

 ; macro invocation

abc             xmac    subr1,de        ; invoke macro
                ; code                  ; this is what is
abcloop1        ; code                  ; is produced when
                ; code                  ; the xmac macro is
                BGE     abcloop1        ; expanded
abcloop2        ; code
                BL      subr1
                BGT     abcloop2
                ; code
                ADR     de
                ; code
```


Copyright © 2002-2007 ARM Limited. All rights reserved.

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
```

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
```


Copyright © 2002-2007 ARM Limited. All rights reserved.

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
```


Copyright © 2002-2007 ARM Limited. All rights reserved.

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/or directives. [is a synonym for IF.

The ELSE directive marks the beginning of a sequence of instructions and/or directives that you want to be assembled if the preceding condition fails. | is a synonym for ELSE.

The ENDIF directive marks the end of a sequence of instructions and/or directives that you want to be conditionally assembled.] is a synonym for ENDIF.

The ELIF directive creates a structure equivalent to ELSE IF, without the requirement for nesting or repeating the condition. See *Using ELIF* on page 7-37 for details.

Syntax

```
IF logical-expression      ...    {ELSE      ...}    ENDIF
```

where:

logical-expression

is an expression that evaluates to either {TRUE} or {FALSE}.

See *Relational operators* on page 3-40.

Usage

Use IF with ENDIF, and optionally with ELSE, for sequences of instructions and/or directives that are only to be assembled or acted on under a specified condition.

IF...ENDIF conditions can be nested (see *Nesting directives* on page 7-31).

Copyright © 2002-2007 ARM Limited. All rights reserved.

Using ELIF

Without using ELIF, you can construct a nested set of conditional instructions like this:

```
IF logical-expression
    instructions
ELSE
    IF logical-expression2
        instructions
    ELSE
        IF logical-expression3
            instructions
        ENDIF
    ENDIF
ENDIF
```

A nested structure like this can be nested up to 256 levels deep.

You can write the same structure more simply using ELIF:

```
IF logical-expression
    instructions
ELIF logical-expression2
    instructions
ELIF logical-expression3
    instructions
ENDIF
```

This structure only adds one to the current nesting depth, for the IF...ENDIF pair.

Copyright © 2002-2007 ARM Limited. All rights reserved.

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/directives
        ELSE
            ; alternative set of instructions/directives
        ENDIF
```

Invoking armasm as follows defines NEWVERSION, so the first set of instructions and directives are assembled:

```
armasm --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/directives
        ELSE
            ; alternative set of instructions/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
```

 Copyright © 2002-2007 ARM Limited. All rights reserved.

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

```
WHILE logical-expression
code
WEND
```

where:

logical-expression

is an expression that can evaluate to either {TRUE} or {FALSE} (see *Logical expressions* on page 3-33).

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

```
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
```


Copyright © 2002-2007 ARM Limited. All rights reserved.

7.5 Frame directives

This section describes the following directives:

Correct use of these directives:

- enables the `armlink --callgraph` option to calculate stack usage of assembler functions.

  The following rules are used to determine stack usage:

  — If a function is not marked with `PROC` or `ENDP`, stack usage is unknown.

  — If a function is marked with `PROC` or `ENDP` but with no `FRAME PUSH` or `FRAME POP`, stack usage is assumed to be zero. This means that there is no requirement to manually add `FRAME PUSH 0` or `FRAME POP 0`.

  — If a function is marked with `PROC` or `ENDP` and with `FRAME PUSH n` or `FRAME POP n`, stack usage is assumed to be `n` bytes.

- helps you to avoid errors in function construction, particularly when you are modifying existing code
- enables the assembler to alert you to errors in function construction
- enables backtracing of function calls during debugging
- enables the debugger to profile assembler functions.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

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.

Copyright © 2002-2007 ARM Limited. All rights reserved.

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 7-44).

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 7-43).

Example

