RealView® Developer Kit

Version 2.2

Assembler Guide

ARM®

Copyright © 2005 ARM Limited. All rights reserved.
ARM DUI 0283B

RealView Developer Kit
Assembler Guide

Copyright © 2005 ARM Limited. All rights reserved.

Release Information

The following changes have been made to this book.

Change History

Date	Issue	Change
April 2005	B	Release for RVDK v2.2

Proprietary Notice

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

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

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

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

Confidentiality Status

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

Product Status

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

Web Address

`http://www.arm.com`

Contents
RealView Developer Kit Assembler Guide

Copyright © 2005 ARM Limited. All rights reserved.

 Copyright © 2005 ARM Limited. All rights reserved.

Preface

This preface introduces the *RealView® Developer Kit (RVDK) v2.2 Assembler Guide*. It contains the following sections:

Copyright © 2005 ARM Limited. All rights reserved.

About this book

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

Intended audience

This book is written for all developers who are producing applications using RVDK. It assumes that you are an experienced software developer and that you are familiar with the ARM development tools.

Using this book

This book is organized into the following chapters:

Chapter 1 ***Introduction***

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

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

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

Chapter 3 ***Assembler Reference***

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

Chapter 4 ***ARM and Thumb instructions***

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

Chapter 5 ***Vector Floating-point Programming***

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

Chapter 6 ***Directives Reference***

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

Copyright © 2005 ARM Limited. All rights reserved.

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

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

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.

Copyright © 2005 ARM Limited. All rights reserved.

ARM publications

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

- *RealView Developer Kit v2.2 Debugger User Guide* (ARM DUI 0281)
- *RealView Developer Kit v2.2 Compiler and Libraries Guide* (ARM DUI 0282)
- *RealView Developer Kit v2.2 Command Line Reference* (ARM DUI 0284)
- *RealView Developer Kit v2.2 Linker and Utilities Guide* (ARM DUI 0285)
- *RealView Developer Kit v2.2 Project Management Guide* (ARM DUI 0291).

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

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

Other publications

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

Copyright © 2005 ARM Limited. All rights reserved.

Feedback

ARM Limited welcomes feedback on both RVDK and the documentation.

Feedback on RealView Developer Kit

If you have any problems with RVDK, 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 © 2005 ARM Limited. All rights reserved.

 Copyright © 2005 ARM Limited. All rights reserved.

Chapter 1
Introduction

This chapter introduces the assemblers provided with *RealView® Developer Kit (RVDK) v2.2* (RVDK). It contains the following section:

- *About the RealView Developer Kit assemblers* on page 1-2.

Copyright © 2005 ARM Limited. All rights reserved.

1.1 About the RealView Developer Kit assemblers

RVDK has:

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

The language that these assemblers take as input is basically the same. However, there are limitations on what features of the language you can use in the inline assemblers.

Note

Support for C++ may be obtained as an upgrade on certain versions of RVDK, by means of an appropriate FLEX*lm* license.

1.1.1 ARM Assembly Language

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

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

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

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

Note

None of the processors supported by this toolkit supports Thumb-2.

1.1.2 Using the examples

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

 Copyright © 2005 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 © 2005 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, or VFP instruction sets. For this information see:

- Chapter 4 *ARM and Thumb instructions*
- Chapter 5 *Vector Floating-point Programming*.

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

For the convenience of programmers who are already familiar with ARM and Thumb assembly languages, this chapter includes a section outlining the differences between the latest version of the ARM assembly language and earlier versions of the ARM and Thumb assembly languages, see *Assembly Language changes* on page 2-47.

Note

None of the processors supported by this toolkit supports Thumb-2.

2.1.1 Code examples

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

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

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

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

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

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

See *RealView Developer Kit v2.2 Linker and Utilities Guide* for details on armlink and fromelf.

 Copyright © 2005 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.

2.2.1 Architecture versions

The information and examples in this book assume that you are using a processor that implements ARM architecture v4 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, and Thumb-2 instruction sets

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

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

For more information, see *Instruction set overview* on page 2-6.

2.2.3 ARM and Thumb state

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

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

Each instruction set includes instructions to change processor state.

You must also switch the assembler mode to produce the correct opcodes using `ARM` (`CODE32`), `THUMB`, or `CODE16` directives. See *Instruction set and syntax selection directives* on page 6-58 for details.

ARM processors always start executing code in ARM state.

Copyright © 2005 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).

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

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.

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:

- *30 general-purpose, 32-bit registers* on page 2-5
- *The Program Counter (PC)* on page 2-5
- *The Current Program Status Register (CPSR)* on page 2-5
- *Saved Program Status Registers (SPSRs)* on page 2-6.

 Copyright © 2005 ARM Limited. All rights reserved.

30 general-purpose, 32-bit registers

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

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

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

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

The *Program Counter* (PC)

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

```
MOV  pc,lr
```

During execution, r15 (pc) does not contain the address of the currently executing instruction. The address of the currently executing instruction is typically pc–8 for ARM, or pc–4 for Thumb.

The *Current Program Status Register* (CPSR)

The CPSR holds:

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

The ALU status flags in the CPSR are used to determine whether conditional instructions are executed or not. See *Conditional execution* on page 2-16 for more information.

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

On ARMv5TE, the CPSR also holds the Q flag (see *The ALU status flags* on page 2-17).

Copyright © 2005 ARM Limited. All rights reserved.

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.

2.2.6 Instruction set overview

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

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

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

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

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

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

- *Branch instructions*
- *Data processing instructions* on page 2-7
- *Single register load and store instructions* on page 2-7
- *Multiple register load and store instructions* on page 2-7
- *Status register access instructions* on page 2-7
- *Coprocessor instructions* on page 2-7.

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.

 Copyright © 2005 ARM Limited. All rights reserved.

Data processing instructions

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

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

Single register load and store instructions

These instructions load or store the value of a single register from or to memory. They can load or store a 32-bit word, a 16-bit halfword, or an 8-bit unsigned byte. Byte and halfword loads can either be sign extended or zero extended to fill the 32-bit register.

Multiple register load and store instructions

These instructions load or store any subset of the general-purpose registers from or to memory. See *Load and store multiple register instructions* on page 2-36 for a detailed description of these instructions.

Status register access instructions

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

Coprocessor instructions

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

Copyright © 2005 ARM Limited. All rights reserved.

2.2.7 Instruction capabilities

This section contains the following subsections:

- *Conditional execution*
- *Register access*
- *Access to the inline barrel shifter* on page 2-9.

Conditional execution

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

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

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

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

See *Conditional execution* on page 2-16 for a detailed description.

Register access

In ARM state, all instructions can access r0 to r14, and most can also access r15 (pc). The `MRS` and `MSR` instructions can move the contents of the CPSR and SPSRs to a general-purpose register, where they can be manipulated by normal data processing operations. See *MRS* on page 4-95 and *MSR* on page 4-96 for more information.

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

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

 Copyright © 2005 ARM Limited. All rights reserved.

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 2-24 for more information on using the barrel shifter to generate constants.

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

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

Copyright © 2005 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 need 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.

RVDK uses an ELF proprietary file format called *ARM Toolkit Proprietary ELF* (ATPE). The file format for each version of RVDK is restricted to the proprietary ATPE format for the permitted device. This is referred to as *ATPE_Custom*.

2.3.1 Layout of assembly language source files

The general form of source lines in assembly language is:

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

Note

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

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

Case rules

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

Line length

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

 Copyright © 2005 ARM Limited. All rights reserved.

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

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

Comments

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

Copyright © 2005 ARM Limited. All rights reserved.

Constants

Constants can be numeric, Boolean, character, or string:

Numbers Numeric constants are accepted in the following forms:

- decimal, for example, `123`
- hexadecimal, for example, `0x7B`
- *n_xxx* where:
  - *n* is a base between 2 and 9
  - *xxx* is a number in that base.

Boolean The Boolean constants `TRUE` and `FALSE` must be written as `{TRUE}` and `{FALSE}`.

Characters Character constants consist of opening and closing single quotes, enclosing either a single character or an escaped character, using the standard C escape characters.

Strings Strings consist of opening and closing double quotes, enclosing characters and spaces. If double quotes or dollar signs are used within a string as literal text characters, they must be represented by a pair of the appropriate character. For example, you must use `$$` if you require a single `$` in the string. The standard C escape sequences can be used within string constants.

 Copyright © 2005 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 *`install_directory`*`\RVDK\Examples\...\asm` directory. See *Code examples* on page 2-2 for instructions on how to assemble, link, and execute the example.

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

Example 2-1

```
        AREA     ARMex, CODE, READONLY
                                ; Name this block of code ARMex
        ENTRY                   ; Mark first instruction to execute
start
        MOV      r0, #10        ; Set up parameters
        MOV      r1, #3
        ADD      r0, r0, r1     ; r0 = r0 + r1
stop
        MOV      r0, #0x18      ; angel_SWIreason_ReportException
        LDR      r1, =0x20026   ; ADP_Stopped_ApplicationExit
        SWI      0x123456       ; ARM semihosting SWI

        END                     ; Mark end of file
```

ELF sections and the AREA directive

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

The output of an assembly or compilation can include:

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

The linker places each section in a program image according to section placement rules. Sections that are adjacent in source files are not necessarily adjacent in the application image. See the chapter describing basic linker functionality in *RealView Developer Kit v2.2 Linker and Utilities Guide* for more information on how the linker places sections.

RVDK uses an ELF proprietary file format called *ARM Toolkit Proprietary ELF* (ATPE). The file format for each version of RVDK is restricted to the proprietary ATPE format for the permitted device. This is referred to as *ATPE_Custom*.

Copyright © 2005 ARM Limited. All rights reserved.

In a source file, the start of a section is marked by the AREA directive. This directive names the section and sets its attributes. The attributes are placed after the name, separated by commas. See *AREA* on page 6-66 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 2-13 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 2-13 begins executing at the label `start`, where it loads the decimal values 10 and 3 into registers r0 and r1. These registers are added together and the result placed in r0.

Application termination

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

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

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

The END directive

This directive instructs the assembler to stop processing this source file. Every assembly language source module must finish with an END directive on a line by itself.

 Copyright © 2005 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 or register-relative expression. See *B, BL, BX, BLX, and BXJ* on page 4-79 for more information.

The BL instruction:

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

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

Note

Calls between separately assembled or compiled modules must comply with the restrictions and conventions defined by the procedure call standard.

Example 2-2 shows a subroutine that adds the values of its two parameters and returns a result in r0. It is supplied as subrout.s in the directory *install_directory*\RVDK\Examples\...\asm. 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
        SWI     0x123456            ; ARM semihosting SWI

doadd   ADD     r0, r0, r1          ; Subroutine code
        BX      lr                  ; Return from subroutine
        END                         ; Mark end of file
```


Copyright © 2005 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 *Current Program Status Register* (CPSR) according to the result of the operation.

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

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

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

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

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

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

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

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

 Copyright © 2005 ARM Limited. All rights reserved.

2.4.1 The ALU status flags

The CPSR contains the following ALU status flags:

N Set when the result of the operation was Negative.

Z Set when the result of the operation was Zero.

C Set when the operation resulted in a Carry.

V Set when the operation caused oVerflow.

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

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

Copyright © 2005 ARM Limited. All rights reserved.

2.4.2 Conditional execution in ARM state

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

The relation of condition code suffixes to the N, Z, C and V flags is also shown in Table 2-2.

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.

Copyright © 2005 ARM Limited. All rights reserved.

2.4.3 Example

```
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 © 2005 ARM Limited. All rights reserved.

2.4.4 Using conditional execution in ARM state

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

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

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

2.4.5 Example of the use of conditional execution

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

In C the algorithm can be expressed as:

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

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

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

 Copyright © 2005 ARM Limited. All rights reserved.

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

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

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

In addition to improving code size, this code executes faster in most cases. Table 2-3 and Table 2-4 on page 2-22 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

Copyright © 2005 ARM Limited. All rights reserved.

Table 2-4 All instructions conditional

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

Converting to pre-Thumb-2 Thumb code

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

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

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

Branch prediction and caches

To optimize code for execution speed you need 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 © 2005 ARM Limited. All rights reserved.

2.4.6 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 4-70), or when overflow has occurred in certain multiply instructions (see *SMULxy and SMLAxy* on page 4-63 and *SMULWy and SMLAWy* on page 4-65).

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 needing to check the flag after each instruction.

To clear the Q flag, use an `MSR` instruction (see *MSR* on page 4-96).

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 4-95).

Copyright © 2005 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 ARM instruction without performing a data load from memory. This is because ARM instructions are only 32 bits long. Thumb instructions have a similar limitation.

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

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

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*
- how to use the `LDR` pseudo-instructions to load any 32-bit constant, see *Loading with LDR Rd, =const* on page 2-27.

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 on page 2-25 shows the range of values that can be loaded in a single instruction in ARM. *Thumb-2 immediate constants* on page 2-26 shows the range of values that can be loaded in a single instruction in Thumb2.

You do not need 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.`

 Copyright © 2005 ARM Limited. All rights reserved.

ARM state immediate constants

In ARM state:

- `MOV` can load any eight-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`.

Table 2-5 shows the range of values that this provides.

Table 2-5 ARM state immediate constants

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)	b
`efgh000000000000000000000000abcd`	(bit pattern)	-	-	(bit pattern)	b
`gh000000000000000000000000abcdef`	(bit pattern)	-	-	(bit pattern)	b
`00000000000000000000abcdefghijkl`	0-4095	1	`0-0xFFF`	-	c

a. The `MVN` values are not available directly as operands in other instructions.
b. These values are available in ARM state only. All the other values in this table are also available in Thumb-2.
c. These values are only available in ARMv6T2 and above, except those that appear elsewhere in the table. They are not available directly as operands in other instructions.

Copyright © 2005 ARM Limited. All rights reserved.

Thumb-2 immediate constants

In Thumb state, in ARMv6T2 and above:

- the 32-bit `MOV` instruction can load:
  - any eight-bit constant value, giving a range of `0x0-0xFF` (0-255)
  - any eight-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 also load any 12-bit number, giving a range of `0x0-0xFFF` (0-4095). `MVN` cannot load the bitwise complement of these values.

Table 2-5 on page 2-25 shows the range of values that this provides.

Table 2-6 Thumb state immediate constants

Binary	Decimal	Step	Hexadecimal	MVN value [a]	Note
`00000000000000000000000abcdefgh`	0-255	1	`0-0xFF`	–1 to –256	-
`00000000000000000000000abcdefgh0`	0-510	2	`0-0x1FE`	–2 to –512	-
`00000000000000000000000abcdefgh00`	0-1020	4	`0-0x3FC`	–4to –1024	-
	...	...	...	...	-
`0abcdefgh000000000000000000000000`	0-255 x 2^{23}	2^{23}	`0-0x7F800000`	1-256 x -2^{24}	-
`abcdefgh000000000000000000000000`	0-255 x 2^{24}	2^{24}	`0-0xFF000000`	1-256 x -2^{24}	-
`abcdefghabcdefghabcdefghabcdefgh`	(bit pattern)	-	`0xXYXYXYXY`	-	-
`00000000abcdefgh00000000abcdefgh`	(bit pattern)	-	`0x00XY00XY`	`0xFFXYFFXY`	-
`abcdefgh00000000abcdefgh00000000`	(bit pattern)	-	`0xXY00XY00`	`0xXYFFXYFF`	-
`0000000000000000000abcdefghijkl`	0-4095	1	`0-0xFFF`	-	a

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

Copyright © 2005 ARM Limited. All rights reserved.

2.5.2 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* for more information.

See *LDR pseudo-instruction* on page 4-105 for a description of the syntax of the LDR pseudo-instruction.

Placing literal pools

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

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

- less than 4KB in ARM state, but can be in either direction
- forward and less than 1KB in Thumb state.

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

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

Copyright © 2005 ARM Limited. All rights reserved.

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 6-16 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 2-3 shows how this works in practice. It is supplied as loadcon.s in the *install_directory*\RVDK\Examples\...\asm directory. The instructions listed as comments are the ARM instructions that are generated by the assembler. See *Code examples* on page 2-2 for instructions on how to assemble, link, and execute the example.

Example 2-3

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

 Copyright © 2005 ARM Limited. All rights reserved.

2.6 Loading addresses into registers

It is often necessary to load an address into a register. You might need 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 2-33.

2.6.1 Direct loading with ADR and ADRL

The `ADR` and `ADRL` pseudo-instructions enable you to generate an address, within a certain range, without performing a data load. `ADR` and `ADRL` accept a program-relative expression, which is a label with an optional offset, where the address of the label is relative to the current PC.

Note

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

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

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

Copyright © 2005 ARM Limited. All rights reserved.

ADR

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

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

The available range depends on the instruction set in use:

ARM 255 bytes to a byte or halfword-aligned address 1020 bytes to a word-aligned address.

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

32-bit Thumb-2 4095 bytes to a byte, halfword, or word-aligned address.

See *ADR pseudo-instruction* on page 4-101 for details.

ADRL

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

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

The available range depends on the instruction set in use:

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

16-bit Thumb ADRL is not available.

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

See *Loading addresses with LDR Rd, = label* on page 2-33 for information on loading addresses that are outside the range of the ADRL pseudo-instruction.

Implementing a jump table with ADR

Example 2-4 on page 2-31 shows ARM code that implements a jump table. It is supplied as jump.s in the *install_directory*\RVDK\Examples\...\asm directory. See *Code examples* on page 2-2 for instructions on how to assemble, link, and execute the example.

The ADR pseudo-instruction loads the address of the jump table.

 Copyright © 2005 ARM Limited. All rights reserved.

In the example, the function `arithfunc` takes three arguments and returns a result in r0. The first argument determines which operation is 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 this example it assigns the value 2 to *num*. When *num* is used elsewhere in the code, the value 2 is substituted. Using `EQU` in this way is similar to using `#define` to define a constant in C.

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

`LDR` The `LDR pc,[r3,r0,LSL#2]` instruction loads the address of the required clause of the jump table into the PC. It:

- multiplies the clause number in r0 by 4 to give a word offset
- adds the result to the address of the jump table
- loads the contents of the combined address into the PC.

Example 2-4 ARM code jump table

```
        AREA    Jump, CODE, READONLY     ; Name this block of code
        CODE32                           ; 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
        SWI     0x123456                 ; ARM semihosting SWI
arithfunc                                ; Label the function
        CMP     r0, #num                 ; Treat function code as unsigned
integer
        BXHS    lr                       ; If code is >= num then simply return
        ADR     r3, JumpTable            ; Load address of jump table
        LDR     pc, [r3,r0,LSL#2]        ; Jump to the appropriate routine
JumpTable
```


Copyright © 2005 ARM Limited. All rights reserved.

```
        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
```

 Copyright © 2005 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 2-27. 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 2-27 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 2-5 shows how this works. It is supplied as `ldrlabel.s` in the *install_directory*`\RVDK\Examples\...\asm` directory. 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 that are generated by the assembler.

Example 2-5

```
        AREA    LDRlabel, CODE,READONLY
        ENTRY                           ; Mark first instruction to execute
start
        BL      func1                   ; Branch to first subroutine
        BL      func2                   ; Branch to second subroutine
stop    MOV     r0, #0x18               ; angel_SWIreason_ReportException
        LDR     r1, =0x20026            ; ADP_Stopped_ApplicationExit
        SWI     0x123456                ; ARM semihosting SWI
func1
        LDR     r0, =start              ; => LDR R0,[PC, #offset into
                                        ; Literal Pool 1]
        LDR     r1, =Darea + 12         ; => LDR R1,[PC, #offset into
```


Copyright © 2005 ARM Limited. All rights reserved.

```
                                        ; Literal Pool 1]
        LDR     r2, =Darea + 6000       ; => LDR R2, [PC, #offset into
                                        ; Literal Pool 1]
        MOV     pc,lr                   ; Return
        LTORG                           ; Literal Pool 1
func2
        LDR     r3, =Darea + 6000       ; => LDR r3, [PC, #offset into
                                        ; Literal Pool 1]
                                        ; (sharing with previous literal)
        ; LDR   r4, =Darea + 6004       ; If uncommented produces an error
                                        ; as Literal Pool 2 is out of range
        BX      lr                      ; Return
Darea   SPACE   8000                    ; Starting at the current location,
                                        ; clears a 8000 byte area of memory
                                        ; to zero
        END                             ; Literal Pool 2 is out of range of
                                        ; the LDR instructions above
```

 Copyright © 2005 ARM Limited. All rights reserved.

An LDR Rd, =label example: string copying

Example 2-6 shows an ARM code routine that overwrites one string with another string. It uses the LDR pseudo-instruction to load the addresses of the two strings from a data section. The following are particularly significant:

DCB — The DCB directive defines one or more bytes of store. In addition to integer values, DCB accepts quoted strings. Each character of the string is placed in a consecutive byte. See *DCB* on page 6-20 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 2-6 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
        SWI     0x123456                ; ARM semihosting 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 © 2005 ARM Limited. All rights reserved.

2.7 Load and store multiple register instructions

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

Multiple register transfer instructions provide an efficient way of moving the contents of several registers to and from memory. They are most often used for block copy and for stack operations at subroutine entry and exit. The advantages of using a multiple register transfer instruction instead of a series of single data transfer instructions include:

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

Note

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

You can use the `--checkreglist` assembler command line option to check that registers in register lists are specified in increasing order. See *Command syntax* on page 3-2 for more information.

 Copyright © 2005 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 either:
  - be updated to point to the next block of data in memory
  - be 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 and Thumb-2 instructions, any or all of r0-r7, and optionally r14 (`PUSH` only) or r15 (`POP` only).

Copyright © 2005 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 load and store multiple 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 2-7 for a list of stack-oriented suffixes.

Table 2-7 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 © 2005 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 safely be made 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

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

Copyright © 2005 ARM Limited. All rights reserved.

2.7.3 Block copy with LDM and STM

Example 2-7 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 *`install_directory`*`\RVDK\Examples\...\asm` directory. See *Code examples* on page 2-2 for instructions on how to assemble, link, and execute the example.

Example 2-7 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 to
call
start
            LDR     r0, =src                 ; r0 = pointer to source block
            LDR     r1, =dst                 ; r1 = pointer to destination block
            MOV     r2, #num                 ; r2 = number of words to copy
wordcopy    LDR     r3, [r0], #4             ; load a word from the source and
            STR     r3, [r1], #4             ; store it to the destination
            SUBS    r2, r2, #1               ; decrement the counter
            BNE     wordcopy                 ; ... copy more
stop        MOV     r0, #0x18                ; angel_SWIreason_ReportException
            LDR     r1, =0x20026             ; ADP_Stopped_ApplicationExit
            SWI     0x123456                 ; ARM semihosting SWI

            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 2-8 on page 2-41 lists the block copy module rewritten to use `LDM` and `STM` for copying.

 Copyright © 2005 ARM Limited. All rights reserved.

Example 2-8 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 to
call
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
            SWI     0x123456                 ; ARM semihosting 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 © 2005 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 6-30 for more details.

2.8.1 Test-and-branch macro example

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*. The macro prototype statement defines the name (TestAndBranch) you use to invoke the macro. It also defines *parameters* ($label, $dest, $reg, and $cc). You must give values to the parameters when you invoke the macro. The assembler substitutes the values you give into the code.

This macro can be invoked as follows:

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

After substitution this becomes:

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

 Copyright © 2005 ARM Limited. All rights reserved.

2.8.2 Unsigned integer division macro example

Example 2-9 shows a macro that performs an unsigned integer division. It takes four parameters:

`$Bot` The register that holds the divisor.

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

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

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

Example 2-9

```
        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
```


Copyright © 2005 ARM Limited. All rights reserved.

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 2-11 for more information.

Example 2-10 shows the code that this macro produces if it is invoked as follows:

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

Example 2-10

```
        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 © 2005 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* and *name*`@@`*ver* for a default *ver* of *name*. The version symbols must be enclosed in vertical bars. For example, to define a default version:

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

To define a non default version:

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

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

Copyright © 2005 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 6-38 for details of these directives.

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

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

 Copyright © 2005 ARM Limited. All rights reserved.

2.11 Assembly Language changes

Table 2-8 shows the main differences between the RVDK v2.2 and earlier versions of the ARM assembly language. The old ARM syntax is still accepted by the assembler.

Table 2-8 Changes from earlier ARM assembly language

Change	Old ARM syntax	Preferred syntax
The default addressing mode for LDM and STM is now IA	LDMIA, STMIA	LDM, STM
You can now 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 now 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 now 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 TE ADDEQ r1, r2, r3 LDRNE r1, [r2, r3]

In addition, some flexibility is permitted that was not permitted in previous assemblers. Table 2-9 shows the relaxations.

Table 2-9 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 now write source code for Thumb processors using the ARM assembly language.

Copyright © 2005 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 2-10 shows the main differences between Thumb assembly language and ARM assembly language. The old Thumb syntax is accepted by the assembler only if it is preceded by a `CODE16` directive, or the source file is assembled with the `--16` command line option.

Table 2-10 Differences between ARM and Thumb assembly language

Change	Old Thumb syntax	ARM syntax
The default addressing mode for `LDM` and `STM` is `IA`	`LDMIA, STMIA`	`LDM, STM`
You must use the `S` postfix on instructions that update the flags. This change is essential to avoid conflict with 32-bit Thumb-2 instructions.	`ADD r1, r2, r3` `SUB r4, r5, #6`	`ADDS r1, r2, r3` `SUBS r4, r5, #6`
The preferred form for ALU instructions specifies three registers, even if the destination register is the same as the first operand.	`ADD r7, r8` `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 © 2005 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 © 2005 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.

Note

None of the processors supported by this toolkit supports Thumb-2.

Invoke the ARM assembler using this command:

armasm *options* {*inputfile*}

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

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

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

--apcs , [*qualifiers*]

This option specifies whether you are using the *Procedure Call Standard for the ARM Architecture* (AAPCS). It can also specify some attributes of code sections. See *AAPCS* on page 3-5 for details.

--arm Is a synonym for --32.

--brief_diagnostics

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

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

--checkreglist

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

--cpu *name* Sets the target CPU. See *CPU names* on page 3-7.

--debug Instructs the assembler to generate DWARF debug tables.

The default is DWARF2. For backwards compatibility, the --dwarf2 option is permitted, but not required.

 Copyright © 2005 ARM Limited. All rights reserved.

`--depend dependfile`

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

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

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

`--dllexport_all`

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

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

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

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

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

`--errors errorfile`

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

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

`--exceptions_unwind`

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

`--fpmode model`

Specifies the floating-point conformance, and sets library attributes and floating-point optimizations. See *Floating-point model* on page 3-6.

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

`-i{dir} [,dir]…`

Adds directories to the source file search path so that arguments to `GET`, `INCLUDE`, or `INCBIN` directives do not need to be fully qualified (see *GET or INCLUDE* on page 6-76).

`--keep` Instructs the assembler to keep local labels in the symbol table of the object file, for use by the debugger (see *KEEP* on page 6-80).

Copyright © 2005 ARM Limited. All rights reserved.

`--list[listingfile]`

Instructs the assembler to output a detailed listing of the assembly language produced by the assembler to *listingfile*. See *Listing output to a file* on page 3-11 for details.

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

`--memaccess attributes`

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

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

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

`--no_exceptions`

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

`--no_exceptions_unwind`

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

`--no_hide_all`

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

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

`--no_warn` Turns off warning messages.

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

`--predefine "directive"`

Instructs the assembler to pre-execute one of the `SET` directives. See *Pre-executing a SET directive* on page 3-10 for details.

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

 Copyright © 2005 ARM Limited. All rights reserved.

`--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.

`--unsafe` Downgrades errors to warnings. See *Controlling the output of diagnostic messages* on page 3-11).

`--via file` Instructs the assembler to open *`file`* and read in command-line arguments to the assembler. For more information see the *Via File Syntax* appendix in *RealView Developer Kit v2.2 Compiler and Libraries Guide*.

`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
```

3.1.2 AAPCS

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

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

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.

 Copyright © 2005 ARM Limited. All rights reserved.

/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.
/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.
/swstackcheck	specifies that the code in *inputfile* carries out software stack-limit checking.
/noswstackcheck	specifies that the code in *inputfile* does not carry out software stack-limit checking. This is the default.
/swstna	specifies that the code in *inputfile* is compatible both with code which carries out stack-limit checking, and with code that does not carry out stack-limit checking.

3.1.3 Floating-point model

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

--fpmode *model*

this option selects the target floating-point model.

model can be one of:

ieee_full	All facilities, operations, and representations guaranteed by the IEEE standard are available in single and double-precision. Modes of operation can be selected dynamically at runtime.

 Copyright © 2005 ARM Limited. All rights reserved.

This defines the symbols:

```
__FP_IEEE
__FP_FENV_EXCEPTIONS
__FP_FENV_ROUNDING
__FP_INEXACT_EXCEPTION
```

`ieee_fixed`

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

This defines the symbols:

```
__FP_IEEE
__FP_FENV_EXCEPTIONS
```

`ieee_no_fenv`

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

This defines the symbol `__FP_IEEE`.

`std` IEEE finite values with denormals flushed to zero, round-to-nearest and no exceptions. It is C and C++ compatible. This is the default option.

Finite values are as predicted by the IEEE standard. It is not guaranteed that NaNs and infinities are produced in all circumstances defined by the IEEE model, or that when they are produced, they have the same sign. Also, it is not guaranteed that the sign of zero is that predicted by the IEEE model.

`fast` Some value altering optimizations, where accuracy is sacrificed to fast execution. This is not IEEE compatible, and is not standard C.

This defines the symbol `__FP_FAST`.

3.1.4 CPU names

There is an option to specify the CPU name:

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

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

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

Copyright © 2005 ARM Limited. All rights reserved.

Obtaining a list of valid CPU names

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

```
armasm --cpu list
```

3.1.5 FPU names

There is an option to specify the FPU name:

`--fpu` *`name`* this option 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 `fpu` *`name`* are:

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

`softvfp` Selects software floating-point library (FPLib) with pure-endian doubles. This is the default if no `--fpu` option is specified, and the `--cpu` option selected does not imply a particular FPU.

Note

The use of `fpa` and `softfpa` is no longer supported in RVDK.

The compilation tools only accept `-fpu none` or `-fpu softvfp` (default).

See the *C and C++ Compilers* chapter in *RealView Developer Kit v2.2 Compiler and Libraries Guide* for details of the effect 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
```

 Copyright © 2005 ARM Limited. All rights reserved.

3.1.6 Memory access attributes

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

`--memaccess` *attributes*

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

`+L41` Enable unaligned `LDR`s.

`-L22` Do not enable halfword loads.

`-S22` Do not enable halfword stores.

`-L22-S22` Do not enable halfword loads and stores.

Copyright © 2005 ARM Limited. All rights reserved.

3.1.7 Pre-executing a SET directive

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

`--predefine "directive"`

You must enclose *directive* in quotes. See *SETA, SETL, and SETS* on page 6-8. The assembler also executes a corresponding 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.

3.1.8 Splitting long LDMs and STMs

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

`--split_ldm`

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

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

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

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

Also, avoiding large multiple register transfers:

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

 Copyright © 2005 ARM Limited. All rights reserved.

3.1.9 Listing output to a file

Use the following option to list output to a file:

`--list [listingfile]`

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

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

`--no_terse` turns the `terse` flag off. When this option is on, lines skipped due to conditional assembly do not appear in the listing. If the `terse` option is off, these lines do appear in the listing. The default is on.

`--width` sets the listing page width. The default is 79 characters.

`--length` sets the listing page length. Length zero means an 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.

3.1.10 Controlling the output of diagnostic messages

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

`--brief_diagnostics`

Enables or disables a mode where a shorter form of the diagnostic output is used. When enabled, the original source line is not displayed and the error message text is not wrapped when it is too long to fit on a single line. The default is `--no_brief_diagnostics`.

`--diag_style {arm|ide}`

Specifies the style used to display diagnostic messages:

`arm` Display messages using the ARM assembler style. This is the default if `--diag_style` is not specified.

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

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

sets the diagnostic messages that have the specified tag(s) to the error severity.

Copyright © 2005 ARM Limited. All rights reserved.

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

Sets the diagnostic messages that have the specified tag(s) to the remark severity.

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

Sets the diagnostic messages that have the specified tag(s) to the warning severity.

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

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

`--unsafe` Enables assembly of a file containing instructions that are not available on the specified architecture and processor. It changes corresponding error messages to warning messages. It also suppresses warnings about operator precedence (see *Binary operators* on page 3-34).

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

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

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

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

Table 3-1 Severity of diagnostic messages

Severity	Description
Catastrophic error	Catastrophic errors indicate problems that cause the assembly to stop. These errors include command-line errors, internal errors, and missing include files. If multiple source files are being assembled, then no further source files are assembled.
Error	Errors indicate violations in the syntactic or semantic rules of Assembly language. Assembly continues, but object code is not generated.
Warning	Warnings indicate unusual conditions in your code that might indicate a problem. Assembly continues, and object code is generated unless any problems with an Error severity are detected.
Remark	Remarks indicate common, but not recommended, use of Assembly language. These diagnostics are not issued by default. Assembly continues, and object code is generated unless any problems with an Error severity are detected.

 Copyright © 2005 ARM Limited. All rights reserved.

3.1.11 Controlling exception table generation

There are four options that control exception table generation:

`--exceptions` Instructs the assembler to switch on exception table generation for all functions encountered.

`--no_exceptions`

Instructs the assembler to switch off exception table generation. No tables are generated.

`--exceptions_unwind`

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

`--no_exceptions_unwind`

Instructs the assembler to produce *nounwind* tables for every function.

For finer control, use `FRAME UNWIND ON` and `FRAME UNWIND OFF` directives, see *FRAME UNWIND ON* on page 6-50 and *FRAME UNWIND OFF* on page 6-50.

Unwind tables

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

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

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

The assembler can generate *nounwind* table entries for all functions and non-functions. The assembler can generate an *unwind* table for a functions only if the function contains sufficient `FRAME` directives to describe the use of the stack within the function. Functions must conform to the conditions set out in the *Exception Handling ABI for the ARM Architecture* [EHABI], section 9.1 *Constraints on Use*. If the assembler cannot generate an *unwind* table it generates a *nounwind* table.

Copyright © 2005 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 directives in all upper case, as in this manual. Alternatively, you can write directives in all lower case. You must not write a directive in mixed upper and lower case.

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

symbol is usually a label (see *Labels* on page 3-21). In instructions and pseudo-instructions it is always a label. In some directives it is a symbol for a variable or a constant. The description of the directive makes this clear in each case.

symbol must begin in the first column and cannot contain any whitespace character such as a space or a tab (see *Symbol naming rules* on page 3-18).

Copyright © 2005 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)
- `sl` and `SL` (stack limit, r10)
- `fp` and `FP` (frame pointer, r11)
- `ip` and `IP` (intra-procedure-call scratch register, r12)
- `sp` and `SP` (stack pointer, r13)
- `lr` and `LR` (link register, r14)
- `pc` and `PC` (program counter, r15).

3.3.2 Predeclared program status register names

The following program status register names are predeclared:

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

3.3.3 Predeclared floating-point register names

The following floating-point register names are predeclared:

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

Note

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

3.3.4 Predeclared coprocessor names

The following coprocessor names and coprocessor register names are predeclared:

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

Copyright © 2005 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. See also *Determining the armasm version at assembly time* on page 3-17
|ads$version|	Has the same value as {ARMASM_VERSION}, see above.
{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. If an architecture was specified in the command line --cpu option, {CPU} holds the value "Generic ARM".
{ENDIAN}	Has the value little.
{FPIC}	Has the value True if /fpic is set. The default is False.
{FPU}	Holds the name of the selected fpu. The default is SoftVFP.
{INPUTFILE}	Holds the name of the current source file.
{INTER}	Has the value True if /inter is set. The default is False.
{LINENUM}	Holds an integer indicating the line number in the current source file.
{NOSWST}	Has the value True if /noswst is set. The default is False.
{OPT}	Value of the currently-set listing option. The OPT directive can be used to save the current listing option, force a change in it, or restore its original value.
{PC} or .	Address of current instruction.
{PCSTOREOFFSET}	Is the offset between the address of the STR pc,[...] or STM Rb,{..., pc} instruction and the value of pc stored out. This varies depending on the CPU or architecture specified.
{ROPI}	Has the value True if /ropi is set. The default is False.
{RWPI}	Has the value True if /rwpi is set. The default is False.
{SWST}	Has the value True if /swst is set. The default is False.
{VAR} or @	Current value of the storage area location counter.

 Copyright © 2005 ARM Limited. All rights reserved.

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

```
IF {ARCHITECTURE} = "5TE"
```

|ads$version| must be all lower case. The other built-in variables can be upper-case, lower-case, or mixed.

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

Table 3-3 Built-in Boolean constants

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

3.4.1 Determining the armasm version at assembly time

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

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

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


Copyright © 2005 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:

- *Variables* on page 3-19
- *Numeric constants* on page 3-19
- *Labels* on page 3-21
- *Local labels* on page 3-22.

3.5.1 Symbol naming rules

The following general rules apply to symbol names:

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

  ```
  ||ASSERT||
  ```

  The bars are not part of the symbol.

If you need 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 © 2005 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 3-26).

The possible values of a logical variable are `{TRUE}` or `{FALSE}` (see *Logical expressions* on page 3-29).

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 3-25).

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

3.5.3 Numeric constants

Numeric constants are 32-bit integers. You can set them using unsigned numbers in the range 0 to $2^{32} – 1$, or signed numbers in the range $–2^{31}$ to $2^{31} – 1$. However, the assembler makes no distinction between $–n$ and $2^{32} – n$. Relational operators such as >= use the unsigned interpretation. This means that $0 > –1$ is `{FALSE}`.

Use the `EQU` directive to define constants (see *EQU* on page 6-70). You cannot change the value of a numeric constant after you define it.

See also *Numeric expressions* on page 3-26 and *Numeric literals* on page 3-27.

Copyright © 2005 ARM Limited. All rights reserved.

3.5.4 Assembly time substitution of variables

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

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

Use a dot to mark the end of the variable name if the following character would be permissible in a symbol name (see *Symbol naming rules* on page 3-18). You must set the contents of the variable before you can use it.

If you need 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 © 2005 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 6-20
- *DCD and DCDU* on page 6-21
- *DCFD and DCFDU* on page 6-23
- *DCFS and DCFSU* on page 6-24
- *DCI* on page 6-25
- *DCQ and DCQU* on page 6-26
- *DCW and DCWU* on page 6-27.

Register-relative labels

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

- *MAP* on page 6-17
- *SPACE* on page 6-19
- *DCDO* on page 6-22
- *EQU* on page 6-70.

Absolute addresses

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

Copyright © 2005 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 ELF section.

RVDK uses an ELF proprietary file format called *ARM Toolkit Proprietary ELF* (ATPE). The file format for each version of RVDK is restricted to the proprietary ATPE format for the permitted device. This is referred to as *ATPE_Custom*.

Local labels are typically used for loops and conditional code within a routine, or for small subroutines that are only used locally. They are particularly useful in macros (see *MACRO and MEND* on page 6-30).

Use the `ROUT` directive to limit the scope of local labels (see *ROUT* on page 6-85). A reference to a local label refers to a matching label within the same scope. If there is no matching label within the scope in either direction, the assembler generates an error message and the assembly fails.

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

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

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

Syntax

The syntax of a local label is:

n{*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` or `B` is specified, the assembler searches backwards first, then forwards.

 Copyright © 2005 ARM Limited. All rights reserved.

If neither A or 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 © 2005 ARM Limited. All rights reserved.

3.6 Expressions, literals, and operators

This section contains the following subsections:

 Copyright © 2005 ARM Limited. All rights reserved.

3.6.1 String expressions

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

- *String literals*
- *Variables* on page 3-19
- *Unary operators* on page 3-32
- *String manipulation operators* on page 3-34
- *SETA, SETL, and SETS* on page 6-8.

Characters that cannot be placed in string literals can be placed in string expressions using the `:CHR:` unary operator. Any ASCII character from 0 to 255 is permitted.

The value of a string expression cannot exceed 512 characters in length. It can be of zero length.

Example

```
improb  SETS    "literal":CC:(strvar2:LEFT:4)
            ; sets the variable improb to the value "literal"
            ; with the left-most four characters of the
            ; contents of string variable strvar2 appended
```

3.6.2 String literals

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

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

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

Examples

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


Copyright © 2005 ARM Limited. All rights reserved.

3.6.3 Numeric expressions

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

- *Numeric constants* on page 3-19
- *Variables* on page 3-19
- *Numeric literals* on page 3-27
- *Binary operators* on page 3-34
- *SETA, SETL, and SETS* on page 6-8.

Numeric expressions can contain register-relative or program-relative expressions if the overall expression evaluates to a value that does not include a register or the PC.

Numeric expressions evaluate to 32-bit integers. You can interpret them as unsigned numbers in the range 0 to $2^{32} – 1$, or signed numbers in the range $–2^{31}$ to $2^{31} – 1$. However, the assembler makes no distinction between $–n$ and $2^{32} – n$. Relational operators such as >= use the unsigned interpretation. This means that $0 > –1$ is `{FALSE}`.

Example

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


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

3.6.5 Floating-point literals

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

`{-}`*`digits`*`E{-}`*`digits`*

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

`0x`*`hexdigits`*

`&`*`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 range for single-precision floating point values is:

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

The range for double-precision floating point values is:

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

Examples

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

 Copyright © 2005 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 6-17).

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

Example

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

3.6.7 Logical expressions

Logical expressions consist of combinations of logical literals ({TRUE} or {FALSE}), logical variables, Boolean operators, relations, and parentheses (see *Boolean operators* on page 3-37).

Relations consist of combinations of variables, literals, constants, or expressions with appropriate relational operators (see *Relational operators* on page 3-36).

3.6.8 Logical literals

The logical literals are:

- {TRUE}
- {FALSE}.

Copyright © 2005 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 3-32 and *Binary operators* on page 3-34).

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.

Note

The 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.

Table 3-4 shows the order of precedence of operators in `armasm`, and a comparison with the order in C.

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

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

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

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 © 2005 ARM Limited. All rights reserved.

Table 3-4 Operator precedence in armasm

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)
<< >>
< <= > >=
== !=
&
^
|
&&
||

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.

Copyright © 2005 ARM Limited. All rights reserved.

3.6.10 Unary operators

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

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

Table 3-6 Unary operators that return strings

Operator	Usage	Description
:CHR:	:CHR:A	Returns the character with ASCII code A.
:LOWERCASE:	:LOWERCASE:string	Returns the given string, with all uppercase characters converted to lowercase.
:REVERSE_CC:	:REVERSE_CC:cond_code	Returns the inverse of the condition code in cond_code.
:STR:	:STR:A	Returns an 8-digit hexadecimal string corresponding to a numeric expression, or the string "T" or "F" if used on a logical expression.
:UPPERCASE:	:UPPERCASE:string	Returns the given string, with all lowercase characters converted to uppercase.

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

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

Operator	Alias	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.
: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.

Copyright © 2005 ARM Limited. All rights reserved.

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

Operator	Alias	Usage	Description
:LEN:		:LEN:A	Length of string A.
:LNOT:		:LNOT:A	Logical complement of A.
:NOT:	~	:NOT:A ~A	Bitwise complement of A.
:RCONST:		:RCONST:Rn	Number of register, 0-15 corresponding to r0-r15.

Copyright © 2005 ARM Limited. All rights reserved.

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* on page 3-30.

Multiplicative operators

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

Table 3-8 shows the multiplicative operators.

Table 3-8 Multiplicative operators

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

String manipulation operators

Table 3-9 shows the string manipulation operators.

In the slicing operators `LEFT` and `RIGHT`:

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

In `CC`, `A` and `B` must both be strings.

Table 3-9 String manipulation operators

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

Copyright © 2005 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 © 2005 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	Aliases	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 © 2005 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 © 2005 ARM Limited. All rights reserved.

3.7 Diagnostic messages

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

3.7.1 Interlocks

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

```
armasm --diag_warning 1563
```

This warning is off by default.

 Copyright © 2005 ARM Limited. All rights reserved.

3.8 Using the C preprocessor

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

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

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

Example 3-1 Preprocessing an assembly language source file

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


Copyright © 2005 ARM Limited. All rights reserved.

 Copyright © 2005 ARM Limited. All rights reserved.

Chapter 4
ARM and Thumb instructions

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

Copyright © 2005 ARM Limited. All rights reserved.

4.1 Instruction summary

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

The § column in Table 4-1 shows the ARM architecture in which each instruction first appeared.

Table 4-1 Location of instructions

Mnemonic	Brief description	Page	§
ADC, ADD	Add with Carry, Add	page 4-41	All
ADR pseudo-instruction	Load program or register-relative address (short range)	page 4-101	All
ADRL pseudo-instruction	Load program or register-relative address (medium range)	page 4-103	All
AND	Logical AND	page 4-45	All
ASR	Arithmetic Shift Right	page 4-56	All
B	Branch	page 4-79	All
BIC	Bit Clear	page 4-45	All
BKPT	Breakpoint	page 4-93	5
BL	Branch with Link	page 4-79	All
CDP, CDP2	Coprocessor Data Processing operation	page 4-83	All, 5
CLZ	Count leading zeros	page 4-48	5
CMN, CMP	Compare Negative, Compare	page 4-49	All
EOR	Exclusive OR	page 4-45	All
LDC, LDC2	Load Coprocessor	page 4-88	All, 5
LDM	Load Multiple registers	page 4-29	All
LDR	Load Register instructions	page 4-5	All
LDR pseudo-instruction	Load Register pseudo-instruction	page 4-105	All
LSL, LSR	Logical Shift Left, Logical Shift Right	page 4-56	All
MCR, MCR2, MCRR, MCRR2	Move from Register(s) to Coprocessor	page 4-84	All, 5, 5E, 6

Copyright © 2005 ARM Limited. All rights reserved.

Table 4-1 Location of instructions (continued)

Mnemonic	Brief description	Page	§
MLA, MLS	Multiply Accumulate, Multiply and Subtract	page 4-59	All
MOV	Move	page 4-51	All
MRC, MRC2	Move from Coprocessor to Register	page 4-86	All, 5
MRS	Move from PSR to register	page 4-95	All
MSR	Move from register to PSR	page 4-96	All
MUL	Multiply	page 4-59	All
MVN	Move Not	page 4-51	All
NOP	No Operation	page 4-99	All
ORN	Logical OR NOT	page 4-45	T2
ORR	Logical OR	page 4-45	All
PUSH, POP	PUSH, POP registers	page 4-33	All T
QADD, QDADD, QDSUB, QSUB	Saturating Arithmetic	page 4-70	5ExP
ROR	Rotate Right Register	page 4-56	All
RSB, RSC, SBC	Reverse Sub, Reverse Sub with Carry, Sub with Carry	page 4-41	All
SMI	Secure Monitor Interrupt	page 4-98	Z
SMLAL	Signed Multiply Accumulate (64 <= 64 + 32 x 32)	page 4-61	M
SMLALxy	Signed Multiply Accumulate (64 <= 64 + 16 x 16)	page 4-67	5ExP
SMULL	Signed Multiply (64 <= 32 x 32)	page 4-61	M
SMULxy	Signed Multiply (32 <= 16 x 16)	page 4-63	5ExP
SMULWy	Signed Multiply (32 <= 32 x 16)	page 4-65	5ExP
STC	Store Coprocessor	page 4-88	All
STC2	Store Coprocessor	page 4-90	5ExP
STM	Store Multiple registers	page 4-29	All
STR	Store Register instructions	page 4-5	All
SUB	Subtract	page 4-41	All

Copyright © 2005 ARM Limited. All rights reserved.

Table 4-1 Location of instructions (continued)

Mnemonic	Brief description	Page	§
SWI	Software Interrupt	page 4-94	All
SWP, SWPB	Swap registers and memory (ARM only)	page 4-36	All
TEQ, TST	Test Equivalence, Test	page 4-54	All
UMLAL, UMULL	Unsigned Multiply Accumulate, Multiply (64 <= 32 x 32 + 64), (64 <= 32 x 32)	page 4-61	M

Copyright © 2005 ARM Limited. All rights reserved.

4.2 Memory access instructions

This section contains the following subsections:

- *Address alignment* on page 4-6
  Alignment considerations that apply to all memory access instructions.
- *LDR and STR (zero, immediate, or pre-indexed immediate offset)* on page 4-7
  Load and store bytes, halfwords, words and doublewords.
- *LDR and STR (post-indexed immediate offset)* on page 4-12
  Load and store bytes, halfwords, words and doublewords.
- *LDR and STR (register or pre-indexed register offset)* on page 4-16
  Load and store bytes, halfwords, words and doublewords.
- *LDR and STR (post-indexed register offset)* on page 4-20
  Load and store bytes, halfwords, words and doublewords.
- *LDR (PC-relative)* on page 4-23
  Load bytes, halfwords, words and doublewords.
- *LDM and STM* on page 4-29
  Load and Store Multiple Registers.
- *PUSH and POP* on page 4-33
  Push low registers, and optionally the LR, onto the stack.
  Pop low registers, and optionally the PC, off the stack.
- *SWP and SWPB* on page 4-36
  Swap data between registers and memory.

Note

There is also an `LDR` pseudo-instruction (see *LDR pseudo-instruction* on page 4-105). This pseudo-instruction either assembles to an `LDR` instruction, or to a `MOV` or `MVN` instruction.

Compilation tools are restricted to the permitted endianness of the target microcontroller unit (MCU).

Copyright © 2005 ARM Limited. All rights reserved.

4.2.1 Address alignment

In most circumstances, you must ensure that addresses for 4-byte transfers are 4-byte word-aligned, and addresses for 2-byte transfers are 2-byte aligned. In ARM v7, there is a configuration option to permit unaligned accesses.

In ARM v6 and below, if your system has a system coprocessor (cp15), you can enable alignment checking. Non word-aligned 32-bit transfers cause an alignment exception if alignment checking is enabled.

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

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

  For a little-endian memory system, this causes the addressed byte to occupy the least significant byte of the register.

  For a big-endian memory system, it causes the addressed byte to occupy:

  — bits[31:24] if bit[0] of the address is 0

  — bits[15:8] if bit[0] of the address is 1.

 Copyright © 2005 ARM Limited. All rights reserved.

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

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

Note

Also, see *Pseudo-instructions* on page 4-100.

Syntax

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

where:

op can be any one of:
- LDR Load Register
- STR Store Register.

type can be any one of:
- B unsigned Byte
- SB Signed Byte (LDR only)
- H unsigned Halfword
- SH Signed Halfword (LDR only)
- - omitted, for Word
- D Doubleword.

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

cond is an optional condition code (see *Conditional execution* on page 2-16).

Rd is the ARM register to load or save.

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

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

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

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

Copyright © 2005 ARM Limited. All rights reserved.

Zero offset

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

The T suffix is not available for Doubleword instructions.

Immediate offset

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

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

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

Note

None of the processors supported by this toolkit supports Thumb-2.

Pre-indexed immediate offset

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

The range of offsets allowed is:

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

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

 Copyright © 2005 ARM Limited. All rights reserved.

Doubleword register restrictions

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

For ARM instructions:

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

Copyright © 2005 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 restrictions apply to 16-bit instructions:

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

Loading to r15

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

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

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

In ARMv5 and above:

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

You cannot use the `T` suffix when loading to r15.

Architectures

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

 Copyright © 2005 ARM Limited. All rights reserved.

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

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

In T and T2 variants of ARMv5 and above:

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

Examples

```
LDR     r8,[r10]            ; loads r8 from the address in r10.

LDRNE   r2,[r5,#960]!       ; (conditionally) loads r2 from a word
                            ; 960 bytes above the address in r5, and
                            ; increments r5 by 960.

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

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

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


Copyright © 2005 ARM Limited. All rights reserved.

4.2.3 LDR and STR (post-indexed immediate offset)

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

Syntax

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

where:

op can be any one of:

- LDR Load Register
- STR Store Register.

type can be any one of:

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

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

cond is an optional condition code (see *Conditional execution* on page 2-16).

Rd is the ARM register to load or save.

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

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

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

 Copyright © 2005 ARM Limited. All rights reserved.

Operation and restrictions

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

The range of offsets allowed is:

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

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

Doubleword register restrictions

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

For ARM instructions:

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

Copyright © 2005 ARM Limited. All rights reserved.

Loading to r15

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

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

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

In ARMv5 and above:

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

You cannot use the `T` suffix when loading to r15.

Copyright © 2005 ARM Limited. All rights reserved.

Saving from r15

In Thumb code, you cannot save from r15.

In ARM code, avoid saving from r15 if possible.

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

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

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

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

Architectures

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

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

There are no 16-bit versions of these instructions.

In T and T2 variants of ARMv5 and above:

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

Copyright © 2005 ARM Limited. All rights reserved.

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

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

Syntax

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

where:

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

type can be any one of:
- `B` unsigned Byte
- `SB` Signed Byte (`LDR` only)
- `H` unsigned Halfword
- `SH` Signed Halfword (`LDR` only)
- `-` omitted, for Word
- `D` Doubleword.

cond is an optional condition code (see *Conditional execution* on page 2-16).

Rd is the ARM register to load or save.

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

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

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

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

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

Operation and restrictions

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

The range of shifts allowed is:

- `LSL` 0 to 3 for all Thumb-2 instructions except Doubleword

 Copyright © 2005 ARM Limited. All rights reserved.

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

Doubleword register restrictions

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

For ARM instructions:

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

Copyright © 2005 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 restrictions apply to 16-bit instructions:

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

Loading to r15

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

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

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

In ARMv5 and above:

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

You cannot use the `T` suffix when loading to r15.

Copyright © 2005 ARM Limited. All rights reserved.

Saving from r15

In Thumb code, you cannot save from r15.