```
_fn     FUNCTION        ; CFA (Canonical Frame Address) is value
                        ; of sp on entry to function
        PUSH    {r4,fp,ip,lr,pc}
        FRAME PUSH {r4,fp,ip,lr,pc}
        SUB     sp,sp,#4            ; CFA offset now changed
        FRAME ADDRESS sp,24         ; - so we correct it
        ADD     fp,sp,#20
        FRAME ADDRESS fp,4          ; New base register
        ; code using fp to base call-frame on, instead of sp
```

 Copyright © 2002-2007 ARM Limited. All rights reserved.

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:

```
FRAME POP {reglist}
FRAME POP {reglist},n
FRAME POP n
```

where:

reglist is a list of registers restored to the values they had on entry to the function. There must be at least one register in the list.

n is the number of bytes that the stack pointer moves.

Usage

FRAME POP is equivalent to a FRAME ADDRESS and a FRAME RESTORE directive. You can use it when a single instruction loads registers and alters the stack pointer.

You must use FRAME POP immediately after the instruction it refers to.

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 7-42 and *FRAME RESTORE* on page 7-47.

Copyright © 2002-2007 ARM Limited. All rights reserved.

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 7-42 and *FRAME SAVE* on page 7-49.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

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
```


Copyright © 2002-2007 ARM Limited. All rights reserved.

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.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

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 7-43).

Copyright © 2002-2007 ARM Limited. All rights reserved.

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.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

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 7-44).

Copyright © 2002-2007 ARM Limited. All rights reserved.

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 7-51
- *FUNCTION or PROC* on page 7-53.

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
```

 Copyright © 2002-2007 ARM Limited. All rights reserved.

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 7-50
- *FUNCTION or PROC* on page 7-53.

Copyright © 2002-2007 ARM Limited. All rights reserved.

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.

See also *Controlling exception table generation* on page 3-16.

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 3-16.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

7.5.12 FUNCTION or PROC

The FUNCTION directive marks the start of a function. PROC is a synonym for FUNCTION.

Syntax

label FUNCTION [{*reglist1*} [, {*reglist2*}]]

where:

reglist1 is an optional list of callee saved ARM registers. If *reglist1* is not present, and your debugger checks register usage, it will assume that the AAPCS is in use.

reglist2 is an optional list of callee saved VFP registers.

Usage

Use FUNCTION to mark the start of functions. The assembler uses FUNCTION to identify the start of a function when producing DWARF call frame information for ELF.

FUNCTION sets the canonical frame address to be r13 (sp), and the frame state stack to be empty.

Each FUNCTION directive must have a matching ENDFUNC directive. You must not nest FUNCTION/ENDFUNC pairs, and they must not contain PROC or ENDP directives.

You can use the optional *reglist* parameters to inform the debugger about an alternative procedure call standard, if you are using your own. Not all debuggers support this feature. See your debugger documentation for details.

See also *FRAME ADDRESS* on page 7-42 to *FRAME STATE RESTORE* on page 7-51.

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 7-63 for further information.

Copyright © 2002-2007 ARM Limited. All rights reserved.

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 7-53). ENDP is a synonym for ENDFUNC.

Copyright © 2002-2007 ARM Limited. All rights reserved.

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 7-56
  generates diagnostic information during assembly.
- *OPT* on page 7-57
  sets listing options.
- *TTL and SUBT* on page 7-59
  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 7-56.

Example

```
ASSERT  label1 <= label2    ; Tests if the address
                            ; represented by label1
                            ; is <= the address
                            ; represented by label2.
```


Copyright © 2002-2007 ARM Limited. All rights reserved.

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*

where:

numeric-expression

is a numeric expression that is evaluated during assembly. If the expression evaluates to zero:

- no action is taken during pass one
- *string-expression* is printed during pass two.

If the expression does not evaluate to zero, *string-expression* is printed as an error message and the assembly fails.

string-expression

is an expression that evaluates to a string.

Usage

INFO provides a flexible means of creating custom error messages. See *Numeric expressions* on page 3-30 and *String expressions* on page 3-29 for additional information on numeric and string expressions.

See also *ASSERT* on page 7-55.

Examples

```
INFO    0, "Version 1.0"