In ARM code, avoid saving from r15 if possible.

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

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

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

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

Architectures

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

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

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

In T and T2 variants of ARMv5 and above:

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

Copyright © 2005 ARM Limited. All rights reserved.

4.2.5 LDR and STR (post-indexed register offset)

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

Syntax

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

where:

op can be any one of:
- LDR Load Register
- STR Store Register.

type can be any one of:
- B unsigned Byte
- SB Signed Byte (LDR only)
- H unsigned Halfword
- SH Signed Halfword (LDR only)
- - omitted, for Word
- D Doubleword.

cond is an optional condition code (see *Conditional execution* on page 2-16).

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

Rd is the ARM register to load or save.

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

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

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

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

Operation and restrictions

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

 Copyright © 2005 ARM Limited. All rights reserved.

The range of shifts allowed is:

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

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

Doubleword register restrictions

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

For ARM instructions:

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

Loading to r15

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

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

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

In ARMv5 and above:

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

You cannot use the `T` suffix when loading to r15.

Copyright © 2005 ARM Limited. All rights reserved.

Saving from r15

In Thumb code, you cannot save from r15.

In ARM code, avoid saving from r15 if possible.

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

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

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

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

Architectures

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

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

There are no 16-bit versions of these instructions.

In T and T2 variants of ARMv5 and above:

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

 Copyright © 2005 ARM Limited. All rights reserved.

4.2.6 LDR (PC-relative)

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

Syntax

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

where:

type can be any one of:

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

cond is an optional condition code (see *Conditional execution* on page 2-16).

`.w` is an optional instruction width specifier. See *LDR (PC-relative) in Thumb-2* for details.

Rd is the ARM register to load or save.

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

label is a program-relative expression. See *Register-relative and program-relative expressions* on page 3-29 for more information.

label must be within ±4KB of the current instruction.

Usage

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

LDR (PC-relative) in Thumb-2

You can use the `.W` width specifier to force `LDR` to generate a 32-bit instruction in Thumb-2 code.

`LDR.W` always generates a 32-bit instruction, even if the target could be reached using a 16-bit `LDR`.

Copyright © 2005 ARM Limited. All rights reserved.

For forward references, LDR without .W always generates a 16-bit instruction in Thumb code, even if that results in failure for a target that could be reached using a 32-bit Thumb-2 LDR instruction.

Doubleword register restrictions

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

For ARM instructions:

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

Copyright © 2005 ARM Limited. All rights reserved.

16-bit instruction

A 16-bit version of this instruction is available in Thumb-2 code, and in Thumb code on Thumb-capable processors conforming to ARM v6 and earlier.

The following restrictions apply to the 16-bit instruction:

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

Loading to r15

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

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

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

In ARMv5 and above:

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

You cannot use the `T` suffix when loading to r15.

Architectures

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

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

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

In T and T2 variants of ARMv5 and above:

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

Copyright © 2005 ARM Limited. All rights reserved.

4.2.7 PLD

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

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

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

Syntax

There are three forms of the PLD instruction:

- Immediate or zero offset
- Register offset or shifted register offset
- PC-relative.

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

```
PLD{cond} [Rn {, #offset}]
PLD{cond} [Rn, +/-Rm {, shift}]
PLD{cond} label
```

where:

cond is an optional condition code (see *Conditional execution* on page 2-16).

Note

This is an unconditional instruction in ARM. *cond* is only allowed in Thumb-32 code, using a preceding IT instruction.

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

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

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

shift is an optional shift.

label is a program-relative expression. See *Register-relative and program-relative expressions* on page 3-29 for more information.

 Copyright © 2005 ARM Limited. All rights reserved.

Immediate offset

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

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

Zero offset

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

Register or shifted register offset

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

The range of shifts allowed is:

- LSL 0 to 3 for Thumb-2 instructions
- Any one of the following for ARM instructions:
  - LSL 0 to 31
  - LSR 1 to 32
  - ASR 1 to 32
  - ROR 1 to 31
  - RRX

PC-relative

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

Copyright © 2005 ARM Limited. All rights reserved.

Address alignment for preloads

No alignment checking is performed for preload instructions.

Architectures

This ARM instruction is available in ARM v5TE and above.

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

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

Copyright © 2005 ARM Limited. All rights reserved.

4.2.8 LDM and STM

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

See also *PUSH and POP* on page 4-33.

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 2-16).

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 4-32).

See *Restrictions on reglist in 32-bit Thumb-2 instructions* on page 4-30.

`^` 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 © 2005 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.

Copyright © 2005 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 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 4-33 for details.

Loading to r15

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

Loading or storing the base register, with writeback

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

- if the instruction is STM or STMIA and *Rn* is the lowest-numbered register in *reglist*, the initial value of *Rn* is stored
- otherwise, the loaded or stored value of *Rn* is ****** Unknown value "smallcaps" for Role attribute******unpredictable*.

Architectures

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

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

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

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

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

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

Copyright © 2005 ARM Limited. All rights reserved.

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 © 2005 ARM Limited. All rights reserved.

4.2.9 PUSH and POP

Push registers onto the stack.

Pop registers off the stack.

Syntax

`PUSH{cond} reglist`

`POP{cond} reglist`

where:

cond is an optional condition code (see *Conditional execution* on page 2-16).

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

Usage

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

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

Copyright © 2005 ARM Limited. All rights reserved.

POP, with reglist including the pc

This instruction causes a branch to the address popped off the stack into pc. This is usually a return from a subroutine, where the lr was pushed onto the stack at the start of the subroutine.

In ARMv5T and above:

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

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

16-bit instructions

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

The following restriction applies to the 16-bit instructions:

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

Architectures

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

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

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

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

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

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

16-bit examples

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

 Copyright © 2005 ARM Limited. All rights reserved.

ARM and 32-bit Thumb-2 examples

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

Incorrect examples

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


Copyright © 2005 ARM Limited. All rights reserved.

4.2.10 SWP and SWPB

Swap data between registers and memory.

You can use SWP and SWPB to implement semaphores. The use of SWP and SWPB is deprecated in ARMv6 and above.

Syntax

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

where:

cond is an optional condition code (see *Conditional execution* on page 2-16).

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

Rd is an ARM register. Data from memory is loaded into *Rd*.

Rm is an ARM register. The contents of *Rm* is saved to memory.

Rm can be the same register as *Rd*. In this case, the contents of the register is swapped with the contents of the memory location.

Rn is an ARM register. The contents of *Rn* specify the address in memory with which data is to be swapped. *Rn* must be a different register from both *Rd* and *Rm*.

Architectures

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

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

 Copyright © 2005 ARM Limited. All rights reserved.

4.3 General data processing instructions

This section contains the following subsections:

Copyright © 2005 ARM Limited. All rights reserved.

4.3.1 Flexible second operand

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

Syntax

Operand2 has two possible forms:

#constant

Rm{, *shift*}

where:

constant is an expression evaluating to a numeric constant. The range of constants available in ARM and Thumb-2 is not exactly the same. See *Constants in Operand2* on page 4-39 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 © 2005 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 may 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 4-40).

Copyright © 2005 ARM Limited. All rights reserved.

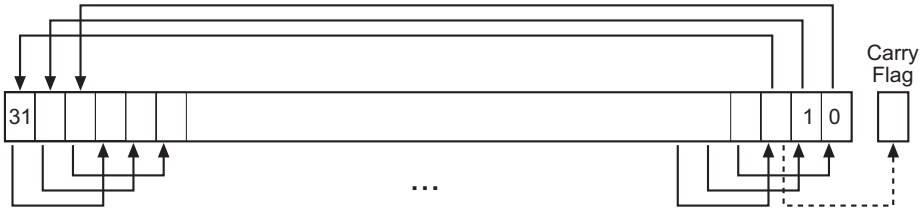

Figure 4-1 ROR

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*).

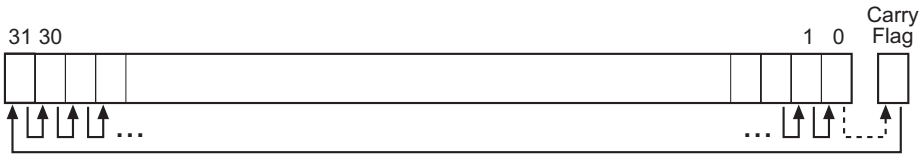

Figure 4-2 RRX

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.

Copyright © 2005 ARM Limited. All rights reserved.

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

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

See also *Parallel add and subtract* on page 4-73.

Syntax

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

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 2-16).

cond is an optional condition code (see *Conditional execution* on page 2-16).

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 4-38 for details of the option. See also *Wide constants* on page 4-42.

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 4-44).

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

Copyright © 2005 ARM Limited. All rights reserved.

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 4-40 for details.

Wide constants

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

You cannot use the S suffix with wide constants.

Use of r15 in Thumb-2 instructions

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

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

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.

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 4-38).

 Copyright © 2005 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`	ARM v6 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.

Architectures

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

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

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

Copyright © 2005 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
RSCLES  r0,r5,r0,LSL r4     ; conditional, flags set
```

Incorrect example

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

Multiword arithmetic examples

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

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

These instructions subtract one 96-bit integer from another:

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

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

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


Copyright © 2005 ARM Limited. All rights reserved.

4.3.3 AND, ORR, EOR, and BIC

Logical AND, OR, Exclusive OR, and Bit Clear.

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

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 2-16).

cond is an optional condition code (see *Conditional execution* on page 2-16).

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 4-38 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*.

In certain circumstances, the assembler can substitute BIC for AND, or AND for BIC. Be aware of this when reading disassembly listings. See *Instruction substitution* on page 4-40 for details.

Use of r15 in ARM instructions

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

If you use r15 as *Rd*:

- Execution branches to the address corresponding to the result.

Copyright © 2005 ARM Limited. All rights reserved.

- 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.

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 4-38).

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 4-38)
- do not affect the V flag.

16-bit instructions

The following forms of these instructions are available in Thumb code:

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

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

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

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

In the first three cases, it doesn't matter if you specify *OP*S *Rd*, *Rm*, *Rd*. The instruction is the same.

Architectures

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

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

 Copyright © 2005 ARM Limited. All rights reserved.

ARM 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
```

Incorrect example

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


Copyright © 2005 ARM Limited. All rights reserved.

4.3.4 CLZ

Count Leading Zeros.

Syntax

`CLZ{cond} Rd, Rm`

where:

cond	is an optional condition code (see *Conditional execution* on page 2-16)
Rd	is the destination register. *Rd* must not be r15.
Rm	is the operand register. *Rm* must not be r15.

Usage

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

Condition flags

This instruction does not change the flags.

Architectures

This ARM instruction is available in ARMv5 and above.

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

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

Examples

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

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

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

 Copyright © 2005 ARM Limited. All rights reserved.

4.3.5 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 2-16)

Rn is the ARM register holding the first operand.

Operand2 is a flexible second operand. See *Flexible second operand* on page 4-38 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 4-40 for details.

Use of r15 in ARM instructions

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

You cannot use r15 for any operand in any data processing instruction that has a register-controlled shift (see *Flexible second operand* on page 4-38).

Use of r15 in Thumb-2 instructions

You cannot use r15 for any operand in these instructions.

Copyright © 2005 ARM Limited. All rights reserved.

Condition flags

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

16-bit instructions

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

`CMP Rn, Rm`	*Rn* and *Rm* can both be Lo or Hi registers
`CMN Rn, Rm`	*Rn* and *Rm* must both be Lo registers
`CMP Rn, #imm`	*Rn* must be a Lo register. *imm* range 0-255.

Architectures

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

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

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

Examples

```
CMP     r2, r9
CMN     r0, #6400
CMPGT   r13, r7, LSL #2
```

Incorrect example

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

 Copyright © 2005 ARM Limited. All rights reserved.

4.3.6 MOV and MVN

Move and Move Not.

Syntax

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

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

where:

`S` is an optional suffix. If `S` is specified, the condition code flags are updated on the result of the operation (see *Conditional execution* on page 2-16).

`cond` is an optional condition code (see *Conditional execution* on page 2-16).

`Rd` is the destination register.

`Operand2` is a flexible second operand. See *Flexible second operand* on page 4-38 for details of the options. See also *Wide constants*.

Usage

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

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

In certain circumstances, the assembler can substitute `MVN` for `MOV`, or `MOV` for `MVN`. Be aware of this when reading disassembly listings. See *Instruction substitution* on page 4-40 for details.

Wide constants

In `MOV` instructions, *`Operand2`* can take any value in the range 0-65535, in addition to the normal range of *`Operand2`* values. These wide constants cannot be used with `MVN` instructions.

You cannot use the `S` suffix with a wide constant.

You cannot use r15 as *`Rd`* with a wide constant.

Use of r15

You cannot use r15 for *`Rd`*, or in *`Operand2`*, in the Thumb-2 `MOV` or `MVN` instructions. The remainder of this section applies to the ARM instructions.

Copyright © 2005 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.

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 ****** Unknown value "smallcaps" for Role attribute******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 4-38).

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 4-38)
- 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`	ARM v5 and earlier: either *Rd* or *Rm*, or both, must be a Hi register. ARMv6 and above: this restriction does not apply.

Architectures

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

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

Copyright © 2005 ARM Limited. All rights reserved.

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 examples

```
MVN     r15,r3,ASR r0   ; r15 not permitted with register controlled shift
```


Copyright © 2005 ARM Limited. All rights reserved.

4.3.7 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 2-16).

Rn is the ARM register holding the first operand.

Operand2 is a flexible second operand. See *Flexible second operand* on page 4-38 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 a `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 4-38).

 Copyright © 2005 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 4-38)
- do not affect the V flag.

16-bit instructions

The following form of one of these instructions is available in Thumb code, and is a 16-bit instruction when used in Thumb-2 code:

`TST` *Rn*, *Rm* *Rn* and *Rm* must both be Lo registers.

Architectures

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

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

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

Examples

```
TST     r0, #0x3F8
TEQEQ   r10, r9
TSTNE   r1, r5, ASR r1
```

Incorrect example

```
TEQ     r15, r1, ROR r0     ; r15 not permitted with register
                            ; controlled shift
```


Copyright © 2005 ARM Limited. All rights reserved.

4.3.8 ASR, LSL, LSR, ROR, and RRX

Arithmetic Shift Right, Logical Shift Left, Logical Shift Right, and Rotate Right.

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 2-16).

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 allowed depends on the instruction:

- ASR allowed shifts 1-32
- LSL allowed shifts 0-31
- LSR allowed shifts 1-32
- ROR allowed shifts 1-31.

Usage

ASR provides the signed value of the contents of a register divided by a power of two. It copies the sign bit into vacated bit positions on the left.

LSL provides the value of a register multiplied by a power of two. LSR provides the unsigned value of a register divided by a variable power of two. Both instructions insert zeros into the vacated bit positions.

ROR provides the value of the contents of a register rotated by a value. The bits that are rotated off the right end are inserted into the vacated bit positions on the left.

 Copyright © 2005 ARM Limited. All rights reserved.

RRX provides the value of the contents of a register shifted right one bit. The old carry flag is shifted into bit[31]. If the S suffix is present, the old bit[0] is placed in the carry flag.

Condition flags

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

The C flag is unaffected if the shift value is 0. Otherwise, the C flag is updated to the last bit shifted out.

16-bit instructions

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

`ASRS Rd, Rm, #sh` *Rd* and *Rm* must both be Lo registers. Allowed shifts 1-32.

`ASRS Rd, Rm, Rs` *Rd*, *Rm*, and *Rs* must all be Lo registers.

`LSLS Rd, Rm, #sh` *Rd* and *Rm* must both be Lo registers. Allowed shifts 0-31.

`LSLS Rd, Rm, Rs` *Rd*, *Rm*, and *Rs* must all be Lo registers.

`LSRS Rd, Rm, #sh` *Rd* and *Rm* must both be Lo registers. Allowed shifts 1-32.

`LSRS Rd, Rm, Rs` *Rd*, *Rm*, and *Rs* must all be Lo registers.

`RORS Rd, Rm, Rs` *Rd*, *Rm*, and *Rs* must all be Lo registers.

Architectures

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

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

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

Examples

```
ASR     r7, r8, r9
LSL     r1, r2, r3
LSLS    r1, r2, r3
LSR     r4, r5, r6
LSRS    r4, r5, r6
ROR     r4, r5, r6
RORS    r4, r5, r6
```


Copyright © 2005 ARM Limited. All rights reserved.

4.4 Multiply instructions

This section contains the following subsections:

 Copyright © 2005 ARM Limited. All rights reserved.

4.4.1 MUL and MLA

Multiply, Multiply-Accumulate, and Multiply-Subtract, with signed or unsigned 32-bit operands, giving the least significant 32 bits of the result.

Syntax

`MUL{S}{cond} Rd, Rm, Rs`

`MLA{S}{cond} Rd, Rm, Rs, Rn`

where:

cond is an optional condition code (see *Conditional execution* on page 2-16).

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 2-16).

Rd is the destination register.

Rm, Rs are registers holding the values to be multiplied.

Rn is a register holding the value to be added or subtracted from.

Usage

The MUL instruction multiplies the values from *Rm* and *Rs*, and places the least significant 32 bits of the result in *Rd*.

The MLA instruction multiplies the values from *Rm* and *Rs*, adds the value from *Rn*, and places the least significant 32 bits of the result in *Rd*.

Do not use r15 for *Rd*, *Rm*, *Rs*, or *Rn*.

Condition flags

If S is specified, the MUL and MLA instructions:

- update the N and Z flags according to the result
- corrupt the C and V flag in ARMv4 and earlier
- do not affect the C or V flag in ARMv5 and above.

16-bit instructions

The following instruction is available in Thumb code:

`MULS Rd, Rd, Rs` *Rd* and *Rs* must both be Lo registers.

Copyright © 2005 ARM Limited. All rights reserved.

Architectures

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

The MULS Thumb instruction is available in all T variants of the ARM architecture.

Examples

```
MUL     r10, r2, r5
MLA     r10, r2, r1, r5
MULS    r0, r2, r2
MULLT   r2, r3, r2
```

Incorrect examples

```
MUL     r15, r0, r3   ; use of r15 not permitted
```


Copyright © 2005 ARM Limited. All rights reserved.

4.4.2 UMULL, UMLAL, SMULL, and SMLAL

Signed and Unsigned Long Multiply, with optional Accumulate, with 32-bit operands, and 64-bit result and accumulator.

Syntax

Op{S}{*cond*} *RdLo*, *RdHi*, *Rm*, *Rs*

where:

Op is one of UMULL, UMLAL, SMULL, or SMLAL.

S is an optional suffix available in ARM state only. If S is specified, the condition code flags are updated on the result of the operation (see *Conditional execution* on page 2-16).

cond is an optional condition code (see *Conditional execution* on page 2-16).

RdLo, *RdHi* are the destination registers. For UMLAL and SMLAL they also hold the accumulating value.

Rm, *Rs* are ARM registers holding the operands.

Usage

Do not use r15 for *RdHi*, *RdLo*, *Rm*, or *Rs*.

RdLo and *RdHi* must be different registers.

The UMULL instruction interprets the values from *Rm* and *Rs* as unsigned integers. It multiplies these integers and places the least significant 32 bits of the result in *RdLo*, and the most significant 32 bits of the result in *RdHi*.

The UMLAL instruction interprets the values from *Rm* and *Rs* as unsigned integers. It multiplies these integers, and adds the 64-bit result to the 64-bit unsigned integer contained in *RdHi* and *RdLo*.

The SMULL instruction interprets the values from *Rm* and *Rs* as two's complement signed integers. It multiplies these integers and places the least significant 32 bits of the result in *RdLo*, and the most significant 32 bits of the result in *RdHi*.

The SMLAL instruction interprets the values from *Rm* and *Rs* as two's complement signed integers. It multiplies these integers, and adds the 64-bit result to the 64-bit signed integer contained in *RdHi* and *RdLo*.

Copyright © 2005 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 supported versions of the ARM architecture.

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

There are no 16-bit Thumb versions of these instructions.

Examples

```
UMULL       r0, r4, r5, r6
UMLALS      r4, r5, r3, r8
```

Incorrect examples

```
UMULL       r1, r15, r10, r2    ; use of r15 not permitted
```


Copyright © 2005 ARM Limited. All rights reserved.

4.4.3 SMUL*xy* and SMLA*xy*

Signed Multiply and Multiply Accumulate, with 16-bit operands and a 32-bit result and accumulator.

Syntax

`SMUL<x><y>{cond} Rd, Rm, Rs`

`SMLA<x><y>{cond} Rd, Rm, Rs, Rn`

where:

`<x>` is either `B` or `T`. `B` means use the bottom half (bits [15:0]) of `Rm`, `T` means use the top half (bits [31:16]) of `Rm`.

`<y>` is either `B` or `T`. `B` means use the bottom half (bits [15:0]) of `Rs`, `T` means use the top half (bits [31:16]) of `Rs`.

`cond` is an optional condition code (see *Conditional execution* on page 2-16).

`Rd` is the destination register.

`Rm, Rs` are the registers holding the values to be multiplied.

`Rn` is the register holding the value to be added.

Usage

Do not use r15 for `Rd`, `Rm`, `Rs`, or `Rn`.

`SMULxy` multiplies the 16-bit signed integers from the selected halves of `Rm` and `Rs`, and places the 32-bit result in `Rd`.

`SMLAxy` multiplies the 16-bit signed integers from the selected halves of `Rm` and `Rs`, adds the 32-bit result to the 32-bit value in `Rn`, and places the result in `Rd`.

Condition flags

These instructions do not affect the N, Z, C, or V flags.

If overflow occurs in the accumulation, `SMLAxy` sets the Q flag. To read the state of the Q flag, use an `MRS` instruction (see *MRS* on page 4-95).

Note

`SMLAxy` never clears the Q flag. To clear the Q flag, use an `MSR` instruction (see *MSR* on page 4-96).

Copyright © 2005 ARM Limited. All rights reserved.

Architectures

These ARM instructions are available in ARMv6 and above, and E variants of ARMv5.

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

There are no 16-bit Thumb versions of these instructions.

Examples

```
SMULTBEQ    r8, r7, r9
SMLABBNE    r0, r2, r1, r10
SMLABT      r0, r0, r3, r5
```

Incorrect examples

```
SMLATB      r0,r7,r8,r15    ; use of r15 not permitted
SMLATTS     r0,r6,r2        ; use of S suffix not permitted
```


Copyright © 2005 ARM Limited. All rights reserved.

4.4.4 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, Rm, Rs`

`SMLAW<y>{cond} Rd, Rm, Rs, Rn`

where:

`<y>` is either `B` or `T`. `B` means use the bottom half (bits [15:0]) of `Rs`, `T` means use the top half (bits [31:16]) of `Rs`.

`cond` is an optional condition code (see *Conditional execution* on page 2-16).

`Rd` is the destination register.

`Rm, Rs` are the registers holding the values to be multiplied.

`Rn` is the register holding the value to be added.

Usage

Do not use r15 for `Rd`, `Rm`, `Rs`, or `Rn`.

`SMULWy` multiplies the signed integer from the selected half of `Rs` by the signed integer from `Rm`, and places the upper 32-bits of the 48-bit result in `Rd`.

`SMLAWy` multiplies the signed integer from the selected half of `Rs` by the signed integer from `Rm`, adds the 32-bit result to the 32-bit value in `Rn`, and places the result in `Rd`.

Condition flags

These instructions do not change the flags.

Architectures

These ARM instructions are available in ARMv6 and above, and E variants of ARMv5.

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

There are no 16-bit Thumb versions of these instructions.

Copyright © 2005 ARM Limited. All rights reserved.

Examples

```
SMULWB      r2, r4, r7
SMULWTVS    r0, r0, r9
```

 Copyright © 2005 ARM Limited. All rights reserved.

4.4.5 SMLAL*xy*

Signed Multiply-Accumulate with 16-bit operands and a 64-bit accumulator.

Syntax

`SMLAL<x><y>{cond} RdLo, RdHi, Rm, Rs`

where:

`<x>` is either `B` or `T`. `B` means use the bottom half (bits [15:0]) of `Rm`, `T` means use the top half (bits [31:16]) of `Rm`.

`<y>` is either `B` or `T`. `B` means use the bottom half (bits [15:0]) of `Rs`, `T` means use the top half (bits [31:16]) of `Rs`.

`cond` is an optional condition code (see *Conditional execution* on page 2-16).

`RdHi, RdLo` are the destination registers. They also hold the accumulate value.

`Rm, Rs` are the registers holding the values to be multiplied.

Usage

Do not use r15 for `RdHi`, `RdLo`, `Rm`, or `Rs`.

`RdHi` and `RdLo` must be different registers.

`SMLALxy` multiplies the signed integer from the selected half of `Rs` by the signed integer from the selected half of `Rm`, and adds the 32-bit result to the 64-bit value in `RdHi` and `RdLo`.

Condition flags

This instruction does not change the flags.

Note

`SMLALxy` cannot raise an exception. If overflow occurs on this instruction, the result wraps round without any warning.

Architectures

This ARM instruction is available in ARMv6 and above, and E variants of ARMv5.

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

Copyright © 2005 ARM Limited. All rights reserved.

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

Examples

```
SMLALTB     r2, r3, r7, r1
SMLALBTVS   r0, r1, r9, r2
```

Incorrect example

```
SMLALTT     r8, r9, r3, r15    ; use of r15 not permitted
```

 Copyright © 2005 ARM Limited. All rights reserved.

4.5 Saturating instructions

This section contains the following subsections:

- *What saturating means*
- *QADD, QSUB, QDADD, and QDSUB* on page 4-70.

Some of the parallel instructions are also saturating, see *Parallel instructions* on page 4-72.

4.5.1 What saturating means

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 occur, 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 4-96).

The Q flag can also be set by two other instructions (see *SMULxy and SMLAxy* on page 4-63 and *SMULWy and SMLAWy* on page 4-65), but these instructions do not saturate.

Copyright © 2005 ARM Limited. All rights reserved.

4.5.2 QADD, QSUB, QDADD, and QDSUB

Signed Add, Subtract, Double and Add, Double and Subtract, saturating the result to the signed range $-2^{31} \leq x \leq 2^{31} - 1$.

See also *Parallel add and subtract* on page 4-73.

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 2-16).

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 4-73 for similar parallel instructions, available in ARMv6 and above only.

 Copyright © 2005 ARM Limited. All rights reserved.

Condition flags

If saturation occurs, these instruction set the Q flag. To read the state of the Q flag, use an `MRS` instruction (see *MRS* on page 4-95).

No other flags are affected.

Architectures

These ARM instructions are available in ARMv6 and above, and E variants of ARMv5.

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

There are no 16-bit Thumb versions of these instructions.

Examples

```
QADD    r0, r1, r9
QDSUBLT r9, r0, r1
```

Incorrect examples

```
QSUBS   r3, r4, r2    ; use of S suffix not permitted
QDADD   r11, r15, r0  ; use of r15 not permitted
```


Copyright © 2005 ARM Limited. All rights reserved.

4.6 Parallel instructions

This section contains the following subsections:

- *Parallel add and subtract* on page 4-73
  Various byte-wise and halfword-wise additions and subtractions.

Copyright © 2005 ARM Limited. All rights reserved.

4.6.1 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 CPSR GE flags.
Q	Signed saturating arithmetic.
SH	Signed arithmetic, halving the results.
U	Unsigned arithmetic modulo 2^8 or 2^{16}. Sets CPSR GE flags.
UQ	Unsigned saturating arithmetic.
UH	Unsigned arithmetic, halving the results.

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 2-16).

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 © 2005 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 CPSR GE flags, see the *Condition flags* section below.
- 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 CPSR as follows:

- For byte-wise operations, the GE flags are used in the same way as the C (Carry) flag for 32-bit `SUB` and `ADD` instructions:
  - `GE[0]` for bits[7:0] of the result
  - `GE[1]` for bits[15:8] of the result
  - `GE[2]` for bits[23:16] of the result
  - `GE[3]` for bits[31:24] of the result.
- For halfword-wise operations, the GE flags are used in the same way as the C (Carry) flag for normal word-wise `SUB` and `ADD` instructions:
  - `GE[1:0]` for bits[15:0] of the result
  - `GE[3:2]` for bits[31:16] of the result.

Note

For halfword-wise operations, GE[1:0] are set or cleared together, and GE[3:2] are set or cleared together.

Architectures

These ARM instructions are available in ARMv6 and above.

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

There are no 16-bit Thumb variants of these instructions.

 Copyright © 2005 ARM Limited. All rights reserved.

Examples

```
SHADD8      r4, r3, r9
USAXNE      r0, r0, r2
```

Incorrect examples

```
QHADD       r2, r9, r3    ; No such instruction, should be QHADD8 or QHADD16
UQSUB16NE   r1, r15, r0   ; Use of r15 not permitted
SAX         r10, r8, r5   ; Must have a prefix.
```


Copyright © 2005 ARM Limited. All rights reserved.

4.7 Packing and unpacking instructions

This section contains the following subsections:

- *SBFX and UBFX* on page 4-77
  Signed or Unsigned Bit Field extract.

 Copyright © 2005 ARM Limited. All rights reserved.

4.7.1 SBFX and UBFX

Signed and Unsigned Bit Field extract. Copies adjacent bits from one register into the least significant bits of a second register, and sign extends or zero extends to 32 bits.

Syntax

`op{cond} Rd, Rm, #lsb, #width`

where:

op is either SBFX or UBFX.

cond is an optional condition code (see *Conditional execution* on page 2-16).

Rd is the destination register.

Rm is the source register.

lsb is the bit number of least significant bit in the bitfield, in the range 0 to 31.

width is the width of the bitfield, in the range 1 to (1+*lsb*).

Do not use r15 for *Rd* or *Rn*

Condition flags

These instructions do not alter any flags.

Architectures

These ARM instructions are available in ARMv6T2 and above.

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

There are no 16-bit Thumb versions of these instructions.

Copyright © 2005 ARM Limited. All rights reserved.

4.8 Branch instructions

This section contains the following subsections:

 Copyright © 2005 ARM Limited. All rights reserved.

4.8.1 B, BL, BX, BLX, and BXJ

Branch, Branch with Link, Branch and exchange instruction set, Branch with Link and exchange instruction set, Branch and change to Jazelle state.

Syntax

op{*cond*} *label*

op{*cond*} *Rm*

where:

op is one of:

B{.w}	branch
BL	branch with link
BX	branch and exchange instruction set
BLX	branch with link, and exchange instruction set
BXJ	branch, and change to to Jazelle execution.

cond is an optional condition code (see *Conditional execution* on page 2-16). *cond* is not available on all forms of this instruction, see *Instruction availability and branch ranges* on page 4-80.

.w is an optional instruction width specifier. See *B in Thumb-2* on page 4-81 for details.

label is a program-relative expression. See *Register-relative and program-relative expressions* on page 3-29 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 © 2005 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 4-2 shows the instructions that are available in ARM and Thumb state. Instructions that are not shown in this table are not available. Notes in brackets show the first architecture version where the instruction is available.

Table 4-2 Branch instruction availability and range

Instruction	ARM		16-bit Thumb		32-bit Thumb-2
`B label`	32MB	(All)	2KB	(All T)	16MB
`B{cond} label`	32MB	(All)	–252 to +258	(All T)	1MB
`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 [a]	(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 [a]	(5T)	16MB
`BLX Rm`	Available	(5)	Available	(5T)	Use 16-bit
`BLX{cond} Rm`	Available	(5)	-		-
`BXJ Rm`	Available	(5J, 6)	-		Available
`BXJ{cond} Rm`	Available	(5J, 6)	-		-

a. This is an instruction pair.

Copyright © 2005 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 *`label`* is placed by the linker. When necessary, the linker adds code to enable longer branches (see the chapter describing basic linker functionality in *RealView Developer Kit v2.2 Linker and Utilities Guide*). The added code is called a *veneer*.

B in Thumb-2

You can use the `.W` width specifier to force `B` to generate a 32-bit instruction in Thumb-2 code.

`B.W` always generates a 32-bit instruction, even if the target could be reached using a 16-bit `B`.

For forward references, `B` without `.W` always generates a 16-bit instruction in Thumb code, even if that results in failure for a target that could be reached using a 32-bit Thumb-2 `B` instruction.

Condition flags

These instructions do not change the flags.

Architectures

See *Instruction availability and branch ranges* on page 4-80 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 © 2005 ARM Limited. All rights reserved.

4.9 Coprocessor instructions

This section does not describe Vector Floating-point instructions (see Chapter 5 *Vector Floating-point Programming*).

It contains the following sections:

 Copyright © 2005 ARM Limited. All rights reserved.

4.9.1 CDP and CDP2

Coprocessor data operations.

Syntax

`CDP{cond} coproc, opcode1, CRd, CRn, CRm{, opcode2}`

`CDP2 coproc, opcode1, CRd, CRn, CRm{, opcode2}`

where:

cond	is an optional condition code (see *Conditional execution* on page 2-16).
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.

Note

`CDP2` is always unconditional.

Exceptions

Undefined Instruction.

Architectures

The `CDP` ARM instruction is available in all versions of the ARM architecture.

The `CDP2` ARM instruction is available in ARMv5 and above.

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

There are no 16-bit Thumb versions of these instructions.

Copyright © 2005 ARM Limited. All rights reserved.

4.9.2 MCR, MCR2, MCRR, and MCRR2

Move to Coprocessor from ARM Register or Registers. Depending on the coprocessor, you might be able to specify various operations in addition.

Syntax

`MCR{cond} coproc, opcode1, Rd, CRn, CRm{, opcode2}`

`MCR2 coproc, opcode1, Rd, CRn, CRm{, opcode2}`

`MCRR{cond} coproc, opcode1, Rd, Rn, CRm`

`MCRR2 coproc, opcode1, Rd, Rn, CRm`

where:

cond is an optional condition code (see *Conditional execution* on page 2-16).

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.

Rd, *Rn* are ARM source registers. Do not use r15 for *Rd* or *Rn*.

CRn, *CRm* are coprocessor registers.

opcode2 is an optional coprocessor-specific opcode.

Usage

The use of these instructions depends on the coprocessor. See the coprocessor documentation for details.

Note

`MCR2` and `MCRR2` are always unconditional in ARM state.

Exceptions

Undefined Instruction.

Architectures

The `MCR` ARM instruction is available in all versions of the ARM architecture.

The `MCR2` ARM instruction is available in ARMv5 and above.

 Copyright © 2005 ARM Limited. All rights reserved.

The MCRR ARM instruction is available in ARMv6 and above, and E variants of ARMv5 excluding xP variants.

The MCRR2 ARM instruction is available in ARMv6 and above.

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

There are no 16-bit Thumb versions of these instructions.

Copyright © 2005 ARM Limited. All rights reserved.

4.9.3 MRC, MRC2, MRRC and MRRC2

Move to ARM Register or Registers from Coprocessor. Depending on the coprocessor, you might be able to specify various operations in addition.

Syntax

`MRC{cond} coproc, opcode1, Rd, CRn, CRm{, opcode2}`

`MRC2 coproc, opcode1, Rd, CRn, CRm{, opcode2}`

`MRRC{cond} coproc, opcode1, Rd, Rn, CRm`

`MRRC2 coproc, opcode1, Rd, Rn, CRm`

where:

cond is an optional condition code (see *Conditional execution* on page 2-16).

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.

Rd, *Rn* are ARM source registers. Do not use r15 for *Rd* or *Rn* in MRRC or MRRC2.

In MRC and MRC2, if *Rd* is r15, only the flags field is affected.

CRn, *CRm* are coprocessor registers.

opcode2 is an optional coprocessor-specific opcode.

Usage

The use of these instructions depends on the coprocessor. See the coprocessor documentation for details.

Note

MRC2 and MRRC2 are always unconditional.

Exceptions

Undefined Instruction.

Architectures

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

 Copyright © 2005 ARM Limited. All rights reserved.

The MRC2 ARM instruction is available in ARMv5 and above.

The MRRC ARM instruction is available in ARMv6 and above, and E variants of ARMv5 excluding xP variants.

The MRRC2 ARM instruction is available in ARMv6 and above.

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

There are no 16-bit Thumb versions of these instructions.

Copyright © 2005 ARM Limited. All rights reserved.

4.9.4 LDC and STC

Transfer Data between memory and Coprocessor.

Syntax

These instructions have three possible forms:

- zero offset
- pre-indexed offset
- post-indexed offset.

The syntax of the three forms, in the same order, are:

`op{L}{cond} coproc, CRd, [Rn]`

`op{L}{cond} coproc, CRd, [Rn, #{-}offset]{!}`

`op{L}{cond} coproc, CRd, [Rn], #{-}offset`

where:

op is either LDC or STC.

cond is an optional condition code (see *Conditional execution* on page 2-16).

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 save.

Rn is the register on which the memory address is based. If r15 is specified, the value used is the address of the current instruction plus eight.

\- is an optional minus sign. If - is present, the offset is subtracted from *Rn*. Otherwise, the offset is added to *Rn*.

offset is an expression evaluating to a multiple of 4, in the range 0 to 1020.

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

Usage

The use of these instructions depends on the coprocessor. See the coprocessor documentation for details.

 Copyright © 2005 ARM Limited. All rights reserved.

Exceptions

Undefined Instruction. Data Abort.

Architectures

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

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

There are no 16-bit Thumb versions of these instructions.

Notes

Use of PC relative addressing in the STC instruction is deprecated in ARMv6T2.

Copyright © 2005 ARM Limited. All rights reserved.

4.9.5 LDC2 and STC2

Transfer Data between memory and Coprocessor, alternative instructions.

Syntax

These instructions have three possible forms:

- zero offset
- pre-indexed offset
- post-indexed offset.

The syntax of the three forms, in the same order, are:

```
op{L} coproc, CRd, [Rn]
op{L} coproc, CRd, [Rn, #{-}offset]{!}
op{L} coproc, CRd, [Rn], #{-}offset
```

where:

op	is either LDC2 or STC2.
L	is an optional suffix specifying a long transfer.
coproc	is the name of the coprocessor the instruction is for. The standard name is p*n*, where *n* is an integer in the range 0 to 15.
CRd	is the coprocessor register to load or save.
Rn	is the register on which the memory address is based. If r15 is specified, the value used is the address of the current instruction plus eight.
-	is an optional minus sign. If - is present, the offset is subtracted from *Rn*. Otherwise, the offset is added to *Rn*.
offset	is an expression evaluating to a multiple of 4, in the range 0 to 1020.
!	is an optional suffix. If ! is present, the address including the offset is written back into *Rn*.

Usage

The use of these instructions depends on the coprocessor. See the coprocessor documentation for details.

 Copyright © 2005 ARM Limited. All rights reserved.

Note

LDC2 and STC2 are always unconditional.

Exceptions

Undefined Instruction. Data Abort.

Architectures

These ARM instructions are available in ARMv5 and above.

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

There are no 16-bit Thumb versions of these instructions.

Copyright © 2005 ARM Limited. All rights reserved.

4.10 Miscellaneous instructions

This section contains the following subsections:

- *BKPT* on page 4-93

  Breakpoint.

- *SWI* on page 4-94

  SoftWare Interrupt.

- *MRS* on page 4-95

  Move the contents of the CPSR or SPSR to a general-purpose register.

- *MSR* on page 4-96

  Load specified fields of the CPSR or SPSR with an immediate constant, or from the contents of a general-purpose register.

- *SMI* on page 4-98

  Secure Monitor Interrupt.

- *NOP* on page 4-99

  Set Event, Wait For Event, Wait for Interrupt, and Yield.

Note

Compilation tools are restricted to the permitted endianness of the target MCU.

Copyright © 2005 ARM Limited. All rights reserved.

4.10.1 BKPT

Breakpoint.

Syntax

`BKPT` *immed*

where:

immed is an expression evaluating to an integer in the range:

- 0-65536 (a 16-bit value) in an ARM instruction
- 0-255 (an 8-bit value) in a 16-bit Thumb instruction.

Usage

The `BKPT` instruction causes the processor to enter Debug state. Debug tools can use this to investigate system state when the instruction at a particular address is reached.

In both ARM state and Thumb state, *immed* is ignored by the ARM hardware. However, a debugger can use it to store additional information about the breakpoint.

Architectures

This ARM instruction is available in ARMv5 and above.

This 16-bit Thumb instruction is available in T variants of ARMv5 and above.

There is no 32-bit Thumb-2 version of this instruction.

Copyright © 2005 ARM Limited. All rights reserved.

4.10.2 SWI

SoftWare Interrupt.

Syntax

`SWI{cond} immed`

where:

cond is an optional condition code (see *Conditional execution* on page 2-16).

immed is an expression evaluating to an integer in the range:

- 0 to 2^{24}–1 (a 24-bit value) in an ARM instruction
- 0-255 (an 8-bit value) in a 16-bit Thumb instruction.

Usage

The `SWI` instruction causes a SWI exception. This means that the processor mode changes to Supervisor, the CPSR is saved to the Supervisor mode SPSR, and execution branches to the SWI vector.

immed is ignored by the processor. However, it can be retrieved by the exception handler to determine what service is being requested.

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.

Copyright © 2005 ARM Limited. All rights reserved.

4.10.3 MRS

Move the contents of the CPSR or SPSR to a general-purpose register.

Syntax

`MRS{cond} Rd, psr`

where:

cond	is an optional condition code (see *Conditional execution* on page 2-16).
Rd	is the destination register. *Rd* must not be r15.
psr	is either CPSR or SPSR.

Usage

Use `MRS` in combination with `MSR` as part of a read-modify-write sequence for updating a PSR, for example to change processor mode, or to clear the Q flag.

In process swap code, the programmer's model state of the process being swapped out must be saved, including relevant PSR contents. Similarly, the state of the process being swapped in must also be restored. These operations make use of `MRS`/store and load/`MSR` instruction sequences.

Caution

You must not attempt to access the SPSR when the processor is in User or System mode. This is your responsibility. The assembler cannot warn you about this because it does not know in what processor mode the code will be executed.

If you do this, the result is ****** Unknown value "smallcaps" for Role attribute******unpredictable*.

The CPSR execution state bits can only be read when the processor is in debug state, halting debug-mode. Otherwise, the execution state bits in the CPSR read as zero.

Condition flags

This instruction does not change the flags.

Architectures

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

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

Copyright © 2005 ARM Limited. All rights reserved.

4.10.4 MSR

Load specified fields of the CPSR or SPSR with an immediate constant, or from the contents of a general-purpose register.

Syntax

`MSR{cond} <psr>_<fields>, #constant`

`MSR{cond} <psr>_<fields>, Rm`

where:

cond is an optional condition code (see *Conditional execution* on page 2-16).

<psr> is either CPSR or SPSR.

<fields> specifies the *Program Status Register* (PSR) field or fields to be moved. *<fields>* can be one or more of:

- `c` control field mask byte, PSR[7:0]
- `x` extension field mask byte, PSR[15:8]
- `s` status field mask byte, PSR[23:16]
- `f` flags field mask byte, PSR[31:24].

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.

Rm is the source register.

Usage

See *MRS* on page 4-95.

In User mode:

- Writes to unallocated, privileged or execution state bits in the CPSR are ignored. This ensures that User mode programs cannot change to a privileged mode.
- Writes to the state bits in the CPSR (defined by StateMask) are ignored in Thumb-2. Previously the behavior was ****** Unknown value "smallcaps" for Role attribute******unpredictable*.

If you access the SPSR when in User or System mode, the result is ****** Unknown value "smallcaps" for Role attribute******unpredictable*.

Copyright © 2005 ARM Limited. All rights reserved.

Condition flags

This instruction updates the flags explicitly if the `f` field is specified.

Architectures

This ARM instruction is available in ARMv3 and above.

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

Copyright © 2005 ARM Limited. All rights reserved.

4.10.5 SMI

Secure Monitor Interrupt.

For details see the *ARM Architecture Reference Manual Security Extensions supplement*.

Syntax

`SMI{cond} #immed_16`

where:

cond is an optional condition code (see *Conditional execution* on page 2-16).

immed_16 is a 16-bit immediate value. This is ignored by the ARM processor, but can be used by the SMI exception handler to determine what service is being requested.

Architectures

This ARM instruction is available in Z variants of ARMv6 and above.

This 32-bit Thumb-2 instruction is available in Z variants of ARMv6T2 and above.

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

 Copyright © 2005 ARM Limited. All rights reserved.

4.10.6 NOP

No Operation.

Syntax

`NOP{`*cond*`}`

where:

cond is an optional condition code (see *Conditional execution* on page 2-16).

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.

Architectures

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

Copyright © 2005 ARM Limited. All rights reserved.

4.11 Pseudo-instructions

The ARM assembler supports a number of pseudo-instructions that are translated into the appropriate combination of ARM, Thumb, or Thumb-2 instructions at assembly time.

The pseudo-instructions are described in the following sections:

- *ADR pseudo-instruction* on page 4-101

  Load a program-relative or register-relative address (short range, position independent)

- *ADRL pseudo-instruction* on page 4-103

  Load a program-relative or register-relative address into a register (medium range, position independent)

- *LDR pseudo-instruction* on page 4-105

  Load a register with a 32-bit constant value or an address (unlimited range, but not position independent). Available for all ARM architectures.

 Copyright © 2005 ARM Limited. All rights reserved.

4.11.1 ADR pseudo-instruction

Load a program-relative or register-relative address into a register.

Syntax

`ADR{cond}{.W} register,label`

where:

cond is an optional condition code (see *Conditional execution* on page 2-16).

`.w` is an optional instruction width specifier. See *ADR in Thumb-2* on page 4-102 for details.

register is the register to load.

label is a program-relative or register-relative expression. See *Register-relative and program-relative expressions* on page 3-29 for more information.

Usage

`ADR` always assembles to one instruction. The assembler attempts to produce a single `ADD` or `SUB` instruction to load the address. If the address cannot be constructed in a single instruction, an error is generated and the assembly fails.

`ADR` produces position-independent code, because the address is program-relative or register-relative.

Use the `ADRL` pseudo-instruction to assemble a wider range of effective addresses.

If *label* is program-relative, it must evaluate to an address in the same assembler area as the `ADR` pseudo-instruction, see *AREA* on page 6-66.

Range

The available range depends on the instruction set in use:

ARM 255 bytes to a byte or halfword-aligned address 1020 bytes to a word-aligned address.

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

32-bit Thumb-2 4095 bytes to a byte, halfword, or word-aligned address.

Copyright © 2005 ARM Limited. All rights reserved.

The given range is relative to a point two words after the address of the current instruction. In ARM and Thumb-2, more distant addresses can be in range if the alignment is 16-bytes or more relative to this point.

ADR in Thumb-2

You can use the `.W` width specifier to force `ADR` to generate a 32-bit instruction in Thumb-2 code.

`ADR.W` always generates a 32-bit instruction, even if the address could be generated in a 16-bit `ADD` or `SUB`.

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

 Copyright © 2005 ARM Limited. All rights reserved.

4.11.2 ADRL pseudo-instruction

Load a program-relative or register-relative address into a register. It is similar to the ADR pseudo-instruction. ADRL can load a wider range of addresses than ADR because it generates two data processing instructions.

Note

ADRL is not available when assembling 16-bit Thumb instructions. Use it only in ARM or Thumb-2 code.

Syntax

ADRL{*cond*} *register*,*label*

where:

cond is an optional condition code (see *Conditional execution* on page 2-16).

register is the register to load.

label is a program-relative or register-relative expression. See *Register-relative and program-relative expressions* on page 3-29 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 4-105 for information on loading a wider range of addresses (see also *Loading constants into registers* on page 2-24).

ADRL produces position-independent code, because the address is program-relative or register-relative.

If *label* is program-relative, it must evaluate to an address in the same assembler area as the ADRL pseudo-instruction, see *AREA* on page 6-66.

Copyright © 2005 ARM Limited. All rights reserved.

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.

16-bit Thumb `ADRL` is not available.

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

The given range is relative to a point two words after the address of the current instruction. In ARM and Thumb-2, more distant addresses can be in range if the alignment is 16-bytes or more relative to this point.

Copyright © 2005 ARM Limited. All rights reserved.

4.11.3 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 4-5 for information on the LDR *instruction*.

Also, see *Loading with LDR Rd, =const* on page 2-27, for information on loading constants with the LDR pseudo-instruction.

Syntax

```
LDR{cond}{.w} register,=[expr | label-expr]
```

where:

cond is an optional condition code (see *Conditional execution* on page 2-16).

.w is an optional instruction width specifier. See *LDR in Thumb-2* on page 4-106 for details.

register 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 © 2005 ARM Limited. All rights reserved.

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.

RVDK uses an ELF proprietary file format called *ARM Toolkit Proprietary ELF* (ATPE). The file format for each version of RVDK is restricted to the proprietary ATPE format for the permitted device. This is referred to as *ATPE_Custom*.

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 6-16 for more information.

See *Loading constants into registers* on page 2-24 for a more detailed explanation of how to use LDR, and for more information on MOV and MVN.

Architectures

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

For 32-bit Thumb-2, see *LDR in Thumb-2*.

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

LDR in Thumb-2

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

LDR.W always generates a 32-bit instruction, even if the constant could be loaded in a 16-bit MOV, or there is a literal pool within reach of a 16-bit pc-relative load.

LDR without .W always generates a 16-bit instruction in Thumb code, even if that results in a 16-bit pc-relative load for a constant that could be generated in a 32-bit MOV or MVN instruction.

 Copyright © 2005 ARM Limited. All rights reserved.

Examples

```
LDR     r3,=0xff0    ; loads 0xff0 into r3
                     ; =>  MOV r3,#0xff0
LDR     r1,=0xfff    ; loads 0xfff into r1
                     ; =>  LDR r1,[pc,offset_to_litpool]
                     ;     ...
                     ;     litpool DCD 0xfff
LDR     r2,=place    ; loads the address of
                     ; place into r2
                     ; =>  LDR r2,[pc,offset_to_litpool]
                     ;     ...
                     ;     litpool DCD place
```


Copyright © 2005 ARM Limited. All rights reserved.

 Copyright © 2005 ARM Limited. All rights reserved.

Chapter 5
Vector Floating-point Programming

This chapter provides reference information about programming the Vector Floating-point coprocessor in assembly language. It contains the following sections:

- *The vector floating-point coprocessor* on page 5-4
- *Floating-point registers* on page 5-5
- *Vector and scalar operations* on page 5-7
- *VFP and condition codes* on page 5-8
- *VFP system registers* on page 5-10
- *Flush-to-zero mode* on page 5-13
- *VFP instructions* on page 5-15
- *VFP directives and vector notation* on page 5-36.

See Table 5-1 on page 5-2 for locations of descriptions of individual instructions.

Note

Vector floating-point hardware is not supported.

Copyright © 2005 ARM Limited. All rights reserved.

Table 5-1 Location of descriptions of VFP instructions

Mnemonic	Brief description	Page	Operation	Architecture
FABS	Absolute value	page 5-16	Vector	All
FADD	Add	page 5-17	Vector	All
FCMP	Compare	page 5-18	Scalar	All
FCPY	Copy	page 5-16	Vector	All
FCVTDS	Convert single-precision to double-precision	page 5-19	Scalar	All
FCVTSD	Convert double-precision to single-precision	page 5-20	Scalar	All
FDIV	Divide	page 5-21	Vector	All
FLD	Load	page 5-22	Scalar	All
FLDM	Load multiple	page 5-24	-	All
FMAC	Multiply-accumulate	page 5-26	Scalar	All
FMDHR, FMDLR	Transfer from one ARM® register to half of double-precision	page 5-28	Scalar	All
FMDRR	Transfer from two ARM registers to double-precision	page 5-27	Scalar	VFPv2
FMRDH, FMRDL	Transfer from half of double-precision to ARM register	page 5-28	Scalar	All
FMRRD	Transfer from double-precision to two ARM registers	page 5-27	Scalar	VFPv2
FMRRS	Transfer between two ARM registers and two single-precision	page 5-30	Scalar	VFPv2
FMRS	Transfer from single-precision to ARM register	page 5-29	Scalar	All
FMRX	Transfer from VFP system register to ARM register	page 5-31	-	All
FMSC	Multiply-subtract	page 5-26	Vector	All
FMSR	Transfer from ARM register to single-precision	page 5-29	Scalar	All
FMSRR	Transfer between two ARM registers and two single-precision	page 5-30	Scalar	VFPv2
FMSTAT	Transfer VFP status flags to ARM CPSR status flags	page 5-31	-	All
FMUL	Multiply	page 5-32	Vector	All
FMXR	Transfer from ARM register to VFP system register	page 5-31	-	All

Copyright © 2005 ARM Limited. All rights reserved.

Table 5-1 Location of descriptions of VFP instructions (continued)

Mnemonic	Brief description	Page	Operation	Architecture
FNEG	Negate	page 5-16	Vector	All
FNMAC	Negate-multiply-accumulate	page 5-26	Vector	All
FNMSC	Negate-multiply-subtract	page 5-26	Vector	All
FNMUL	Negate-multiply	page 5-32	Vector	All
FSITO	Convert signed integer to floating-point	page 5-33	Scalar	All
FSQRT	Square Root	page 5-34	Vector	All
FST	Store	page 5-22	Scalar	All
FSTM	Store multiple	page 5-24	-	All
FSUB	Subtract	page 5-17	Vector	All
FTOSI, FTOUI	Convert floating-point to signed or unsigned integer	page 5-35	Scalar	All
FUITO	Convert unsigned integer to floating-point	page 5-33	Scalar	All

Copyright © 2005 ARM Limited. All rights reserved.

5.1 The vector floating-point coprocessor

The *Vector Floating-Point* (VFP) coprocessor, together with associated support code, provides single-precision and double-precision floating-point arithmetic, as defined by *ANSI/IEEE Std. 754-1985 IEEE Standard for Binary Floating-Point Arithmetic*. This document is referred to as the IEEE 754 standard in this chapter. There is a summary of the standard in the floating-point chapter in *RealView Developer Kit v2.2 Compiler and Libraries Guide*.