IF endofdata <= label1
    INFO    4, "Data overrun at label1"
ENDIF
```


Copyright © 2002-2007 ARM Limited. All rights reserved.

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 7-2 OPT directive settings

OPT n	Effect
1	Turns on normal listing.
2	Turns off normal listing.
4	Page throw. Issues an immediate form feed and starts a new page.
8	Resets the line number counter to zero.
16	Turns on listing for SET, GBL and LCL directives.
32	Turns off listing for SET, GBL and LCL directives.
64	Turns on listing of macro expansions.
128	Turns off listing of macro expansions.
256	Turns on listing of macro invocations.
512	Turns off listing of macro invocations.
1024	Turns on the first pass listing.
2048	Turns off the first pass listing.
4096	Turns on listing of conditional directives.
8192	Turns off listing of conditional directives.
16384	Turns on listing of MEND directives.
32768	Turns off listing of MEND directives.

Copyright © 2002-2007 ARM Limited. All rights reserved.

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 3-13 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
```

 Copyright © 2002-2007 ARM Limited. All rights reserved.

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.
```


Copyright © 2002-2007 ARM Limited. All rights reserved.

7.7 Instruction set and syntax selection directives

This section describes the following directives:

- *ARM, THUMB, THUMBX, CODE16 and CODE32* on page 7-61.

Copyright © 2002-2007 ARM Limited. All rights reserved.

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. If necessary, they also insert up to three bytes of padding to align to the next word boundary.

In this mode, the assembler accepts both the latest assembly language and the earlier version.

Syntax

```
ARM
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 new syntax.
- THUMBX must precede Thumb-2EE code written in new syntax.
- CODE16 must precede Thumb code written in old Thumb syntax .

These directives do not assemble to instructions that changes the state. They only instruct the assembler to assemble ARM, Thumb-2, Thumb-2EE, or Thumb instructions as appropriate, and insert padding if necessary.

Example

This example shows how ARM and CODE16 can be used to branch from ARM to 16-bit Thumb instructions.

```
        AREA 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
```


Copyright © 2002-2007 ARM Limited. All rights reserved.

7.8 Miscellaneous directives

This section describes the following directives:

Copyright © 2002-2007 ARM Limited. All rights reserved.

7.8.1 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* * *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.

Copyright © 2002-2007 ARM Limited. All rights reserved.

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 7-16) 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 7-66 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
```

 Copyright © 2002-2007 ARM Limited. All rights reserved.

```
        ALIGN               ; ensures that subroutine1 addresses
subroutine1                 ; the following instruction.
        MOV r5,#0x5
```


Copyright © 2002-2007 ARM Limited. All rights reserved.

7.8.2 AREA

The AREA directive instructs the assembler to assemble a new code or data section. Sections are independent, named, indivisible chunks of code or data that are manipulated by the linker. See *ELF sections and the AREA directive* on page 2-14 for more information.

Syntax

AREA *sectionname*{,*attr*}{,*attr*}...

where:

sectionname is the name that the section is to be given.

You can choose any name for your sections. However, names starting with a digit must be enclosed in bars or a missing section name error is generated. For example, |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 7-63.

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.

 Copyright © 2002-2007 ARM Limited. All rights reserved.

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 *RealView Compilation Tools Linker and Utilities 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.

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.

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

Copyright © 2002-2007 ARM Limited. All rights reserved.

(see Chapter 3 *Using the Basic Linker Functionality* in the *RealView Compilation Tools Linker and Utilities Guide*).

`READONLY` Indicates that this section should not be written to. This is the default for Code areas.

`READWRITE` Indicates that this section can be read from and written to. This is the default for Data areas.

Usage

Use the `AREA` directive to subdivide your source file into ELF sections. You can use the same name in more than one `AREA` directive. All areas with the same name are placed in the same ELF section. Only the attributes of the first `AREA` directive of a particular name are applied.

You should normally use separate ELF sections for code and data. Large programs can usually be conveniently divided into several code sections. Large independent data sets are also usually best placed in separate sections.

The scope of local labels is defined by `AREA` directives, optionally subdivided by `ROUT` directives (see *Local labels* on page 3-27 and *ROUT* on page 7-81).

There must be at least one `AREA` directive for an assembly.

Example

The following example defines a read-only code section named `Example`.