Short vectors of up to eight single-precision or four double-precision numbers are handled particularly efficiently. Most arithmetic instructions can be used on these vectors, enabling single-instruction, multiple-data (SIMD) parallelism. In addition, the floating-point load and store instructions have multiple register forms, enabling vectors to be transferred to and from memory efficiently.

For more details of the vector floating-point coprocessor, see *ARM Architecture Reference Manual*.

Note

RVDK v2.2 does not support any floating-point hardware.

5.1.1 VFP architectures

There are two versions of the VFP architecture. VFPv2 has all the instructions that VFPv1 has, and four additional instructions.

The additional instructions enable you to transfer two 32-bit words between ARM registers and VFP registers with one instruction.

Note

Support for VFPv1 is obsolete. The default is VFPv2.

 Copyright © 2005 ARM Limited. All rights reserved.

5.2 Floating-point registers

The Vector Floating-point 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)`.

You can use:

- some registers for single-precision values at the same time as you are using others for double-precision values
- the same registers for single-precision values and double-precision values at different times.

Do not attempt to use corresponding single-precision and double-precision registers at the same time. No damage is caused but the results are meaningless.

5.2.1 Register banks

The VFP registers are arranged as four banks of:

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

See Figure 5-1 for further clarification.

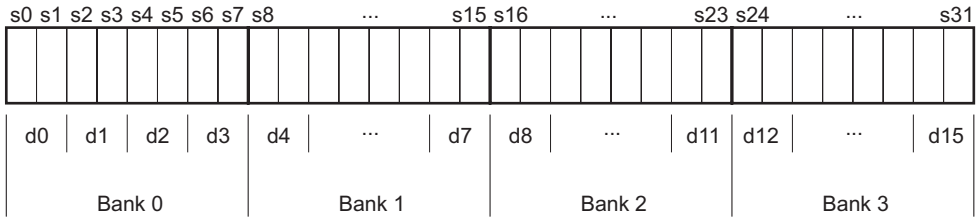

Figure 5-1 VFP register banks

Copyright © 2005 ARM Limited. All rights reserved.

5.2.2 Vectors

A vector can use up to eight single-precision registers, or four double-precision registers, from the same bank. The number of registers used by a vector is controlled by the LEN bits in the FPSCR (see *FPSCR, the floating-point status and control register* on page 5-10).

A vector can start from any register. The first register used by a vector is specified in the register fields in the individual instructions.

Vector wrap-around

If the vector extends beyond the end of a bank, it wraps around to the beginning of the same bank, for example:

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

A vector cannot contain registers from more than one bank.

Vector stride

Vectors can occupy consecutive registers, as in the examples above, or they can occupy alternate registers. This is controlled by the STRIDE bits in the FPSCR (see *FPSCR, the floating-point status and control register* on page 5-10). For example:

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

Restriction on vector length

A vector cannot use the same register twice. Enabling for vector wrap-around, this means that you cannot have:

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

 Copyright © 2005 ARM Limited. All rights reserved.

5.3 Vector and scalar operations

You can use VFP arithmetic instructions to operate:

- on scalars
- on vectors
- on scalars and vectors together.

Use the LEN bits in the FPSCR to control the length of vectors (see *FPSCR, the floating-point status and control register* on page 5-10).

When LEN is 1 all operations are scalar.

5.3.1 Control of scalar, vector and mixed operations

When LEN is greater than 1, the behavior of arithmetic operations depends on which register bank the destination and operand registers are in (see *Register banks* on page 5-5).

The behavior of instructions of the following general forms:

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

is as follows:

- If *Fd* is in the first bank of registers, s0 to s7 or d0 to d3, the operation is scalar.
- If the *Fm* is in the first bank of registers, but *Fd* is not, the operation is mixed.
- If neither *Fd* nor *Fm* are in the first bank of registers, the operation is vector.

Scalar operations

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

Vector operations

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

Mixed scalar and vector operations

For single-operand instructions, *Op* acts on the single value in *Fm*. LEN copies of the result are placed in the vector starting at *Fd*.

For multiple-operand instructions, *Op* acts on the single value in *Fm*, together with the values in the vector starting at *Fn*. The results are placed in the vector starting at *Fd*.

Copyright © 2005 ARM Limited. All rights reserved.

5.4 VFP and condition codes

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

In Thumb® state on a Thumb-2 processor, you can use an `IT` instruction to set condition codes on up to four following VFP instructions.

Note

None of the processors supported by this toolkit supports Thumb-2.

The only VFP instruction that can be used to update the status flags is `FCMP`. It does not update the flags in the CPSR directly, but updates a separate set of flags in the FPSCR (see *FPSCR, the floating-point status and control register* on page 5-10).

Note

To use these flags to control conditional instructions, including conditional VFP instructions, you must first copy them into the CPSR using an `FMSTAT` instruction (see *FMRX, FMXR, and FMSTAT* on page 5-31).

Following an `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 needed
- *Not-a-Number* (NaN) values have no ordering relationship with numbers or with each other, so additional conditions are required to account for *unordered* results.

The meanings of the condition code mnemonics are shown in Table 5-2 on page 5-9.

Copyright © 2005 ARM Limited. All rights reserved.

Table 5-2 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 CPSR determines the meaning of condition codes.

Copyright © 2005 ARM Limited. All rights reserved.

5.5 VFP system registers

Three VFP system registers are accessible to you in all implementations of VFP:

- *FPSCR, the floating-point status and control register*
- *FPEXC, the floating-point exception register* on page 5-12
- *FPSID, the floating-point system ID register* on page 5-12.

A particular implementation of VFP can have additional registers (see the technical reference manual for the VFP coprocessor you are using).

5.5.1 FPSCR, the floating-point status and control register

The FPSCR contains all the user-level VFP status and control bits:

bits[31:28] are the N, Z, C, and V flags. These are the VFP status flags. They cannot be used to control conditional execution until they have been copied into the status flags in the CPSR (see *VFP and condition codes* on page 5-8).

bit[24] is the flush-to-zero mode control bit:

0 flush-to-zero mode is disabled.

1 flush-to-zero mode is enabled.

Flush-to-zero mode can provide greater performance, depending on your hardware and software, at the expense of loss of range (see *Flush-to-zero mode* on page 5-13).

Note

Flush-to-zero mode must not be used when IEEE 754 compatibility is a requirement.

bits[23:22] control rounding mode as follows:

0b00 *Round to Nearest* (RN) mode

0b01 *Round towards Plus infinity* (RP) mode

0b10 *Round towards Minus infinity* (RM) mode

0b11 *Round towards Zero* (RZ) mode.

bits[21:20] STRIDE is the distance between successive values in a vector (see *Vectors* on page 5-6). Stride is controlled as follows:

0b00 stride = 1

0b11 stride = 2.

 Copyright © 2005 ARM Limited. All rights reserved.

bits[18:16] LEN is the number of registers used by each vector (see *Vectors* on page 5-6). It is 1 + the value of bits[18:16]:

0b000 LEN = 1

.

.

0b111 LEN = 8.

bits[12:8] are the exception trap enable bits:

IXE inexact exception enable

UFE underflow exception enable

OFE overflow exception enable

DZE division by zero exception enable

IOE invalid operation exception enable.

This Guide does not cover the use of floating-point exception trapping. For information see the technical reference manual for the VFP coprocessor you are using.

bits[4:0] are the cumulative exception bits:

IXC inexact exception

UFC underflow exception

OFC overflow exception

DZC division by zero exception

IOC invalid operation exception.

Cumulative exception bits are set when the corresponding exception occurs. They remain set until you clear them by writing directly to the FPSCR.

all other bits are unused in the basic VFP specification. They can be used in particular implementations (see the technical reference manual for the VFP coprocessor you are using). Do not modify these bits except in accordance with any use in a particular implementation.

To alter some bits without affecting other bits, use a read-modify-write procedure (see *Modifying individual bits of a VFP system register* on page 5-12).

Copyright © 2005 ARM Limited. All rights reserved.

5.5.2 FPEXC, the floating-point exception register

You can only access the FPEXC in privileged modes. It contains the following bits:

bit[31] is the EX bit. You can read it in all VFP implementations. In some implementations you might also be able to write to it.

If the value is 0, the only significant state in the VFP system is the contents of the general purpose registers plus FPSCR and FPEXC.

If the value is 1, you need implementation-specific information to save state (see the technical reference manual for the VFP coprocessor you are using).

bit[30] is the EN bit. You can read and write it in all VFP implementations.

If the value is 1, the VFP coprocessor is enabled and operates normally.

If the value is 0, the VFP coprocessor is disabled. When the coprocessor is disabled, you can read or write the FPSID or FPEXC registers, but other VFP instructions are treated as Undefined Instructions.

bits[29:0] might be used by particular implementations of VFP. You can use all the VFP functions described in this chapter without accessing these bits.

You must not alter these bits except in accordance with their use in a particular implementation (see the technical reference manual for the VFP coprocessor you are using).

To alter some bits without affecting other bits, use a read-modify-write procedure (see *Modifying individual bits of a VFP system register*).

5.5.3 FPSID, the floating-point system ID register

The FPSID is a read-only register. You can read it to find out which implementation of the VFP architecture your program is running on.

5.5.4 Modifying individual bits of a VFP system register

To alter some bits of a VFP system register without affecting other bits, use a read-modify-write procedure similar to the following example:

```
FMRX    r10,FPSCR           ; copy FPSCR into r10
BIC     r10,r10,#0x00370000 ; clears STRIDE and LEN
ORR     r10,r10,#0x00030000 ; sets STRIDE = 1, LEN = 4
FMXR    FPSCR,r10           ; copy r10 back into FPSCR
```

See *FMRX, FMXR, and FMSTAT* on page 5-31.

 Copyright © 2005 ARM Limited. All rights reserved.

5.6 Flush-to-zero mode

Some implementations of VFP use support code to handle denormalized numbers. The performance of such systems, in calculations involving denormalized numbers, is much less than it is in normal calculations.

Flush-to-zero mode replaces denormalized numbers with +0. This does not comply with IEEE 754 arithmetic, but in some circumstances can improve performance considerably.

5.6.1 When to use flush-to-zero mode

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

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

You can change between flush-to-zero and normal mode at any time, if different parts of your code have different requirements. Numbers already in registers are not affected by changing mode.

5.6.2 The effects of using flush-to-zero mode

With certain exceptions (see *Operations not affected by flush-to-zero mode* on page 5-14), flush-to-zero mode has the following effects on floating-point operations:

- A denormalized number is treated as +0 when used as an input to a floating point operation. The source register is not altered.
- If the result of a single-precision floating-point operation, before rounding, is in the range -2^{-126} to $+2^{-126}$, it is replaced by +0.
- If the result of a double-precision floating-point operation, before rounding, is in the range -2^{-1022} to $+2^{-1022}$, it is replaced by +0.

An inexact exception occurs whenever a denormalized number is used as an operand, or a result is flushed to zero. Underflow exceptions do not occur in flush-to-zero mode.

Copyright © 2005 ARM Limited. All rights reserved.

5.6.3 Operations not affected by flush-to-zero mode

The following operations can be carried out on denormalized numbers even in flush-to-zero mode, without flushing the results to zero:

- Copy, absolute value, and negate (see *FABS, FCPY, and FNEG* on page 5-16)
- Load and store (see *FLD and FST* on page 5-22)
- Load multiple and store multiple (see *FLDM and FSTM* on page 5-24)
- Transfer between floating-point registers and ARM general-purpose registers (see *FMDRR and FMRRD* on page 5-27 and *FMRRS and FMSRR* on page 5-30).

 Copyright © 2005 ARM Limited. All rights reserved.

5.7 VFP instructions

This section contains the following subsections:

Copyright © 2005 ARM Limited. All rights reserved.

5.7.1 FABS, FCPY, and FNEG

Floating-point copy, absolute value, and negate.

These instructions can be scalar, vector, or mixed (see *Vector and scalar operations* on page 5-7).

Syntax

`<op><precision>{cond} Fd, Fm`

where:

`<op>`	must be one of `FCPY`, `FABS`, or `FNEG`.
`<precision>`	must be either `S` for single-precision, or `D` for double-precision.
`cond`	is an optional condition code (see *VFP and condition codes* on page 5-8).
`Fd`	is the VFP register for the result.
`Fm`	is the VFP register holding the operand.

The precision of `Fd` and `Fm` must match the precision specified in `<precision>`.

Usage

The `FCPY` instruction copies the contents of `Fm` into `Fd`.

The `FABS` instruction takes the contents of `Fm`, clears the sign bit, and places the result in `Fd`. This gives the absolute value.

The `FNEG` instruction takes the contents of `Fm`, changes the sign bit, and places the result in `Fd`. This gives the negation of the value.

If the operand is a NaN, the sign bit is determined in each case as above, but no exception is produced.

Exceptions

None of these instructions can produce any exceptions.

Examples

```
FABSD   d3, d5
FNEGSMI s15, s15
```

 Copyright © 2005 ARM Limited. All rights reserved.

5.7.2 FADD and FSUB

Floating-point add and subtract.

FADD and FSUB can be scalar, vector, or mixed (see *Vector and scalar operations* on page 5-7).

Syntax

FADD<*precision*>{*cond*} *Fd*, *Fn*, *Fm*

FSUB<*precision*>{*cond*} *Fd*, *Fn*, *Fm*

where:

<*precision*>	must be either S for single-precision, or D for double-precision.
cond	is an optional condition code (see *VFP and condition codes* on page 5-8).
Fd	is the VFP register for the result.
Fn	is the VFP register holding the first operand.
Fm	is the VFP register holding the second operand.

The precision of *Fd*, *Fn* and *Fm* must match the precision specified in <*precision*>.

Usage

The FADD instruction adds the values in *Fn* and *Fm* and places the result in *Fd*.

The FSUB instruction subtracts the value in *Fm* from the value in *Fn* and places the result in *Fd*.

Exceptions

FADD and FSUB instructions can produce Invalid Operation, Overflow, or Inexact exceptions.

Examples

```
FSUBSEQ     s2, s4, s17
FADDDGT     d4, d0, d12
FSUBD       d0, d0, d12
```


Copyright © 2005 ARM Limited. All rights reserved.

5.7.3 FCMP

Floating-point compare.

FCMP is always scalar.

Syntax

FCMP{E}<*precision*>{*cond*} *Fd*, *Fm*

FCMP{E}Z<*precision*>{*cond*} *Fd*

where:

E — is an optional parameter. If E is present, an exception is raised if either operand is any kind of NaN. Otherwise, an exception is raised only if either operand is a signaling NaN.

Z — is a parameter specifying comparison with zero.

<*precision*> — must be either S for single-precision, or D for double-precision.

cond — is an optional condition code (see *VFP and condition codes* on page 5-8).

Fd — is the VFP register holding the first operand.

Fm — is the VFP register holding the second operand. Omit *Fm* for a compare with zero instruction.

The precision of *Fd* and *Fm* must match the precision specified in <*precision*>.

Usage

The FCMP instruction subtracts the value in *Fm* from the value in *Fd* and sets the VFP condition flags on the result (see *VFP and condition codes* on page 5-8).

Exceptions

FCMP instructions can produce Invalid Operation exceptions.

Examples

```
FCMPS       s3, s0
FCMPEDNE    d5, d13
FCMPZSEQ    s2
```


Copyright © 2005 ARM Limited. All rights reserved.

5.7.4 FCVTDS

Convert single-precision floating-point to double-precision.

FCVTDS is always scalar.

Syntax

FCVTDS{*cond*} *Dd*, *Sm*

where:

cond is an optional condition code (see *VFP and condition codes* on page 5-8).

Dd is a double-precision VFP register for the result.

Sm is a single-precision VFP register holding the operand.

Usage

The FCVTDS instruction converts the single-precision value in *Sm* to double-precision and places the result in *Dd*.

Exceptions

FCVTDS instructions can produce Invalid Operation exceptions.

Examples

```
FCVTDS      d5, s7
FCVTDSGT    d0, s4
```


Copyright © 2005 ARM Limited. All rights reserved.

5.7.5 FCVTSD

Convert double-precision floating-point to single-precision.

FCVTSD is always scalar.

Syntax

FCVTSD{*cond*} *Sd, Dm*

where:

cond	is an optional condition code (see *VFP and condition codes* on page 5-8).
Sd	is a single-precision VFP register for the result.
Dm	is a double-precision VFP register holding the operand.

Usage

The FCVTSD instruction converts the double-precision value in *Dm* to single-precision and places the result in *Sd*.

Exceptions

FCVTSD instructions can produce Invalid Operation, Overflow, Underflow, or Inexact exceptions.

Examples

```
FCVTSD      s3, d14
FCVTSDMI    s0, d1
```

 Copyright © 2005 ARM Limited. All rights reserved.

5.7.6 FDIV

Floating-point divide. FDIV can be scalar, vector, or mixed (see *Vector and scalar operations* on page 5-7).

Syntax

FDIV<*precision*>{*cond*} *Fd*, *Fn*, *Fm*

where:

<*precision*>	must be either S for single-precision, or D for double-precision.
cond	is an optional condition code (see *VFP and condition codes* on page 5-8).
Fd	is the VFP register for the result.
Fn	is the VFP register holding the first operand.
Fm	is the VFP register holding the second operand.

The precision of *Fd*, *Fn* and *Fm* must match the precision specified in <*precision*>.

Usage

The FDIV instruction divides the value in *Fn* by the value in *Fm* and places the result in *Fd*.

Exceptions

FDIV operations can produce Division by Zero, Invalid Operation, Overflow, Underflow, or Inexact exceptions.

Examples

```
FDIVS       s8, s0, s12
FDIVSNE     s2, s27, s28
FDIVD       d10, d2, d10
```


Copyright © 2005 ARM Limited. All rights reserved.

5.7.7 FLD and FST

Floating-point load and store.

Syntax

FLD<*precision*>{*cond*} *Fd*, [*Rn*{, #*offset*}]

FST<*precision*>{*cond*} *Fd*, [*Rn*{, #*offset*}]

FLD<*precision*>{*cond*} *Fd*, *label*

FST<*precision*>{*cond*} *Fd*, *label*

where:

<*precision*> must be either S for single-precision, or D for double-precision.

cond is an optional condition code (see *VFP and condition codes* on page 5-8).

Fd is the VFP register to be loaded or saved. The precision of *Fd* must match the precision specified in <*precision*>.

Rn is the ARM register holding the base address for the transfer.

offset is an optional numeric expression. It must evaluate to a numeric constant at assembly time. The value must be a multiple of 4, and lie in the range –1020 to +1020. The value is added to the base address to form the address used for the transfer.

label is a program-relative expression. See *Register-relative and program-relative expressions* on page 3-29 for more information.

label must be within ±1KB of the current instruction.

Usage

The FLD instruction loads a floating-point register from memory. The FST instruction saves the contents of a floating-point register to memory.

One word is transferred if <*precision*> is S. Two words are transferred if <*precision*> is D.

Examples

```
FLDD    d5, [r7, #-12]
FLDSNE  s3, [r2, #72+count]
FSTS    s2, [r5]
```

 Copyright © 2005 ARM Limited. All rights reserved.

```
FLDD    d2, [r15, #addr-{PC}]
FLDS    s9, fpconst
```


Copyright © 2005 ARM Limited. All rights reserved.

5.7.8 FLDM and FSTM

Floating-point load multiple and store multiple.

Syntax

`FLDM<addressmode><precision>{cond} Rn,{!} VFPregisters`

`FSTM<addressmode><precision>{cond} Rn,{!} VFPregisters`

where:

<addressmode> must be one of:

- `IA` meaning Increment address After each transfer.
- `DB` meaning Decrement address Before each transfer.
- `EA` meaning Empty Ascending stack operation. This is the same as `DB` for loads, and the same as `IA` for saves.
- `FD` meaning Full Descending stack operation. This is the same as `IA` for loads, and the same as `DB` for saves.

<precision> must be one of:

- `S` for single-precision
- `D` for double-precision
- `X` for unspecified precision.

cond is an optional condition code (see *VFP and condition codes* on page 5-8).

Rn is the ARM register holding the base address for the transfer.

`!` is optional. `!` specifies that the updated base address must be written back to *Rn*.

Note

If `!` is not specified, *<addressmode>* must be `IA`.

VFPregisters is a list of consecutive floating-point registers enclosed in braces, `{` and `}`. The list can be comma-separated, or in range format. There must be at least one register in the list.

Copyright © 2005 ARM Limited. All rights reserved.

Usage

The FLDM instruction loads several consecutive floating-point registers from memory.

The FSTM instruction saves the contents of several consecutive floating-point registers to memory.

If <*precision*> is specified as D, *VFPregisters* must be a list of double-precision registers, and two words are transferred for each register in the list.

If <*precision*> is specified as S, *VFPregisters* must be a list of single-precision registers, and one word is transferred for each register in the list.

Unspecified precision

If <*precision*> is specified as X, *VFPregisters* must be specified as double-precision registers. However, any or all of the specified double-precision registers can actually contain two single-precision values or integers.

The number of words transferred might be $2n$ or $(2n + 1)$, where n is the number of double-precision registers in the list. This is implementation dependent. However, if writeback is specified, *Rn* is always adjusted by $(2n + 1)$ words.

You must only use unspecified-precision loads and saves in matched pairs, to save and restore data. The format of the saved data is implementation-dependent.

Examples

```
FLDMIAS r2, {s1-s5}
FSTMFDD r13!, {d3-d6}
FSTMIAS r0!, {s31}
```

The following instructions are equivalent:

```
FLDMIAS r7, {s3-s7}
FLDMIAS r7, {s3,s4,s5,s6,s7}
```

The following instructions must always be used as a matching pair:

```
FSTMFDX r13!, {d0-d3}
FLDMFDX r13!, {d0-d3}
```

The following instruction is illegal, as the registers in the list are not consecutive:

```
FLDMIAD r13!, {d0,d2,d3}
```


Copyright © 2005 ARM Limited. All rights reserved.

5.7.9 FMAC, FNMAC, FMSC, and FNMSC

Floating-point multiply-accumulate, negate-multiply-accumulate, multiply-subtract and negate-multiply-subtract. These instructions can be scalar, vector, or mixed (see *Vector and scalar operations* on page 5-7).

Syntax

`<op><precision>{cond} Fd, Fn, Fm`

where:

`<op>`	must be one of `FMAC`, `FNMAC`, `FMSC`, or `FNMSC`.
`<precision>`	must be either `S` for single-precision, or `D` for double-precision.
`cond`	is an optional condition code (see *VFP and condition codes* on page 5-8).
`Fd`	is the VFP register for the result.
`Fn`	is the VFP register holding the first operand.
`Fm`	is the VFP register holding the second operand.

The precision of `Fd`, `Fn` and `Fm` must match the precision specified in `<precision>`.

Usage

The `FMAC` instruction calculates `Fd + Fn * Fm` and places the result in `Fd`.

The `FNMAC` instruction calculates `Fd – Fn * Fm` and places the result in `Fd`.

The `FMSC` instruction calculates `–Fd + Fn * Fm` and places the result in `Fd`.

The `FNMSC` instruction calculates `–Fd – Fn * Fm` and places the result in `Fd`.

Exceptions

These operations can produce Invalid Operation, Overflow, Underflow, or Inexact exceptions.

Examples

```
FMACD       d8, d0, d8
FMACS       s20, s24, s28
FNMSCSLE    s6, s0, s26
```


Copyright © 2005 ARM Limited. All rights reserved.

5.7.10 FMDRR and FMRRD

Transfer contents between two ARM registers and a double-precision floating-point register.

Syntax

`FMDRR{`*cond*`} Dn, Rd, Rn`

`FMRRD{`*cond*`} Rd, Rn, Dn`

where:

cond	is an optional condition code (see *VFP and condition codes* on page 5-8).
Dn	is the VFP double-precision register.
Rd, Rn	are ARM registers. Do not use r15.

Usage

`FMDRR Dn, Rd, Rn` transfers the contents of *Rd* into the low half of *Dn*, and the contents of *Rn* into the high half of *Dn*.

`FMRRD Rd, Rn, Dn` transfers the contents of the low half of *Dn* into *Rd*, and the contents of the high half of *Dn* into *Rn*.

Exceptions

These instructions do not produce any exceptions.

Architectures

These instructions are available in VFPv2 and above.

Examples

```
FMDRR   d5, r3, r4
FMRRDPL r12, r2, d2
```


Copyright © 2005 ARM Limited. All rights reserved.

5.7.11 FMDHR, FMDLR, FMRDH, and FMRDL

Transfer contents between an ARM register and a half of a double-precision floating-point register.

Syntax

```
FMDHR{cond} Dn, Rd
FMDLR{cond} Dn, Rd
FMRDH{cond} Rd, Dn
FMRDL{cond} Rd, Dn
```

where:

cond is an optional condition code (see *VFP and condition codes* on page 5-8).

Dn is the VFP double-precision register.

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

Usage

These instructions are used together as matched pairs:

- Use FMDHR with FMDLR
  - FMDHR copy the contents of *Rd* into the high half of *Dn*
  - FMDLR copy the contents of *Rd* into the low half of *Dn*
- Use FMRDH with FMRDL
  - FMRDH copy the contents of the high half of *Dn* into *Rd*
  - FMRDL copy the contents of the low half of *Dn* into *Rd*.

Exceptions

These instructions do not produce any exceptions.

Examples

```
FMDHR   d5, r3
FMDLR   d5, r12
FMRDH   r5, d3
FMRDL   r9, d3
FMDLRPL d2, r1
```


Copyright © 2005 ARM Limited. All rights reserved.

5.7.12 FMRS and FMSR

Transfer contents between a single-precision floating-point register and an ARM register.

Syntax

`FMRS{cond} Rd, Sn`

`FMSR{cond} Sn, Rd`

where:

cond is an optional condition code (see *VFP and condition codes* on page 5-8).

Sn is the VFP single-precision register.

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

Usage

The `FMRS` instruction transfers the contents of *Sn* into *Rd*.

The `FMSR` instruction transfers the contents of *Rd* into *Sn*.

Exceptions

These instructions do not produce any exceptions.

Examples

```
FMRS        r2, s0
FMSRNE      s30, r5
```


Copyright © 2005 ARM Limited. All rights reserved.

5.7.13 FMRRS and FMSRR

Transfer contents between two consecutive single-precision floating-point registers and two ARM registers.

Syntax

`FMRRS{cond} Rd, Rn, {Sm,Sm'}`

`FMSRR{cond} {Sm,Sm'}, Rd, Rn`

where:

cond	is an optional condition code (see *VFP and condition codes* on page 5-8).
Sm	is a VFP single-precision register.
Sm'	is the next VFP single-precision registers after *Sm*.
Rd, *Rn*	are the ARM registers. Do not use r15.

Usage

The FMRRS instruction transfers the contents of *Sm* into *Rd*, and the contents of *Sm'* into *Rn*.

The FMSRR instruction transfers the contents of *Rd* into *Sm*, and the contents of *Rn* into *Sm'*.

Exceptions

These instructions do not produce any exceptions.

Architectures

These instructions are available in VFPv2 and above.

Examples

```
FMRRS       r2, r3, {s0,s1}
FMSRRNE     {s27,s28}, r5, r2
```

Incorrect examples

```
FMRRS       r2, r3, {s2,s4}     ; VFP registers must be consecutive
FMSRR       {s5,s6}, r15, r0    ; you must not use r15
```


Copyright © 2005 ARM Limited. All rights reserved.

5.7.14 FMRX, FMXR, and FMSTAT

Transfer contents between an ARM register and a VFP system register.

Syntax

`FMRX{cond} Rd, VFPsysreg`

`FMXR{cond} VFPsysreg, Rd`

`FMSTAT{cond}`

where:

cond is an optional condition code (see *VFP and condition codes* on page 5-8).

VFPsysreg is the VFP system register, usually FPSCR, FPSID, or FPEXC (see *Floating-point registers* on page 5-5).

Rd is the ARM register.

Usage

The FMRX instruction transfers the contents of *VFPsysreg* into *Rd*.

The FMXR instruction transfers the contents of *Rd* into *VFPsysreg*.

The FMSTAT instruction is a synonym for `FMRX r15, FPSCR`. It transfers the floating-point condition flags to the corresponding flags in the ARM CPSR (see *VFP and condition codes* on page 5-8).

Note

These instructions stall the ARM until all current VFP operations complete.

Exceptions

These instructions do not produce any exceptions.

Examples

```
FMSTAT
FMSTATNE
FMXR        FPSCR, r2
FMRX        r3, FPSID
```


Copyright © 2005 ARM Limited. All rights reserved.

5.7.15 FMUL and FNMUL

Floating-point multiply and negate-multiply. FMUL and FNMUL can be scalar, vector, or mixed (see *Vector and scalar operations* on page 5-7).

Syntax

FMUL<*precision*>{*cond*} *Fd*, *Fn*, *Fm*

FNMUL<*precision*>{*cond*} *Fd*, *Fn*, *Fm*

where:

<*precision*>	must be either S for single-precision, or D for double-precision.
cond	is an optional condition code (see *VFP and condition codes* on page 5-8).
Fd	is the VFP register for the result.
Fn	is the VFP register holding the first operand.
Fm	is the VFP register holding the second operand.

The precision of *Fd*, *Fn* and *Fm* must match the precision specified in <*precision*>.

Usage

The FMUL instruction multiplies the values in *Fn* and *Fm* and places the result in *Fd*.

The FNMUL instruction multiplies the values in *Fn* and *Fm* and places the negation of the result in *Fd*.

Exceptions

FMUL and FNMUL operations can produce Invalid Operation, Overflow, Underflow, or Inexact exceptions.

Examples

```
FNMULS      s10, s10, s14
FMULDLT     d0, d7, d8
```

 Copyright © 2005 ARM Limited. All rights reserved.

5.7.16 FSITO and FUITO

Convert signed integer to floating-point and unsigned integer to floating-point.

FSITO and FUITO are always scalar.

Syntax

FSITO<*precision*>{*cond*} *Fd*, *Sm*

FUITO<*precision*>{*cond*} *Fd*, *Sm*

where:

<*precision*> must be either S for single-precision, or D for double-precision.

cond is an optional condition code (see *VFP and condition codes* on page 5-8).

Fd is a VFP register for the result. The precision of *Fd* must match the precision specified in <*precision*>.

Sm is a single-precision VFP register holding the integer operand.

Usage

The FSITO instruction converts the signed integer value in *Sm* to floating-point and places the result in *Fd*.

The FUITO instruction converts the unsigned integer value in *Sm* to floating-point and places the result in *Fd*.

Exceptions

FSITOS and FUITOS instructions can produce Inexact exceptions.

FSITOD and FUITOD instructions do not produce any exceptions.

Examples

```
FUITOD      d3, s31 ; unsigned integer to double-precision
FSITOD      d5, s16 ; signed integer to double-precision
FSITOSNE    s2, s2  ; signed integer to single-precision
```


Copyright © 2005 ARM Limited. All rights reserved.

5.7.17 FSQRT

Floating-point square root instruction. This instruction can be scalar, vector, or mixed (see *Vector and scalar operations* on page 5-7).

Syntax

FSQRT<*precision*>{*cond*} *Fd*, *Fm*

where:

<*precision*>	must be either S for single-precision, or D for double-precision.
cond	is an optional condition code (see *VFP and condition codes* on page 5-8).
Fd	is the VFP register for the result.
Fm	is the VFP register holding the operand.

The precision of *Fd* and *Fm* must match the precision specified in <*precision*>.

Usage

The FSQRT instruction calculates the square root of the value of the contents of *Fm* and places the result in *Fd*.

Exceptions

FSQRT operations can produce Invalid Operation or Inexact exceptions.

Examples

```
FSQRTS      s4, s28
FSQRTD      d14, d6
FSQRTSNE    s15, s13
```


Copyright © 2005 ARM Limited. All rights reserved.

5.7.18 FTOSI and FTOUI

Convert floating-point to signed integer and floating-point to unsigned integer.

FTOSI and FTOUI are always scalar.

Syntax

FTOSI{Z}<*precision*>{*cond*} *Sd*, *Fm*

FTOUI{Z}<*precision*>{*cond*} *Sd*, *Fm*

where:

Z — is an optional parameter specifying rounding towards zero. If specified, this overrides the rounding mode currently specified in the FPSCR. The FPSCR is not altered.

<*precision*> — must be either S for single-precision, or D for double-precision.

cond — is an optional condition code (see *VFP and condition codes* on page 5-8).

Sd — is a single-precision VFP register for the integer result.

Fm — is a VFP register holding the operand. The precision of *Fm* must match the precision specified in <*precision*>.

Usage

The FTOSI instruction converts the floating-point value in *Fm* to a signed integer and places the result in *Sd*.

The FTOUI instruction converts the floating-point value in *Fm* to an unsigned integer and places the result in *Sd*.

Exceptions

FTOSI and FTOUI instructions can produce Invalid Operation or Inexact exceptions.

Examples

```
FTOSID      s10, d2
FTOUID      s3, d1
FTOSIZS     s3, s31
```


Copyright © 2005 ARM Limited. All rights reserved.

5.8 VFP directives and vector notation

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

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

- *VFPASSERT SCALAR* on page 5-37
- *VFPASSERT VECTOR* on page 5-38.

If you use VFPASSERT directives, you must specify vector details in all VFP data processing instructions. The vector notation is described below. If you do not use VFPASSERT directives you must not use this vector notation.

In 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 *DN and SN* on page 6-12).

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

Note

The compilation tools only accept -fpu none or -fpu softvfp (default).

 Copyright © 2005 ARM Limited. All rights reserved.

5.8.1 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 also:

- *VFP directives and vector notation* on page 5-36
- *VFPASSERT VECTOR* on page 5-38.

Note

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

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 © 2005 ARM Limited. All rights reserved.

5.8.2 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:

- *VFP directives and vector notation* on page 5-36
- *VFPASSERT SCALAR* on page 5-37.

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.

 Copyright © 2005 ARM Limited. All rights reserved.

Example

```
FMRX    r10,FPSCR
BIC     r10,r10,#0x00370000
ORR     r10,r10,#0x00020000    ; set length = 3, stride = 1
FMXR    FPSCR,r10

VFPASSERT VECTOR           ; assert vector mode, unspecified length and stride
faddd  d4, d4, d0          ; ERROR, scalar in vector mode
fadds  s16<3>, s0, s8<3>   ; okay
fabss  s24<1>, s28<1>      ; wrong length, but not faulted (unspecified)

FMRX    r10,FPSCR
BIC     r10,r10,#0x00370000
ORR     r10,r10,#0x00030000    ; set length = 4, stride = 1
FMXR    FPSCR,r10

VFPASSERT VECTOR<4>        ; assert vector mode, length 4, stride 1
fadds  s24<4>, s0, s8<4>   ; okay
fabss  s24<2>, s24<2>      ; ERROR, wrong length

FMRX    r10,FPSCR
BIC     r10,r10,#0x00370000
ORR     r10,r10,#0x00130000    ; set length = 4, stride = 2
FMXR    FPSCR,r10

VFPASSERT VECTOR<4:2>      ; assert vector mode, length 4, stride 2
fadds  s8<4>, s0, s16<4>   ; ERROR, wrong stride
fabss  s16<4:2>, s28<4:2>  ; okay
fadds  s8<>, s2, s16<>     ; okay (s8 and s16 both have
                           ; length 4 and stride 2.
                           ; s2 is scalar.)
```


Copyright © 2005 ARM Limited. All rights reserved.

 Copyright © 2005 ARM Limited. All rights reserved.

Chapter 6
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 © 2005 ARM Limited. All rights reserved.

6.1 Alphabetical list of directives

Table 6-1 shows a complete list of the ARM directives. Use it to locate individual directives described in the rest of this chapter.

Table 6-1 Location of descriptions of directives

Directive	Page	Directive	Page	Directive	Page
ALIGN	page 6-64	EQU	page 6-70	KEEP	page 6-80
ARM *and* CODE32	page 6-59	EXPORT *or* GLOBAL	page 6-71	LCLA, LCLL, *and* LCLS	page 6-7
AREA	page 6-66	EXPORTAS	page 6-73	LTORG	page 6-16
ASSERT	page 6-53	EXTERN	page 6-74	MACRO *and* MEND	page 6-30
CN	page 6-10	FIELD	page 6-18	MAP	page 6-17
CODE16	page 6-62	FRAME ADDRESS	page 6-40	MEND *see* MACRO	page 6-30
COMMON	page 6-28	FRAME POP	page 6-41	MEXIT	page 6-33
CP	page 6-11	FRAME PUSH	page 6-42	NOFP	page 6-81
DATA	page 6-28	FRAME REGISTER	page 6-44	OPT	page 6-55
DCB	page 6-20	FRAME RESTORE	page 6-45	PRESERVE8 *see* REQUIRE8	page 6-82
DCD *and* DCDU	page 6-21	FRAME SAVE	page 6-47	PROC *see* FUNCTION	page 6-51
DCDO	page 6-22	FRAME STATE REMEMBER	page 6-48	REQUIRE	page 6-81
DCFD *and* DCFDU	page 6-23	FRAME STATE RESTORE	page 6-49	REQUIRE8 *and* PRESERVE8	page 6-82
DCFS *and* DCFSU	page 6-24	FUNCTION *or* PROC	page 6-51	RLIST	page 6-9
DCI	page 6-25	GBLA, GBLL, *and* GBLS	page 6-5	RN	page 6-84
DCQ *and* DCQU	page 6-26	GET *or* INCLUDE	page 6-76	ROUT	page 6-85
DCW *and* DCWU	page 6-27	GLOBAL *see* EXPORT	page 6-71	SETA, SETL, *and* SETS	page 6-8
DN *and* SN	page 6-12	IF, ELSE, ENDIF, *and* ELIF	page 6-34	SN *see* DN	page 6-12
ELIF, ELSE *see* IF	page 6-34	IMPORT	page 6-77	SPACE	page 6-19
END	page 6-68	INCBIN	page 6-79	THUMB	page 6-60

Copyright © 2005 ARM Limited. All rights reserved.

Table 6-1 Location of descriptions of directives (continued)

Directive	Page	Directive	Page	Directive	Page
ENDFUNC *or* ENDP	page 6-52	INCLUDE *see* GET	page 6-76	TTL *and* SUBT	page 6-57
ENDIF *see* IF	page 6-34	INFO	page 6-54	WHILE *and* WEND	page 6-37
ENTRY	page 6-69				

Copyright © 2005 ARM Limited. All rights reserved.

6.2 Symbol definition directives

This section describes the following directives:

- *GBLA, GBLL, and GBLS* on page 6-5
  Declare a global arithmetic, logical, or string variable.
- *LCLA, LCLL, and LCLS* on page 6-7
  Declare a local arithmetic, logical, or string variable.
- *SETA, SETL, and SETS* on page 6-8
  Set the value of an arithmetic, logical, or string variable.
- *RLIST* on page 6-9
  Define a name for a set of general-purpose registers.
- *CN* on page 6-10
  Define a coprocessor register name.
- *CP* on page 6-11
  Define a coprocessor name.
- *DN and SN* on page 6-12
  Define a double-precision or single-precision VFP register name.
- *FN* on page 6-13
  Define an FPA register name (obsolete).

 Copyright © 2005 ARM Limited. All rights reserved.

6.2.1 GBLA, GBLL, and GBLS

The GBLA directive declares a global arithmetic variable, and initializes its value to 0.

The GBLL directive declares a global logical variable, and initializes its value to {FALSE}.

The GBLS directive declares a global string variable and initializes its value to a null string, "".

Syntax

<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 6-8).

See *LCLA, LCLL, and LCLS* on page 6-7 for information on declaring local variables.

Global variables can also be set with the -predefine assembler command-line option. See *Command syntax* on page 3-2 for more information.

Copyright © 2005 ARM Limited. All rights reserved.

Examples

Example 6-1 declares a variable `objectsize`, sets the value of `objectsize` to `0xFF`, and then uses it later in a `SPACE` directive.

Example 6-1

```
            GBLA    objectsize    ; declare the variable name
objectsize  SETA    0xFF          ; set its value
            .
            .                     ; other code
            .
            SPACE   objectsize    ; quote the variable
```

Example 6-2 shows how to declare and set a variable when you invoke `armasm`. Use this when you need to set the value of a variable at assembly time. `--pd` is a synonym for `--predefine`.

Example 6-2

```
armasm --pd "objectsize SETA 0xFF" --o objectfile sourcefile
```


Copyright © 2005 ARM Limited. All rights reserved.

6.2.2 LCLA, LCLL, and LCLS

The LCLA directive declares a local arithmetic variable, and initializes its value to 0.

The LCLL directive declares a local logical variable, and initializes its value to {FALSE}.

The LCLS directive declares a local string variable, and initializes its value to a null string, "".

Syntax

<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 6-30).

Set the value of the variable with a SETA, SETL, or SETS directive (see *SETA, SETL, and SETS* on page 6-8).

See *GBLA, GBLL, and GBLS* on page 6-5 for information on declaring global variables.

Example

```
        MACRO                           ; Declare a macro
$label  message $a                      ; Macro prototype line
        LCLS    err                     ; Declare local string
                                        ; variable err.
err     SETS    "error no: "            ; Set value of err
$label  ; code
        INFO    0, "err":CC::STR:$a     ; Use string
        MEND
```


Copyright © 2005 ARM Limited. All rights reserved.

6.2.3 SETA, SETL, and SETS

The SETA directive sets the value of a local or global arithmetic variable.

The SETL directive sets the value of a local or global logical variable.

The SETS directive sets the value of a local or global string variable.

Syntax

variable <*setx*> *expr*

where:

<*setx*> is one of SETA, SETL, or SETS.

variable is the name of a variable declared by a GBLA, GBLL, GBLS, LCLA, LCLL, or LCLS directive.

expr is an expression, which is:

- numeric, for SETA (see *Numeric expressions* on page 3-26)
- logical, for SETL (see *Logical expressions* on page 3-29)
- string, for SETS (see *String expressions* on page 3-25).

Usage

You must declare *variable* using a global or local declaration directive before using one of these directives. See *GBLA, GBLL, and GBLS* on page 6-5 and *LCLA, LCLL, and LCLS* on page 6-7 for more information.

You can also predefine variable names on the command line. See *Command syntax* on page 3-2 for more information.

Examples

```
                GBLA    VersionNumber
VersionNumber   SETA    21

                GBLL    Debug
Debug           SETL    {TRUE}

                GBLS    VersionString
VersionString   SETS    "Version 1.0"
```

 Copyright © 2005 ARM Limited. All rights reserved.

6.2.4 RLIST

The RLIST (register list) directive gives a name to a set of general-purpose registers.

Syntax

name RLIST {*list-of-registers*}

where:

name is the name to be given to the set of registers. *name* cannot be the same as any of the predefined names listed in *Predefined register and coprocessor names* on page 3-15.

list-of-registers
is a comma-delimited list of register names and/or register ranges. The register list must be enclosed in braces.

Usage

Use RLIST to give a name to a set of registers to be transferred by the LDM or STM instructions.

LDM and STM always put the lowest physical register numbers at the lowest address in memory, regardless of the order they are supplied to the LDM or STM instruction. If you have defined your own symbolic register names it can be less apparent that a register list is not in increasing register order.

Use the -checkreglist assembler option to ensure that the registers in a register list are supplied in increasing register order. If registers are not supplied in increasing register order, a warning is issued.

Example

```
Context RLIST   {r0-r6,r8,r10-r12,r15}
```


Copyright © 2005 ARM Limited. All rights reserved.

6.2.5 CN

The CN directive defines a name for a coprocessor register.

Syntax

name CN *expr*

where:

name is the name to be defined for the coprocessor register. *name* cannot be the same as any of the predefined names listed in *Predefined register and coprocessor names* on page 3-15.

expr evaluates to a coprocessor register number from 0 to 15.

Usage

Use CN to allocate convenient names to registers, to help you remember what you use each register for.

Note

Avoid conflicting uses of the same register under different names.

The names c0 to c15 are predefined.

Example

```
power    CN  6        ; defines power as a symbol for
                      ; coprocessor register 6
```

 Copyright © 2005 ARM Limited. All rights reserved.

6.2.6 CP

The CP directive defines a name for a specified coprocessor. The coprocessor number must be within the range 0 to 15.

Syntax

name CP *expr*

where:

name is the name to be assigned to the coprocessor. *name* cannot be the same as any of the predefined names listed in *Predefined register and coprocessor names* on page 3-15.

expr evaluates to a coprocessor number from 0 to 15.

Usage

Use CP to allocate convenient names to coprocessors, to help you to remember what you use each one for.

Note

Avoid conflicting uses of the same coprocessor under different names.

The names p0 to p15 are predefined for coprocessors 0 to 15.

Example

```
dmu    CP  6       ; defines dmu as a symbol for
                   ; coprocessor 6
```


Copyright © 2005 ARM Limited. All rights reserved.

6.2.7 DN and SN

The DN directive defines a name for a specified double-precision VFP register. The names d0-d15 and D0-D15 are predefined.

The SN directive defines a name for a specified single-precision VFP register. The names s0-s31 and S0-S31 are predefined.

Syntax

name DN *expr*

name SN *expr*

where:

name is the name to be assigned to the VFP register. *name* cannot be the same as any of the predefined names listed in *Predefined register and coprocessor names* on page 3-15.

expr evaluates to a double-precision VFP register number from 0 to 15, or a single-precision VFP register number from 0 to 31 as appropriate.

Usage

Use DN or SN to allocate convenient names to VFP registers, to help you to remember what you use each one for.

Note

Avoid conflicting uses of the same register under different names.

You cannot specify a vector length in a DN or SN directive (see *VFP directives and vector notation* on page 5-36).

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
```

 Copyright © 2005 ARM Limited. All rights reserved.

6.2.8 FN

The FN directive defines a name for a specified FPA floating-point register. The names f0-f7 and F0-F7 are predefined.

Note

The use of the FPA is no longer supported in RVDK.

Syntax

name FN *expr*

where:

name is the name to be assigned to the floating-point register. *name* cannot be the same as any of the predefined names listed in *Predefined register and coprocessor names* on page 3-15.

expr evaluates to a floating-point register number from 0 to 7.

Usage

Use FN to allocate convenient names to FPA floating-point registers, to help you to remember what you use each one for.

Note

Avoid conflicting uses of the same register under different names.

Example

```
energy  FN  6   ; defines energy as a symbol for
                ; floating-point register 6
```


Copyright © 2005 ARM Limited. All rights reserved.

6.3 Data definition directives

This section describes the following directives to allocate memory, define data structures, set initial contents of memory:

- *LTORG* on page 6-16

  Set an origin for a literal pool.

- *MAP* on page 6-17

  Set the origin of a storage map.

- *FIELD* on page 6-18

  Define a field within a storage map.

- *SPACE* on page 6-19

  Allocate a zeroed block of memory.

- *DCB* on page 6-20

  Allocate bytes of memory, and specify the initial contents.

- *DCD and DCDU* on page 6-21

  Allocate words of memory, and specify the initial contents.

- *DCDO* on page 6-22

  Allocate words of memory, and specify the initial contents as offsets from the static base register.

- *DCFD and DCFDU* on page 6-23

  Allocate doublewords of memory, and specify the initial contents as double-precision floating-point numbers.

- *DCFS and DCFSU* on page 6-24

  Allocate words of memory, and specify the initial contents as single-precision floating-point numbers.

- *DCI* on page 6-25

  Allocate words of memory, and specify the initial contents. Mark the location as code not data.

- *DCQ and DCQU* on page 6-26

  Allocate doublewords of memory, and specify the initial contents as 64-bit integers.

- *DCW and DCWU* on page 6-27

 Copyright © 2005 ARM Limited. All rights reserved.

Allocate halfwords of memory, and specify the initial contents.

- *COMMON* on page 6-28
  Allocate a block of memory at a symbol, and specify the alignment.
- *DATA* on page 6-28
  Mark data within a code section. Obsolete, for backwards compatibility only.

Copyright © 2005 ARM Limited. All rights reserved.

6.3.1 LTORG

The LTORG directive instructs the assembler to assemble the current literal pool immediately.

Syntax

LTORG

Usage

The assembler assembles the current literal pool at the end of every code section. The end of a code section is determined by the AREA directive at the beginning of the following section, or the end of the assembly.

These default literal pools can sometimes be out of range of some LDR, LDFD, and LDFS pseudo-instructions. See *LDR pseudo-instruction* on page 4-105. 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 © 2005 ARM Limited. All rights reserved.

6.3.2 MAP

The MAP directive sets the origin of a storage map to a specified address. The storage-map location counter, {VAR}, is set to the same address. ^ is a synonym for MAP.

Syntax

MAP *expr*{,*base-register*}

where:

expr is a numeric or program-relative expression:

- If *base-register* is not specified, *expr* evaluates to the address where the storage map starts. The storage map location counter is set to this address.
- If *expr* is program-relative, you must have defined the label before you use it in the map. The map requires the definition of the label during the first pass of the assembler.

base-register

specifies a register. If *base-register* is specified, the address where the storage map starts is the sum of *expr*, and the value in *base-register* at runtime.

Usage

Use the MAP directive in combination with the FIELD directive to describe a storage map.

Specify *base-register* to define register-relative labels. The base register becomes implicit in all labels defined by following FIELD directives, until the next MAP directive. The register-relative labels can be used in load and store instructions. See *FIELD* on page 6-18 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 © 2005 ARM Limited. All rights reserved.

6.3.3 FIELD

The FIELD directive describes space within a storage map that has been defined using the MAP directive. # is a synonym for FIELD.

Syntax

{*label*} FIELD *expr*

where:

label is an optional label. If specified, *label* is assigned the value of the storage location counter, {VAR}. The storage location counter is then incremented by the value of *expr*.

expr is an expression that evaluates to the number of bytes to increment the storage counter.

Usage

If a storage map is set by a MAP directive that specifies a *base-register*, the base register is implicit in all labels defined by following FIELD directives, until the next MAP directive. These register-relative labels can be quoted in load and store instructions (see *MAP* on page 6-17).

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 © 2005 ARM Limited. All rights reserved.

6.3.4 SPACE

The SPACE directive reserves a zeroed block of memory. % is a synonym for SPACE.

Syntax

{*label*} SPACE *expr*

where:

expr evaluates to the number of zeroed bytes to reserve (see *Numeric expressions* on page 3-26).

Usage

Use the ALIGN directive to align any code following a SPACE directive. See *ALIGN* on page 6-64 for more information.

See also:

- *DCB* on page 6-20
- *DCD and DCDU* on page 6-21
- *DCDO* on page 6-22
- *DCW and DCWU* on page 6-27.

Example

```
        AREA    MyData, DATA, READWRITE
data1   SPACE   255     ; defines 255 bytes of zeroed store
```


Copyright © 2005 ARM Limited. All rights reserved.

6.3.5 DCB

The DCB directive allocates one or more bytes of memory, and defines the initial runtime contents of the memory. = is a synonym for DCB.

Syntax

{*label*} DCB *expr*{,*expr*}...

where:

expr is either:

- A numeric expression that evaluates to an integer in the range –128 to 255 (see *Numeric expressions* on page 3-26).
- A quoted string. The characters of the string are loaded into consecutive bytes of store.

Usage

If DCB is followed by an instruction, use an ALIGN directive to ensure that the instruction is aligned. See *ALIGN* on page 6-64 for more information.

See also:

- *DCD and DCDU* on page 6-21
- *DCQ and DCQU* on page 6-26
- *DCW and DCWU* on page 6-27
- *SPACE* on page 6-19.

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 © 2005 ARM Limited. All rights reserved.

6.3.6 DCD and DCDU

The DCD directive allocates one or more words of memory, aligned on four-byte boundaries, and defines the initial runtime contents of the memory.

& is a synonym for DCD.

DCDU is the same, except that the memory alignment is arbitrary.

Syntax

{*label*} DCD{U} *expr*{,*expr*}

where:

expr is either:
- a numeric expression (see *Numeric expressions* on page 3-26).
- a program-relative expression.

Usage

DCD inserts up to three bytes of padding before the first defined word, if necessary, to achieve four-byte alignment.

Use DCDU if you do not require alignment.

See also:
- *DCB* on page 6-20
- *DCW and DCWU* on page 6-27
- *DCQ and DCQU* on page 6-26
- *SPACE* on page 6-19.

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 © 2005 ARM Limited. All rights reserved.

6.3.7 DCDO

The DCDO directive allocates one or more words of memory, aligned on four-byte boundaries, and defines the initial runtime contents of the memory as an offset from the *static base register*, sb (r9).

Syntax

{*label*} DCDO *expr*{,*expr*}...

where:

expr is a register-relative expression or label. The base register must be sb.

Usage

Use DCDO to allocate space in memory for static base register relative relocatable addresses.

Example

```
IMPORT  externsym
DCDO    externsym   ; 32-bit word relocated by offset of
                    ; externsym from base of SB section.
```

 Copyright © 2005 ARM Limited. All rights reserved.

6.3.8 DCFD and DCFDU

The DCFD directive allocates memory for word-aligned double-precision floating-point numbers, and defines the initial runtime contents of the memory. Double-precision numbers occupy two words and must be word aligned to be used in arithmetic operations.

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 3-28).

Usage

The assembler inserts up to three bytes of padding before the first defined number, if necessary, to achieve four-byte alignment.

Use DCFDU if you do not require alignment.

The word order used when converting *fpliteral* to internal form is controlled by the floating-point architecture selected. You cannot use DCFD or DCFDU if you select the -fpu none option.

The range for double-precision numbers is:

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

See also *DCFS and DCFSU* on page 6-24.

Examples