```
AREA    Example,CODE,READONLY   ; An example code section.
        ; code
```

 Copyright © 2002-2007 ARM Limited. All rights reserved.

7.8.3 END

The END directive informs the assembler that it has reached the end of a source file.

Syntax

```
END
```

Usage

Every assembly language source file must end with END on a line by itself.

If the source file has been included in a parent file by a GET directive, the assembler returns to the parent file and continues assembly at the first line following the GET directive. See *GET or INCLUDE* on page 7-74 for more information.

If END is reached in the top-level source file during the first pass without any errors, the second pass begins.

If END is reached in the top-level source file during the second pass, the assembler finishes the assembly and writes the appropriate output.

7.8.4 ENTRY

The ENTRY directive declares an entry point to a program.

Syntax

```
ENTRY
```

Usage

You must specify at least one ENTRY point for a program. If no ENTRY exists, a warning is generated at link time.

You must not use more than one ENTRY directive in a single source file. Not every source file has to have an ENTRY directive. If more than one ENTRY exists in a single source file, an error message is generated at assembly time.

Example

```
AREA    ARMex, CODE, READONLY
ENTRY                ; Entry point for the application
```


Copyright © 2002-2007 ARM Limited. All rights reserved.

7.8.5 EQU

The EQU directive gives a symbolic name to a numeric constant, a register-relative value or a program-relative value. * is a synonym for EQU.

Syntax

name EQU *expr*{, *type*}

where:

name is the symbolic name to assign to the value.

expr is a register-relative address, a program-relative address, an absolute address, or a 32-bit integer constant.

type is optional. *type* can be any one of:

- ARM
- THUMB
- CODE32
- CODE16
- DATA

You can use *type* only if *expr* is an absolute address. If *name* is exported, the *name* entry in the symbol table in the object file will be marked as ARM, THUMB, CODE32, CODE16, or DATA, according to *type*. This can be used by the linker.

Usage

Use EQU to define constants. This is similar to the use of **#define** to define a constant in C.

See *KEEP* on page 7-78 and *EXPORT or GLOBAL* on page 7-71 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
```

 Copyright © 2002-2007 ARM Limited. All rights reserved.

7.8.6 EXPORT or GLOBAL

The EXPORT directive declares a symbol that can be used by the linker to resolve symbol references in separate object and library files. GLOBAL is a synonym for EXPORT.

Syntax

```
EXPORT {[WEAK]}

EXPORT symbol {[attr}]}

EXPORT symbol [WEAK{,attr}]
```

where:

symbol is the symbol name to export. The symbol name is case-sensitive. If *symbol* is omitted, all symbols are exported.

WEAK *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 *symbol* is visible to other components when the source is linked into a dynamic component.
- PROTECTED *symbol* is visible to other components when the source is linked into a dynamic component, but must not be redefined by other components.
- HIDDEN *symbol* is not visible to other components when the source is linked into a dynamic component.

Copyright © 2002-2007 ARM Limited. All rights reserved.

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 7-75.

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.
```

 Copyright © 2002-2007 ARM Limited. All rights reserved.

7.8.7 EXPORTAS

The EXPORTAS directive enables you to export a symbol to the object file, corresponding to a different symbol in the source file.

Syntax

EXPORTAS *symbol1*, *symbol2*

where:

symbol1 is the symbol name in the source file. *symbol1* must have been defined already. It can be any symbol, including an area name, a label, or a constant.

symbol2 is the symbol name you want to appear in the object file.

The symbol names are case-sensitive.

Usage

Use EXPORTAS to change a symbol in the object file without having to change every instance in the source file.

See also *EXPORT or GLOBAL* on page 7-71.

Examples

```
    AREA data1, DATA       ;; starts a new area data1
    AREA data2, DATA       ;; starts a new area data2
    EXPORTAS data2, data1  ;; the section symbol referred to as data2 will
                           ;; appear in the object file string table as data1.