```
DCFD    1E308,-4E-100
DCFDU   10000,-.1,3.1E26
```


Copyright © 2005 ARM Limited. All rights reserved.

6.3.9 DCFS and DCFSU

The DCFS directive allocates memory for word-aligned single-precision floating-point numbers, and defines the initial runtime contents of the memory. Single-precision numbers occupy one word and must be word aligned to be used in arithmetic operations.

DCDSU is the same, except that the memory alignment is arbitrary.

Syntax

{*label*} DCFS{U} *fpliteral*{,*fpliteral*}...

where:

fpliteral is a single-precision floating-point literal (see *Floating-point literals* on page 3-28).

Usage

DCFS inserts up to three bytes of padding before the first defined number, if necessary to achieve four-byte alignment.

Use DCFSU if you do not require alignment.

The range for single-precision values is:

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

See also *DCFD and DCFDU* on page 6-23.

Example

```
DCFS    1E3,-4E-9
DCFSU   1.0,-.1,3.1E6
```

 Copyright © 2005 ARM Limited. All rights reserved.

6.3.10 DCI

In ARM code, the DCI directive allocates one or more words of memory, aligned on four-byte boundaries, and defines the initial runtime contents of the memory.

In Thumb code, the DCI directive allocates one or more halfwords of memory, aligned on two-byte boundaries, and defines the initial runtime contents of the memory.

Syntax

{*label*} DCI *expr*{,*expr*}

where:

expr is a numeric expression (see *Numeric expressions* on page 3-26).

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 instuction stream, for example, use:

```
DCI 0x46c0
```

to insert the Thumb operation MOV r8,r8.

See also *DCD and DCDU* on page 6-21 and *DCW and DCWU* on page 6-27.

Example

```
MACRO           ; this macro translates newinstr Rd,Rm
                ; to the appropriate machine code
newinst     $Rd,$Rm
DCI         0xe16f0f10 :OR: ($Rd:SHL:12) :OR: $Rm
MEND
```


Copyright © 2005 ARM Limited. All rights reserved.

6.3.11 DCQ and DCQU

The DCQ directive allocates one or more eight-byte blocks of memory, aligned on four-byte boundaries, and defines the initial runtime contents of the memory.

DCQU is the same, except that the memory alignment is arbitrary.

Syntax

{*label*} DCQ{U} {-}*literal*{,{-}*literal*}...

where:

literal is a 64-bit numeric literal (see *Numeric literals* on page 3-27).

The range of numbers permitted is 0 to $2^{64} – 1$.

In addition to the characters normally permitted in a numeric literal, you can prefix *literal* with a minus sign. In this case, the range of numbers permitted is $–2^{63}$ to –1.

The result of specifying -*n* is the same as the result of specifying $2^{64} – n$.

Usage

DCQ inserts up to three bytes of padding before the first defined eight-byte block, if necessary, to achieve four-byte alignment.

Use DCQU if you do not require alignment.

See also:

- *DCB* on page 6-20
- *DCD and DCDU* on page 6-21
- *DCW and DCWU* on page 6-27
- *SPACE* on page 6-19.

Example

```
        AREA    MiscData, DATA, READWRITE
data    DCQ     -225,2_101      ; 2_101 means binary 101.
        DCQU    number+4        ; number must already be defined.
```

 Copyright © 2005 ARM Limited. All rights reserved.

6.3.12 DCW and DCWU

The DCW directive allocates one or more halfwords of memory, aligned on two-byte boundaries, and defines the initial runtime contents of the memory.

DCWU is the same, except that the memory alignment is arbitrary.

Syntax

{*label*} DCW *expr*{,*expr*}...

where:

expr is a numeric expression that evaluates to an integer in the range –32768 to 65535 (see *Numeric expressions* on page 3-26).

Usage

DCW inserts a byte of padding before the first defined halfword if necessary to achieve two-byte alignment.

Use DCWU if you do not require alignment.

See also:

- *DCB* on page 6-20
- *DCD and DCDU* on page 6-21
- *DCQ and DCQU* on page 6-26
- *SPACE* on page 6-19.

Example

```
data    DCW     -225,2*number   ; number must already be defined

        DCWU    number+4
```


Copyright © 2005 ARM Limited. All rights reserved.

6.3.13 COMMON

The COMMON directive allocates a block of memory, of the defined size, at the specified symbol. You specify how the memory is aligned. If alignment is omitted, the default alignment is 4. If size is omitted, the default size is 0.

You can access this memory as you would any other memory, but no space is allocated in object files.

Syntax

COMMON *symbol*{,*size*{,*alignment*}}

where:

symbol is the symbol name. The symbol name is case-sensitive.

size is the number of bytes to reserve.

alignment is the alignment.

Usage

The linker allocates the required space as zero-initialized memory during the link stage.

Example

```
COMMON    xyz,255,4   ; defines 255 bytes of ZI store, word-aligned
```

6.3.14 DATA

The DATA directive is no longer needed. It is ignored by the assembler.

 Copyright © 2005 ARM Limited. All rights reserved.

6.4 Assembly control directives

This section describes the following directives to control conditional assembly, looping, inclusions, and macros:

- *MACRO and MEND* on page 6-30
- *MEXIT* on page 6-33
- *IF, ELSE, ENDIF, and ELIF* on page 6-34
- *WHILE and WEND* on page 6-37.

6.4.1 Nesting directives

The following structures can be nested to a total depth of 256:

- `MACRO` definitions
- `WHILE...WEND` loops
- `IF...ELSE...ENDIF` conditional structures
- `INCLUDE` file inclusions.

The limit applies to all structures taken together, however they are nested. The limit is not 256 of each type of structure.

Copyright © 2005 ARM Limited. All rights reserved.

6.4.2 MACRO and MEND

The MACRO directive marks the start of the definition of a macro. Macro expansion terminates at the MEND directive. See *Using macros* on page 2-42 for more information.

Syntax

Two directives are used to define a macro. The syntax is:

```
          MACRO
{$label}  macroname {$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.

$*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 6-33 if you need to enable an early exit from a macro, for example from within a loop.

Within the macro body, parameters such as $*label*, $*parameter* can be used in the same way as other variables (see *Assembly time substitution of variables* on page 3-20). They are given new values each time the macro is invoked. Parameters must begin with $ to distinguish them from ordinary symbols. Any number of parameters can be used.

$label is optional. It is useful if the macro defines internal labels. It is treated as a parameter to the macro. It does not necessarily represent the first instruction in the macro expansion. The macro defines the locations of any labels.

Use | as the argument to use the default value of a parameter. An empty string is used if the argument is omitted.

 Copyright © 2005 ARM Limited. All rights reserved.

In a macro that uses several internal labels, it is useful to define each internal label as the base label with a different suffix.

Use a dot between a parameter and following text, or a following parameter, if a space is not required in the expansion. Do not use a dot between preceding text and a parameter.

Macros define the scope of local variables (see *LCLA, LCLL, and LCLS* on page 6-7).

Macros can be nested (see *Nesting directives* on page 6-29).

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 © 2005 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
```

 Copyright © 2005 ARM Limited. All rights reserved.

6.4.3 MEXIT

The MEXIT directive is used to exit a macro definition before the end.

Usage

Use MEXIT when you need an exit from within the body of a macro. Any unclosed WHILE...WEND loops or IF...ENDIF conditions within the body of the macro are closed by the assembler before the macro is exited.

See also *MACRO and MEND* on page 6-30.

Example

```
        MACRO
$abc    macro   abc     $param1,$param2
        ; code
        WHILE condition1
            ; code
            IF condition2
                ; code
                MEXIT
            ELSE
                ; code
            ENDIF
        WEND
        ; code
        MEND
```


Copyright © 2005 ARM Limited. All rights reserved.

6.4.4 IF, ELSE, ENDIF, and ELIF

The IF directive introduces a condition that is used to decide whether to assemble a sequence of instructions and/or directives. [is a synonym for IF.

The ELSE directive marks the beginning of a sequence of instructions and/or directives that you want to be assembled if the preceding condition fails. | is a synonym for ELSE.

The ENDIF directive marks the end of a sequence of instructions and/or directives that you want to be conditionally assembled.] is a synonym for ENDIF.

The ELIF directive creates a structure equivalent to ELSE IF, without the need for nesting or repeating the condition. See *Using ELIF* on page 6-35 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 3-36.

Usage

Use IF with ENDIF, and optionally with ELSE, for sequences of instructions and/or directives that are only to be assembled or acted on under a specified condition.

IF...ENDIF conditions can be nested (see *Nesting directives* on page 6-29).

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

Examples

Example 6-3 assembles the first set of instructions if NEWVERSION is defined, or the alternative set otherwise.

Example 6-3 Assembly conditional on a variable being defined

```
        IF :DEF:NEWVERSION
            ; first set of instructions/directives
        ELSE
            ; alternative set of instructions/directives
        ENDIF
```

Invoking armasm as follows defines NEWVERSION, so the first set of instructions and directives are assembled:

```
armasm --PD "NEWVERSION SETL {TRUE}" test.s
```

Invoking armasm as follows leaves NEWVERSION undefined, so the second set of instructions and directives are assembled:

```
armasm test.s
```

Example 6-4 assembles the first set of instructions if NEWVERSION has the value {TRUE}, or the alternative set otherwise.

Example 6-4 Assembly conditional on a variable being defined

```
        IF NEWVERSION = {TRUE}
            ; first set of instructions/directives
        ELSE
            ; alternative set of instructions/directives
        ENDIF
```

Invoking armasm as follows causes the first set of instructions and directives to be assembled:

```
armasm --PD "NEWVERSION SETL {TRUE}" test.s
```

Invoking armasm as follows causes the second set of instructions and directives to be assembled:

```
armasm --PD "NEWVERSION SETL {FALSE}" test.s
```

 Copyright © 2005 ARM Limited. All rights reserved.

6.4.5 WHILE and WEND

The WHILE directive starts a sequence of instructions or directives that are to be assembled repeatedly. The sequence is terminated with a WEND directive.

Syntax

```
WHILE logical-expression

code

WEND
```

where:

logical-expression

is an expression that can evaluate to either {TRUE} or {FALSE} (see *Logical expressions* on page 3-29).

Usage

Use the WHILE directive, together with the WEND directive, to assemble a sequence of instructions a number of times. The number of repetitions can be zero.

You can use IF...ENDIF conditions within WHILE...WEND loops.

WHILE...WEND loops can be nested (see *Nesting directives* on page 6-29).

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 © 2005 ARM Limited. All rights reserved.

6.5 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.

If you require profiling of assembler functions, but do not need frame description directives for other purposes:

- you must use the `FUNCTION` and `ENDFUNC`, or `PROC` and `ENDP`, directives

 Copyright © 2005 ARM Limited. All rights reserved.

- you can omit the other FRAME directives
- you only need 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 © 2005 ARM Limited. All rights reserved.

6.5.1 FRAME ADDRESS

The FRAME ADDRESS directive describes how to calculate the canonical frame address for following instructions. You can only use it in functions with FUNCTION and ENDFUNC or PROC and ENDP directives.

Syntax

FRAME ADDRESS *reg*[,*offset*]

where:

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 alters the offset of the canonical frame address from the register. You must use FRAME ADDRESS immediately after the instruction which changes the calculation of the canonical frame address.

Note

If your code uses a single instruction to save registers and alter the stack pointer, you can use FRAME PUSH instead of using both FRAME ADDRESS and FRAME SAVE (see *FRAME PUSH* on page 6-42).

If your code uses a single instruction to load registers and alter the stack pointer, you can use FRAME POP instead of using both FRAME ADDRESS and FRAME RESTORE (see *FRAME POP* on page 6-41).

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 © 2005 ARM Limited. All rights reserved.

6.5.2 FRAME POP

Use the FRAME POP directive to inform the assembler when the callee reloads registers. You can only use it within functions with FUNCTION and ENDFUNC or PROC and ENDP directives.

You need not do this after the last instruction in a function.

Syntax

There are two alternative syntaxes for FRAME POP:

```
FRAME POP {reglist}
```

```
FRAME POP n
```

where:

reglist is a list of registers restored to the values they had on entry to the function. There must be at least one register in the list.

n is the number of bytes that the stack pointer moves.

Usage

FRAME POP is equivalent to a FRAME ADDRESS and a FRAME RESTORE directive. You can use it when a single instruction loads registers and alters the stack pointer.

You must use FRAME POP immediately after the instruction it refers to.

The assembler calculates the new offset for the canonical frame address. It assumes that:

- each ARM register popped occupied four bytes on the stack
- each FPA floating-point register popped occupied 12 bytes on the stack
- each VFP single-precision register popped occupied four bytes on the stack, plus an extra four-byte word for each list.

See *FRAME ADDRESS* on page 6-40 and *FRAME RESTORE* on page 6-45.

Copyright © 2005 ARM Limited. All rights reserved.

6.5.3 FRAME PUSH

Use the FRAME PUSH directive to inform the assembler when the callee saves registers, normally at function entry. You can only use it within functions with FUNCTION and ENDFUNC or PROC and ENDP directives.

Syntax

There are two alternative syntaxes for FRAME PUSH:

```
FRAME PUSH {reglist}
```

```
FRAME PUSH n
```

where:

reglist is a list of registers stored consecutively below the canonical frame address. There must be at least one register in the list.

n is the number of bytes that the stack pointer moves.

Usage

FRAME PUSH is equivalent to a FRAME ADDRESS and a FRAME SAVE directive. You can use it when a single instruction saves registers and alters the stack pointer.

You must use FRAME PUSH immediately after the instruction it refers to.

The assembler calculates the new offset for the canonical frame address. It assumes that:

- each ARM register pushed occupies four bytes on the stack
- each FPA floating-point register pushed occupied 12 bytes on the stack
- each VFP single-precision register pushed occupies four bytes on the stack, plus an extra four-byte word for each list.

See *FRAME ADDRESS* on page 6-40 and *FRAME SAVE* on page 6-47.

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

6.5.4 FRAME REGISTER

Use the FRAME REGISTER directive to maintain a record of the locations of function arguments held in registers. You can only use it within functions with FUNCTION and ENDFUNC or PROC and ENDP directives.

Syntax

FRAME REGISTER *reg1*,*reg2*

where:

reg1 is the register that held the argument on entry to the function.

reg2 is the register in which the value is preserved.

Usage

Use the FRAME REGISTER directive when you use a register to preserve an argument that was held in a different register on entry to a function.

Copyright © 2005 ARM Limited. All rights reserved.

6.5.5 FRAME RESTORE

Use the FRAME RESTORE directive to inform the assembler that the contents of specified registers have been restored to the values they had on entry to the function. You can only use it within functions with FUNCTION and ENDFUNC or PROC and ENDP directives.

Syntax

FRAME RESTORE {*reglist*}

where:

reglist is a list of registers whose contents have been restored. There must be at least one register in the list.

Usage

Use FRAME RESTORE immediately after the callee reloads registers from the stack. You need not do this after the last instruction in a function.

reglist can contain integer registers or floating-point registers, but not both.

Note

If your code uses a single instruction to load registers and alter the stack pointer, you can use FRAME POP instead of using both FRAME RESTORE and FRAME ADDRESS (see *FRAME POP* on page 6-41).

Copyright © 2005 ARM Limited. All rights reserved.

6.5.6 FRAME RETURN ADDRESS

The FRAME RETURN ADDRESS directive provides for functions that use a register other than r14 for their return address. You can only use it within functions with FUNCTION and ENDFUNC or PROC and ENDP directives.

Note

Any function that uses a register other than r14 for its return address is not AAPCS compliant. Such a function must not be exported.

Syntax

FRAME RETURN ADDRESS *reg*

where:

reg is the register used for the return address.

Usage

Use the FRAME RETURN ADDRESS directive in any function that does not use r14 for its return address. Otherwise, a debugger cannot backtrace through the function.

Use FRAME RETURN ADDRESS immediately after the FUNCTION or PROC directive that introduces the function.

Copyright © 2005 ARM Limited. All rights reserved.

6.5.7 FRAME SAVE

The FRAME SAVE directive describes the location of saved register contents relative to the canonical frame address. You can only use it within functions with FUNCTION and ENDFUNC or PROC and ENDP directives.

Syntax

FRAME SAVE {*reglist*}, *offset*

where:

reglist is a list of registers stored consecutively starting at *offset* from the canonical frame address. There must be at least one register in the list.

Usage

Use FRAME SAVE immediately after the callee stores registers onto the stack.

reglist can include registers which are not required for backtracing. The assembler determines which registers it needs to record in the DWARF call frame information.

Note

If your code uses a single instruction to save registers and alter the stack pointer, you can use FRAME PUSH instead of using both FRAME SAVE and FRAME ADDRESS (see *FRAME PUSH* on page 6-42).

Copyright © 2005 ARM Limited. All rights reserved.

6.5.8 FRAME STATE REMEMBER

The FRAME STATE REMEMBER directive saves the current information on how to calculate the canonical frame address and locations of saved register values. You can only use it within functions with FUNCTION and ENDFUNC or PROC and ENDP directives.

Syntax

```
FRAME STATE REMEMBER
```

Usage

During an inline exit sequence the information about calculation of canonical frame address and locations of saved register values can change. After the exit sequence another branch can continue using the same information as before. Use FRAME STATE REMEMBER to preserve this information, and FRAME STATE RESTORE to restore it.

These directives can be nested. Each FRAME STATE RESTORE directive must have a corresponding FRAME STATE REMEMBER directive. See:

- *FRAME STATE RESTORE* on page 6-49
- *FUNCTION or PROC* on page 6-51.

Example

```
        ; function code
        FRAME STATE REMEMBER
            ; save frame state before in-line exit sequence
        POP     {r4-r6,pc}
            ; no need 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 © 2005 ARM Limited. All rights reserved.

6.5.9 FRAME STATE RESTORE

The FRAME STATE RESTORE directive restores information about how to calculate the canonical frame address and locations of saved register values. You can only use it within functions with FUNCTION and ENDFUNC or PROC and ENDP directives.

Syntax

```
FRAME STATE RESTORE
```

Usage

See:

- *FRAME STATE REMEMBER* on page 6-48
- *FUNCTION or PROC* on page 6-51.

Copyright © 2005 ARM Limited. All rights reserved.

6.5.10 FRAME UNWIND ON

The FRAME UNWIND ON directive instructs the assembler to produce *unwind* tables for this and subsequent functions.

Syntax

```
FRAME UNWIND ON
```

Usage

You can use this directive outside functions. In this case, the assembler produces *unwind* tables for all following functions until it reaches a FRAME UNWIND OFF directive.

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

6.5.11 FRAME UNWIND OFF

The FRAME UNWIND OFF directive instructs the assembler to produce *nounwind* tables for this and subsequent functions.

Syntax

```
FRAME UNWIND OFF
```

Usage

You can use this directive outside functions. In this case, the assembler produces *nounwind* tables for all following functions until it reaches a FRAME UNWIND ON directive.

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

 Copyright © 2005 ARM Limited. All rights reserved.

6.5.12 FUNCTION or PROC

The FUNCTION directive marks the start of an AAPCS-conforming function. PROC is a synonym for FUNCTION.

Syntax

label FUNCTION [{*Reglist1*} [, {*Reglist2*}]]

where:

reglist1 is an optional list of callee saved ARM registers. If *reglist1* is not present, and your debugger checks register usage, it will assume that the AAPCS is in use.

reglist2 is an optional list of callee saved VFP registers.

Usage

Use FUNCTION to mark the start of functions. The assembler uses FUNCTION to identify the start of a function when producing DWARF call frame information for ELF.

RVDK uses an ELF proprietary file format called *ARM Toolkit Proprietary ELF* (ATPE). The file format for each version of RVDK is restricted to the proprietary ATPE format for the permitted device. This is referred to as *ATPE_Custom*.

FUNCTION sets the canonical frame address to be sp, and the frame state stack to be empty.

Each FUNCTION directive must have a matching ENDFUNC directive. You must not nest FUNCTION/ENDFUNC pairs, and they must not contain PROC or ENDP directives.

You can use the optional *reglist* parameters to inform the debugger about an alternative procedure call standard, if you are using your own. Not all debuggers support this feature. See your debugger documentation for details.

See also *FRAME ADDRESS* on page 6-40 to *FRAME STATE RESTORE* on page 6-49.

Note

FUNCTION does not automatically cause alignment to a word boundary (or halfword boundary for Thumb). Use ALIGN if necessary to ensure alignment, otherwise the call frame might not point to the start of the function. See *ALIGN* on page 6-64 for further information.

Copyright © 2005 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
```

6.5.13 ENDFUNC or ENDP

The ENDFUNC directive marks the end of an AAPCS-conforming function (see *FUNCTION or PROC* on page 6-51). ENDP is a synonym for ENDFUNC.

Copyright © 2005 ARM Limited. All rights reserved.

6.6 Reporting directives

This section describes the following directives:

- *ASSERT*
  generates an error message if an assertion is false during assembly.
- *INFO* on page 6-54
  generates diagnostic information during assembly.
- *OPT* on page 6-55
  sets listing options.
- *TTL and SUBT* on page 6-57
  insert titles and subtitles in listings.

6.6.1 ASSERT

The ASSERT directive generates an error message during the second pass of the assembly if a given assertion is false.

Syntax

`ASSERT` *`logical-expression`*

where:

`logical-expression`
is an assertion that can evaluate to either `{TRUE}` or `{FALSE}`.

Usage

Use `ASSERT` to ensure that any necessary condition is met during assembly.

If the assertion is false an error message is generated and assembly fails.

See also *INFO* on page 6-54.

Example

```
ASSERT  label1 <= label2    ; Tests if the address
                            ; represented by label1
                            ; is <= the address
                            ; represented by label2.
```


Copyright © 2005 ARM Limited. All rights reserved.

6.6.2 INFO

The INFO directive supports diagnostic generation on either pass of the assembly.

! is very similar to INFO, but has less detailed reporting.

Syntax

INFO *numeric-expression, string-expression*

where:

numeric-expression

is a numeric expression that is evaluated during assembly. If the expression evaluates to zero:

- no action is taken during pass one
- *string-expression* is printed during pass two.

If the expression does not evaluate to zero, *string-expression* is printed as an error message and the assembly fails.

string-expression

is an expression that evaluates to a string.

Usage

INFO provides a flexible means for creating custom error messages. See *Numeric expressions* on page 3-26 and *String expressions* on page 3-25 for additional information on numeric and string expressions.

See also *ASSERT* on page 6-53.

Examples

```
INFO    0, "Version 1.0"

IF endofdata <= label1
    INFO    4, "Data overrun at label1"
ENDIF
```


Copyright © 2005 ARM Limited. All rights reserved.

6.6.3 OPT

The OPT directive sets listing options from within the source code.

Syntax

OPT *n*

where:

n is the OPT directive setting. Table 6-2 lists valid settings.

Table 6-2 OPT directive settings

OPT n	Effect
1	Turns on normal listing.
2	Turns off normal listing.
4	Page throw. Issues an immediate form feed and starts a new page.
8	Resets the line number counter to zero.
16	Turns on listing for SET, GBL and LCL directives.
32	Turns off listing for SET, GBL and LCL directives.
64	Turns on listing of macro expansions.
128	Turns off listing of macro expansions.
256	Turns on listing of macro invocations.
512	Turns off listing of macro invocations.
1024	Turns on the first pass listing.
2048	Turns off the first pass listing.
4096	Turns on listing of conditional directives.
8192	Turns off listing of conditional directives.
16384	Turns on listing of MEND directives.
32768	Turns off listing of MEND directives.

Usage

Specify the -list assembler option to turn on listing.

Copyright © 2005 ARM Limited. All rights reserved.

By default the -list option produces a normal listing that includes variable declarations, macro expansions, call-conditioned directives, and MEND directives. The listing is produced on the second pass only. Use the OPT directive to modify the default listing options from within your code. See *Command syntax* on page 3-2 for information on the -list option.

You can use OPT to format code listings. For example, you can specify a new page before functions and sections.

Example

```
        AREA    Example, CODE, READONLY
start   ; code
        ; code
        BL      func1
        ; code
        OPT 4               ; places a page break before func1
func1   ; code
```


Copyright © 2005 ARM Limited. All rights reserved.

6.6.4 TTL and SUBT

The TTL directive inserts a title at the start of each page of a listing file. The title is printed on each page until a new TTL directive is issued.

The SUBT directive places a subtitle on the pages of a listing file. The subtitle is printed on each page until a new SUBT directive is issued.

Syntax

TTL *title*

SUBT *subtitle*

where:

title is the title

subtitle is the subtitle.

Usage

Use the TTL directive to place a title at the top of the pages of a listing file. If you want the title to appear on the first page, the TTL directive must be on the first line of the source file.

Use additional TTL directives to change the title. Each new TTL directive takes effect from the top of the next page.

Use SUBT to place a subtitle at the top of the pages of a listing file. Subtitles appear in the line below the titles. If you want the subtitle to appear on the first page, the SUBT directive must be on the first line of the source file.

Use additional SUBT directives to change subtitles. Each new SUBT directive takes effect from the top of the next page.

Example

```
TTL     First Title     ; places a title on the first
                        ; and subsequent pages of a
                        ; listing file.
SUBT    First Subtitle  ; places a subtitle on the
                        ; second and subsequent pages
                        ; of a listing file.
```


Copyright © 2005 ARM Limited. All rights reserved.

6.7 Instruction set and syntax selection directives

This section describes the following directives:

 Copyright © 2005 ARM Limited. All rights reserved.

6.7.1 ARM and CODE32

The ARM directive and the CODE32 directive are synonyms. They instruct the assembler to interpret subsequent instructions as ARM instructions. If necessary, it also inserts up to three bytes of padding to align to the next word boundary.

In this mode, the assembler accepts both the latest assembly language and the earlier version.

Syntax

```
ARM

CODE32
```

Usage

In files that contain a mixture of ARM and Thumb code:

- Use ARM when changing from Thumb state to ARM state. ARM (or CODE32) must precede any ARM code.
- Use THUMB when changing from ARM state to Thumb state, if the following code uses ARM syntax (see *THUMB* on page 6-60).
- Use CODE16 when changing from ARM state to Thumb state, if the following code uses old Thumb syntax (see *CODE16* on page 6-62).

ARM and CODE32 do not assemble to instructions that change the state. They only instruct the assembler to assemble ARM instructions as appropriate, and insert padding if necessary.

Example

This example shows how ARM and CODE16 can be used to branch from ARM to 16-bit Thumb instructions.