one EQU  2
    EXPORTAS one, two
    EXPORT one             ;; the symbol 'two' will appear in the object
                           ;; file's symbol table with the value 2.
```


Copyright © 2002-2007 ARM Limited. All rights reserved.

7.8.8 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 7-77).

Example

```
AREA    Example, CODE, READONLY
GET     file1.s              ; includes file1 if it exists
                             ; in the current place.
GET     c:\project\file2.s   ; includes file2
GET     c:\Program files\file3.s  ; space is permitted
```

 Copyright © 2002-2007 ARM Limited. All rights reserved.

7.8.9 IMPORT and EXTERN

These directives provide the assembler with a name that is not defined in the current assembly.

`IMPORT` imports the name whether or not it is referred to in the current assembly.

`EXTERN` imports the name only if it is referred to in the current assembly.

Syntax

`IMPORT` *symbol* {[*attr*}]}

`IMPORT` *symbol* [`WEAK`{,*attr*}]

`EXTERN` *symbol* {[*attr*}]}

`EXTERN` *symbol* [`WEAK`{,*attr*}]

where:

symbol is a symbol name defined in a separately assembled source file, object file, or library. The symbol name is case-sensitive.

`WEAK` prevents the linker generating an error message if the symbol is not defined elsewhere. It also prevents the linker searching libraries that are not already included.

attr can be any one of:

- `DYNAMIC` *symbol* is visible to other components when the source is linked into a dynamic component.
- `PROTECTED` *symbol* is visible to other components when the source is linked into a dynamic component, but must not be redefined by other components.
- `HIDDEN` *symbol* is not visible to other components when the source is linked into a dynamic component.

Copyright © 2002-2007 ARM Limited. All rights reserved.

Usage

The name is resolved at link time to a symbol defined in a separate object file. The symbol is treated as a program address. If [WEAK] is not specified, the linker generates an error if no corresponding symbol is found at link time.

If [WEAK] is specified and no corresponding symbol is found at link time:

- If the reference is the destination of a B or BL instruction, the value of the symbol is taken as the address of the following instruction. This makes the B or BL instruction effectively a NOP.
- Otherwise, the value of the symbol is taken as zero.

Example

This example tests to see if the C++ library has been linked, and branches conditionally on the result.

```
AREA    Example, CODE, READONLY
EXTERN  __CPP_INITIALIZE[WEAK]  ; If C++ library linked, gets the address of
                                ; __CPP_INITIALIZE function.
LDR     r0,=__CPP_INITIALIZE    ; If not linked, address is zeroed.
CMP     r0,#0                   ; Test if zero.
BEQ     nocplusplus             ; Branch on the result.
```

 Copyright © 2002-2007 ARM Limited. All rights reserved.

7.8.10 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
```


Copyright © 2002-2007 ARM Limited. All rights reserved.

7.8.11 KEEP

The KEEP directive instructs the assembler to retain local symbols in the symbol table in the object file.

Syntax

KEEP {*symbol*}

where:

symbol is the name of the local symbol to keep. If *symbol* is not specified, all local symbols are kept except register-relative symbols.

Usage

By default, the only symbols that the assembler describes in its output object file are:

- exported symbols
- symbols that are relocated against.

Use KEEP to preserve local symbols that can be used to help debugging. Kept symbols appear in the ARM debuggers and in linker map files.

KEEP cannot preserve register-relative symbols (see *MAP* on page 7-19).

Example

```
label   ADC     r2,r3,r4
        KEEP    label       ; makes label available to debuggers
        ADD     r2,r2,r5
```


Copyright © 2002-2007 ARM Limited. All rights reserved.

7.8.12 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.

7.8.13 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.

Copyright © 2002-2007 ARM Limited. All rights reserved.

7.8.14 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}
```

 Copyright © 2002-2007 ARM Limited. All rights reserved.

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
```

7.8.15 ROUT

The ROUT directive marks the boundaries of the scope of local labels (see *Local labels* on page 3-27).

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 7-66).

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
```


Copyright © 2002-2007 ARM Limited. All rights reserved.

 Copyright © 2002-2007 ARM Limited. All rights reserved.