```
        AREA    ChangeState, CODE, READONLY
        ARM
                            ; This section starts in ARM state
        LDR     r0,=start+1 ; Load the address and set the
                            ; least significant bit
        BX      r0          ; Branch and exchange instruction sets

                            ; Not necessarily in same section
        CODE16              ; Following instructions are old Thumb
start   MOV     r1,#10      ; old Thumb instructions
```


Copyright © 2005 ARM Limited. All rights reserved.

6.7.2 THUMB

The THUMB directive instructs the assembler to interpret subsequent instructions as 32-bit Thumb-2 instructions or 16-bit Thumb instructions using new syntax. If necessary, it also inserts padding to align to the next halfword boundary.

Note

None of the processors supported by this toolkit supports Thumb-2.

Syntax

```
THUMB
```

Usage

In files that contain a mixture of ARM and Thumb code:

- Use THUMB when changing from ARM state to Thumb state, if the following code uses ARM syntax.
- Use CODE16 when changing from ARM state to Thumb state, if the following code uses old Thumb syntax (see *CODE16* on page 6-62).
- Use ARM when changing from Thumb state to ARM state. ARM (or CODE32) must precede any ARM code (see *ARM and CODE32* on page 6-59).

THUMB does not assemble to an instruction that changes the state. It only instructs the assembler to assemble Thumb-2 or Thumb instructions as appropriate, and insert padding if necessary.

Example

This example shows how ARM and THUMB can be used to branch from ARM to Thumb instructions.

```
AREA    ChangeState, CODE, READONLY
ARM
                    ; This section starts in ARM state
LDR     r0,=start+1 ; Load the address and set the
                    ; least significant bit
BX      r0          ; Branch and exchange instruction sets

                    ; Not necessarily in same section
THUMB               ; Following instructions are Thumb-2

GSthing        PROC
```

 Copyright © 2005 ARM Limited. All rights reserved.

```
B       {pc}+2 ; #0x8002
B       {pc}+4 ; #0x8004
```


Copyright © 2005 ARM Limited. All rights reserved.

6.7.3 CODE16

The CODE16 directive instructs the assembler to interpret subsequent instructions as 16-bit Thumb instructions using old Thumb syntax. If necessary, it also inserts a byte of padding to align to the next halfword boundary.

Syntax

```
CODE16
```

Usage

In files that contain a mixture of ARM and Thumb code:

- Use CODE16 when changing from ARM state to Thumb state, if the following code uses old Thumb syntax.
- Use THUMB when changing from ARM state to Thumb state, if the following code uses ARM syntax (see *THUMB* on page 6-60).
- Use ARM when changing from Thumb state to ARM state. ARM (or CODE32) must precede any ARM code (see *ARM and CODE32* on page 6-59).

CODE16 does not assemble to instructions that change the state. It only instructs the assembler to assemble Thumb instructions as appropriate, and insert padding if necessary.

Example

This example shows how CODE16 can be used to branch from ARM to Thumb instructions.

```
        AREA    ChangeState, CODE, READONLY
        ARM
                            ; This section starts in ARM state
        LDR     r0,=start+1 ; Load the address and set the
                            ; least significant bit
        BX      r0          ; Branch and exchange instruction sets

                            ; Not necessarily in same section

        CODE16              ; Following instructions are old Thumb
start   MOV     r1,#10      ; old Thumb instructions
```

 Copyright © 2005 ARM Limited. All rights reserved.

6.8 Miscellaneous directives

This section describes the following directives:

Copyright © 2005 ARM Limited. All rights reserved.

6.8.1 ALIGN

The ALIGN directive aligns the current location to a specified boundary by padding with zeros.

Syntax

```
ALIGN {expr{,offset{,pad }}}
```

where:

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.

Operation

The current location is aligned to the next address of the form:

```
offset + n * expr
```

If *expr* is not specified, ALIGN sets the current location to the next word (four byte) boundary.

The unused bytes between the previous and the new current location are filled with copies of the least significant byte of *pad*, or zeros if *pad* is not specified.

Usage

Use ALIGN to ensure that your data and code is aligned to appropriate boundaries. This is typically required in the following circumstances:

- The ADR Thumb pseudo-instruction can only load addresses that are word aligned, but a label within Thumb code might not be word aligned. Use ALIGN 4 to ensure four-byte alignment of an address within Thumb code.
- Use ALIGN to take advantage of caches on some ARM processors. For example, the ARM926EJ-S™ has a cache with 16-byte lines. Use ALIGN 16 to align function entries on 16-byte boundaries and maximize the efficiency of the cache.
- LDRD and STRD doubleword data transfers must be eight-byte aligned. Use ALIGN 8 before memory allocation directives such as DCQ (see *Data definition directives* on page 6-14) if the data is to be accessed using LDRD or STRD.

 Copyright © 2005 ARM Limited. All rights reserved.

- A label on a line by itself can be arbitrarily aligned. Following ARM code is word-aligned (Thumb code is halfword aligned). The label therefore does not address the code correctly. Use `ALIGN 4` (or `ALIGN 2` for Thumb) before the label.

Alignment is relative to the start of the ELF section where the routine is located. The section must be aligned to the same, or coarser, boundaries. The `ALIGN` attribute on the `AREA` directive is specified differently (see *AREA* on page 6-66 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
        ALIGN               ; ensures that subroutine1 addresses
subroutine1                 ; the following instruction.
        MOV r5,#0x5
```


Copyright © 2005 ARM Limited. All rights reserved.

6.8.2 AREA

The AREA directive instructs the assembler to assemble a new code or data section. Sections are independent, named, indivisible chunks of code or data that are manipulated by the linker. See *ELF sections and the AREA directive* on page 2-13 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 6-64.

Note

Do not use ALIGN=0 or ALIGN=1 for 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.

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.

 Copyright © 2005 ARM Limited. All rights reserved.

Identical ELF sections with the same name are overlaid in the same section of memory by the linker. If any are different, the linker generates a warning and does not overlay the sections. See the chapter describing basic linker functionality in *RealView Developer Kit v2.2 Linker and Utilities Guide*.

`COMMON` Is a common data section. You must not define any code or data in it. It is initialized to zeros by the linker. All common sections with the same name are overlaid in the same section of memory by the linker. They do not all need to be the same size. The linker allocates as much space as is required by the largest common section of each name.

`DATA` Contains data, not instructions. `READWRITE` is the default.

`NOALLOC` Indicates that no memory on the target system is allocated to this AREA.

`NOINIT` Indicates that the data section is uninitialized, or initialized to zero. It contains only space reservation directives `SPACE` or `DCB`, `DCD`, `DCDU`, `DCQ`, `DCQU`, `DCW`, or `DCWU` with initialized values of zero. You can decide at link time whether an `AREA` is uninitialized or zero-initialized (see the chapter describing basic linker functionality in *RealView Developer Kit v2.2 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 3-22 and *ROUT* on page 6-85).

There must be at least one `AREA` directive for an assembly.

Copyright © 2005 ARM Limited. All rights reserved.

Example

The following example defines a read-only code section named Example.

```
AREA    Example,CODE,READONLY   ; An example code section.
        ; code
```

6.8.3 END

The END directive informs the assembler that it has reached the end of a source file.

Syntax

```
END
```

Usage

Every assembly language source file must end with END on a line by itself.

If the source file has been included in a parent file by a GET directive, the assembler returns to the parent file and continues assembly at the first line following the GET directive. See *GET or INCLUDE* on page 6-76 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.

 Copyright © 2005 ARM Limited. All rights reserved.

6.8.4 ENTRY

The ENTRY directive declares an entry point to a program.

Syntax

ENTRY

Usage

You must specify at least one ENTRY point for a program. If no ENTRY exists, a warning is generated at link time.

You must not use more than one ENTRY directive in a single source file. Not every source file has to have an ENTRY directive. If more than one ENTRY exists in a single source file, an error message is generated at assembly time.

Example

```
AREA    ARMex, CODE, READONLY
ENTRY                ; Entry point for the application
```


Copyright © 2005 ARM Limited. All rights reserved.

6.8.5 EQU

The EQU directive gives a symbolic name to a numeric constant, a register-relative value or a program-relative value. * is a synonym for EQU.

Syntax

name EQU *expr*{, *type*}

where:

name is the symbolic name to assign to the value.

expr is a register-relative address, a program-relative address, an absolute address, or a 32-bit integer constant.

type is optional. *type* can be any one of:

- ARM
- THUMB
- CODE32
- CODE16
- DATA

You can use *type* only if *expr* is an absolute address. If *name* is exported, the *name* entry in the symbol table in the object file will be marked as ARM, THUMB, CODE32, CODE16, or DATA, according to *type*. This can be used by the linker.

Usage

Use EQU to define constants. This is similar to the use of **#define** to define a constant in C.

See *KEEP* on page 6-80 and *EXPORT or GLOBAL* on page 6-71 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 © 2005 ARM Limited. All rights reserved.

6.8.6 EXPORT or GLOBAL

The EXPORT directive declares a symbol that can be used by the linker to resolve symbol references in separate object and library files. GLOBAL is a synonym for EXPORT.

Syntax

EXPORT {*symbol*}{[WEAK,*attr*]}

where:

symbol is the symbol name to export. The symbol name is case-sensitive. If *symbol* is omitted, all symbols are exported.

[WEAK] means that this instance of *symbol* should only be imported into other sources if no other source exports an alternative instance. If [WEAK] is used without *symbol*, all exported symbols are weak.

[*attr*] is symbol visibility when linked into a dynamic component. By default, symbol binding defines visibility, that is, global and weak symbols are visible to other components and local symbols are not hidden. Valid attributes are:

- DYNAMIC *symbol* is visible to other components, and can be redefined by other components.
- HIDDEN *symbol* cannot be referenced outside the component where it is defined, either directly or indirectly.

  The linker also accepts INTERNAL and currently treats it as HIDDEN. If you specify both, for example:

  EXPORT SymA[WEAK,INTERNAL,HIDDEN]

  the assembler chooses the most restrictive (INTERNAL).
- PROTECTED *symbol* is visible to other components, and cannot be redefined by other components.

Usage

Use EXPORT to give code in other files access to symbols in the current file.

Use the [WEAK] attribute to inform the linker that a different instance of *symbol* takes precedence over this one, if a different one is available from another source. You can use the [WEAK] attribute with any of the symbol visibility attributes.

Symbol visibility can be overridden for duplicate exports. In the following example, the last EXPORT takes precedence for both binding and visibility:

Copyright © 2005 ARM Limited. All rights reserved.

```
        EXPORT  SymA[WEAK]      ; Export as weak-hidden
        EXPORT  SymA[DYNAMIC]   ; SymA becomes non-weak dynamic.
```

See also *IMPORT* on page 6-77.

Example

```
        AREA    Example,CODE,READONLY
        EXPORT  DoAdd           ; Export the function name
                                ; to be used by external
                                ; modules.
DoAdd   ADD     r0,r0,r1
```


Copyright © 2005 ARM Limited. All rights reserved.

6.8.7 EXPORTAS

The EXPORTAS directive enables you to export a symbol to the object file, corresponding to a different symbol in the source file.

Syntax

EXPORTAS *symbol1*, *symbol2*

where:

symbol1 is the symbol name in the source file. *symbol1* must have been defined already. It can be any symbol, including an area name, a label, or a constant.

symbol2 is the symbol name you want to appear in the object file.

The symbol names are case-sensitive.

Usage

Use EXPORTAS to change a symbol in the object file without having to change every instance in the source file.

See also *EXPORT or GLOBAL* on page 6-71.

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 © 2005 ARM Limited. All rights reserved.

6.8.8 EXTERN

The EXTERN directive provides the assembler with a name that is not defined in the current assembly.

EXTERN is very similar to IMPORT, except that the name is not imported if no reference to it is found in the current assembly (see *IMPORT* on page 6-77, and *EXPORT or GLOBAL* on page 6-71).

Syntax

EXTERN *symbol* {[WEAK,*attr*]}

where:

symbol is a symbol name defined in a separately assembled source file, object file, or library. The symbol name is case-sensitive.

[WEAK] prevents the linker generating an error message if the symbol is not defined elsewhere. It also prevents the linker searching libraries that are not already included.

[*attr*] is symbol visibility when linked into a dynamic component. By default, the symbol binding defines visibility, that is, global and weak symbols are visible to other external objects and local symbols are not hidden. Valid attributes are:

- DYNAMIC *symbol* is visible to other components, and can be redefined by other components.
- HIDDEN *symbol* cannot be referenced outside the component where it is defined, either directly or indirectly.

  The linker also accepts INTERNAL and currently treats it as HIDDEN. If you specify both, for example:

  EXTERN SymA[WEAK,INTERNAL,HIDDEN]

  the assembler chooses the most restrictive (INTERNAL).
- PROTECTED *symbol* is visible to other components, and cannot be redefined by other components.

Usage

The name is resolved at link time to a symbol defined in a separate object file. The symbol is treated as a program address. If [WEAK] is not specified, the linker generates an error if no corresponding symbol is found at link time.

 Copyright © 2005 ARM Limited. All rights reserved.

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 © 2005 ARM Limited. All rights reserved.

6.8.9 GET or INCLUDE

The GET directive includes a file within the file being assembled. The included file is assembled at the location of the GET directive. INCLUDE is a synonym for GET.

Syntax

GET *filename*

where:

filename is the name of the file to be included in the assembly. The assembler accepts pathnames in MS-DOS format.

Usage

GET is useful for including macro definitions, EQUs, and storage maps in an assembly. When assembly of the included file is complete, assembly continues at the line following the GET directive.

By default the assembler searches the current place for included files. The current place is the directory where the calling file is located. Use the -i assembler command-line option to add directories to the search path. File names and directory names containing spaces must not be enclosed in double quotes (" ").

The included file can contain additional GET directives to include other files (see *Nesting directives* on page 6-29).

If the included file is in a different directory from the current place, this becomes the current place until the end of the included file. The previous current place is then restored.

GET cannot be used to include object files (see *INCBIN* on page 6-79).

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 © 2005 ARM Limited. All rights reserved.

6.8.10 IMPORT

The IMPORT directive provides the assembler with a name that is not defined in the current assembly.

IMPORT is very similar to EXTERN, except that the name is imported whether or not it is referred to in the current assembly (see *EXTERN* on page 6-74, and *EXPORT or GLOBAL* on page 6-71).

Syntax

IMPORT *symbol* {[WEAK,*attr*]}

where:

symbol is a symbol name defined in a separately assembled source file, object file, or library. The symbol name is case-sensitive.

[WEAK] prevents the linker generating an error message if the symbol is not defined elsewhere. It also prevents the linker searching libraries that are not already included.

[*attr*] is symbol visibility when linked into a dynamic component. By default, symbol binding defines visibility, that is, global and weak symbols are visible to other components and local symbols are not hidden. Valid attributes are:

DYNAMIC *symbol* is visible to other components, and can be redefined by other components.

HIDDEN *symbol* cannot be referenced outside the component where it is defined, either directly or indirectly.

The linker also accepts INTERNAL and currently treats it as HIDDEN. If you specify both, for example:

```
IMPORT SymA[WEAK,INTERNAL,HIDDEN]
```

the assembler chooses the most restrictive (INTERNAL).

PROTECTED *symbol* is visible to other components, and cannot be redefined by other components.

Usage

The name is resolved at link time to a symbol defined in a separate object file. The symbol is treated as a program address. If [WEAK] is not specified, the linker generates an error if no corresponding symbol is found at link time.

Copyright © 2005 ARM Limited. All rights reserved.

If [WEAK] is specified and no corresponding symbol is found at link time:

- If the reference is the destination of a B or BL instruction, the value of the symbol is taken as the address of the following instruction. This makes the B or BL instruction effectively a NOP.
- Otherwise, the value of the symbol is taken as zero.

To avoid trying to access symbols that are not found at link time, use code like the example in *EXTERN* on page 6-74.

 Copyright © 2005 ARM Limited. All rights reserved.

6.8.11 INCBIN

The INCBIN directive includes a file within the file being assembled. The file is included as it is, without being assembled.

Syntax

INCBIN *filename*

where:

filename is the name of the file to be included in the assembly. The assembler accepts pathnames in 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 © 2005 ARM Limited. All rights reserved.

6.8.12 KEEP

The KEEP directive instructs the assembler to retain local symbols in the symbol table in the object file.

Syntax

KEEP {*symbol*}

where:

symbol is the name of the local symbol to keep. If *symbol* is not specified, all local symbols are kept except register-relative symbols.

Usage

By default, the only symbols that the assembler describes in its output object file are:

- exported symbols
- symbols that are relocated against.

Use KEEP to preserve local symbols that can be used to help debugging. Kept symbols appear in the ARM debuggers and in linker map files.

KEEP cannot preserve register-relative symbols (see *MAP* on page 6-17).

Example

```
label   ADC     r2,r3,r4
        KEEP    label       ; makes label available to debuggers
        ADD     r2,r2,r5
```


Copyright © 2005 ARM Limited. All rights reserved.

6.8.13 NOFP

The NOFP directive ensures that there are no floating-point instructions in an assembly language source file.

Syntax

```
NOFP
```

Usage

Use NOFP to ensure that no floating-point instructions are used in situations where there is no support for floating-point instructions either in software or in target hardware.

If a floating-point instruction occurs after the NOFP directive, an `Unknown opcode` error is generated and the assembly fails.

If a NOFP directive occurs after a floating-point instruction, the assembler generates the error:

```
Too late to ban floating point instructions
```

and the assembly fails.

6.8.14 REQUIRE

The REQUIRE directive specifies a dependency between sections.

Syntax

```
REQUIRE label
```

where:

label is the name of the required label.

Usage

Use REQUIRE to ensure that a related section is included, even if it is not directly called. If the section containing the REQUIRE directive is included in a link, the linker also includes the section containing the definition of the specified label.

Copyright © 2005 ARM Limited. All rights reserved.

6.8.15 REQUIRE8 and PRESERVE8

The REQUIRE8 directive specifies that the current file requires eight-byte alignment of the stack. It sets the REQ8 build attribute to inform the linker.

The PRESERVE8 directive specifies that the current file preserves eight-byte alignment of the stack. It sets the PRES8 build attribute to inform the linker.

The linker ensures that any code that requires eight-byte alignment of the stack is only called, directly or indirectly, by code that preserves eight-byte alignment of the stack.

Syntax

```
REQUIRE8 {bool}
```

```
PRESERVE8 {bool}
```

where:

bool is an optional Boolean constant, either {TRUE} or {FALSE}.

Usage

If you are using unaligned data accesses in ARMv6 systems, LDRD and STRD instructions (doubleword transfers) do not require eight-byte alignment. Instead, they require four-byte alignment.

However, on other systems, LDRD and STRD instructions only work correctly if the address they access is eight-byte aligned.

Where required, if your code includes LDRD or STRD transfers to or from the stack, use REQUIRE8 to set the REQ8 build attribute on your file. The assembler gives a warning if you use LDRD or STRD addressed sp relative without specifying REQUIRE8.

Where required, if your code preserves eight-byte alignment of the stack, use PRESERVE8 to set the PRES8 build attribute on your file. If your code does not preserve eight-byte alignment of the stack, use PRESERVE8 {FALSE} to ensure that the PRES8 build attribute is not set.

Note

If you omit both PRESERVE8 and PRESERVE8 {FALSE}, the assembler decides whether to set the PRES8 build attribute or not, by examining instructions that modify the sp. ARM recommends that you specify PRESERVE8 explicitly.

You can enable a warning with:

```
armasm --diag_warning 1546
```

 Copyright © 2005 ARM Limited. All rights reserved.

See *Command syntax* on page 3-2 for details.

This gives you warnings like:

```
"test.s", line 37: Warning: A1546W: Stack pointer update potentially
                                   breaks 8 byte stack alignment
       37 00000044         STMFD    sp!,{r2,r3,lr}
```

Examples

```
REQUIRE8
REQUIRE8     {TRUE}      ; equivalent to REQUIRE8
REQUIRE8     {FALSE}     ; equivalent to absence of REQUIRE8
PRESERVE8    {TRUE}      ; equivalent to PRESERVE8
PRESERVE8    {FALSE}     ; NOT exactly equivalent to absence of PRESERVE8
```


Copyright © 2005 ARM Limited. All rights reserved.

6.8.16 RN

The RN directive defines a register name for a specified register.

Syntax

name RN *expr*

where:

name is the name to be assigned to the register. *name* cannot be the same as any of the predefined names listed in *Predefined register and coprocessor names* on page 3-15.

expr evaluates to a register number from 0 to 15.

Usage

Use RN to allocate convenient names to registers, to help you to remember what you use each register for. Be careful to avoid conflicting uses of the same register under different names.

Examples

```
regname     RN  11  ; defines regname for register 11

sqr4        RN  r6  ; defines sqr4 for register 6
```


Copyright © 2005 ARM Limited. All rights reserved.

6.8.17 ROUT

The ROUT directive marks the boundaries of the scope of local labels (see *Local labels* on page 3-22).

Syntax

{*name*} ROUT

where:

name is the name to be assigned to the scope.

Usage

Use the ROUT directive to limit the scope of local labels. This makes it easier for you to avoid referring to a wrong label by accident. The scope of local labels is the whole area if there are no ROUT directives in it (see *AREA* on page 6-66).

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 © 2005 ARM Limited. All rights reserved.

 Copyright © 2005 ARM Limited. All rights reserved.

Glossary

The items in this glossary are listed in alphabetical order, with any symbols and numerics appearing at the end.

AAPCS *See* Procedure Call Standard for The ARM Architecture.

ABI for the ARM Architecture (base standard) (BSABI)
The ABI for the ARM Architecture is a collection of specifications, some open and some specific to ARM architecture, that regulate the inter-operation of binary code in a range of ARM architecture-based execution environments. The base standard specifies those aspects of code generation that must be standardized to support inter-operation and is aimed at authors and vendors of C and C++ compilers, linkers, and runtime libraries.

American National Standards Institute (ANSI)
An organization that specifies standards for, among other things, computer software. This is superseded by the International Standards Organization.

ANSI *See* American National Standards Institute.

Architecture The term used to identify a group of processors that have similar characteristics.

ARM instruction A word that encodes an operation for an ARM processor operating in ARM state. ARM instructions must be word-aligned.

Copyright © 2005 ARM Limited. All rights reserved.

ARM state A processor that is executing ARM instructions is operating in ARM state. The processor switches to Thumb state (and to recognizing Thumb instructions) when directed to do so by a state-changing instruction such as `BX`, `BLX`.

See also Thumb state.

ARM Toolkit Proprietary ELF (ATPE)
The binary file format used by RealView Developer Kit. ATPE object format is produced by the compiler and assembler tools. The ARM linker accepts ATPE object files and can output an ATPE executable file. RealView Debugger can load only ATPE format images, or binary ROM images produced by the fromELF utility.

See also RealView Developer Suite.

ATPE *See* ARM Toolkit Proprietary ELF.

Big-endian Memory organization where the least significant byte of a word is at a higher address than the most significant byte.

Byte A unit of memory storage consisting of eight bits.

Canonical Frame Address (CFA)
In DWARF, this is an address on the stack specifying where the call frame of an interrupted function is located.

CFA *See* Canonical Frame Address.

Coprocessor An additional processor that is used for certain operations. Usually used for floating-point math calculations, signal processing, or memory management.

CPSR *See* Current Processor Status Register.

Current Processor Status Register (CPSR)
A register containing the current state of control bits and flags.

See also Saved Processor Status Register.

Debug With Arbitrary Record Format (DWARF)
ARM code generation tools generate debug information in DWARF2 format by default. From RVCT v2.2, you can optionally generate DWARF3 format (Draft Standard 9).

Debugger An application that monitors and controls the execution of a second application. Usually used to find errors in the application program flow.

Deprecated A deprecated option or feature is one that you are strongly discouraged from using. Deprecated options and features will not be supported in future versions of the product.

Doubleword A 64-bit unit of information. Contents are taken as being an unsigned integer unless otherwise stated.

 Copyright © 2005 ARM Limited. All rights reserved.

DWARF *See* Debug With Arbitrary Record Format.

ELF *See* Executable and Linking Format.

Executable and Linking Format (ELF)
The industry standard binary file format used by RealView Developer Kit. ELF object format is produced by the ARM object producing tools such as `armcc` and `armasm`. The ARM linker accepts ELF object files and can output either an ELF executable file, or a partially linked ELF object.

Global variables Variables that are accessible to all code in the application.

See also Local variables.

Halfword A 16-bit unit of information. Contents are taken as being an unsigned integer unless otherwise stated.

Hint A hint instruction provides information to the hardware that the hardware can take advantage of. An implementation can choose whether to implement hint instructions or not. If they are not implemented, they execute as `NOP`.

Image An executable file that has been loaded onto a processor for execution.

A binary execution file loaded onto a processor and given a thread of execution. An image can have multiple threads. An image is related to the processor on which its default thread runs.

IMPLEMENTATION DEFINED
Means that the behavior is not architecturally defined, but must be defined and documented by individual implementations.

International Standards Organization (ISO)
An organization that specifies standards for, among other things, computer software. This supersedes the American National Standards Institute.

Interrupt A change in the normal processing sequence of an application caused by, for example, an external signal.

Interworking Producing an application that uses both ARM and Thumb code.

ISO *See* International Standards Organization.

IT block A block of up to four instructions following an *If-Then* (`IT`) instruction. Each instruction in the block is conditional. The conditions for the instructions are either all the same, or some can be the inverse of others.

Jazelle The Jazelle architecture extends the existing ARM architecture to enable direct execution of selected JVM (Java Virtual Machine) opcodes.

Copyright © 2005 ARM Limited. All rights reserved.

Library A collection of assembler or compiler output objects grouped together into a single repository.

Linker Software that produces a single image from one or more source assembler or compiler output objects.

Little-endian Memory organization where the least significant byte of a word is at a lower address than the most significant byte.

Local variable A variable that is only accessible to the subroutine that created it.

See also Global variables.

Memory hint A memory hint instruction enables you to provide advance information to memory systems about future memory accesses, without actually loading or storing any data.

MPCore An integrated Symmetric Multiprocessor system (SMP) delivered as a traditional uniprocessor core. The chip contains up to four ARM1136J-S™ based CPUs with cache coherency.

PIC Position Independent Code.

See also ROPI.

PID Position Independent Data.

See also RWPI.

Procedure Call Standard for the ARM Architecture (AAPCS)
Procedure Call Standard for the ARM Architecture defines how registers and the stack will be used for subroutine calls.

PSR *See* Processor Status Register

Processor Status Register (PSR)
A register containing various control bits and flags.

See also Current Processor Status Register

See also Saved Processor Status Register.

Read-Only Position Independent (ROPI)
Code and read-only data addresses can be changed at runtime.

Read Write Position Independent (RWPI)
Read/write data addresses can be changed at runtime.

 Copyright © 2005 ARM Limited. All rights reserved.

RealView Compilation Tools (RVCT)
RealView Compilation Tools is a suite of tools, together with supporting documentation and examples, that enables you to write and build applications for the ARM family of *RISC* processors.

RealView Developer Kit (RVDK)
RealView Developer Kit is a suite of software development applications, together with supporting documentation and examples, that enables you to write and debug applications for the ARM family of *RISC* processors.

RealView Developer Suite (RVDS)
The latest suite of software development applications, together with supporting documentation and examples, that enable you to write and debug applications for the ARM family of *RISC* processors.

RealView ICE Micro Edition (RVI-ME)
A JTAG-based debug tool for embedded systems.

ROPI *See* Read-Only Position Independent.

RVCT *See* RealView Compilation Tools.

RVDK *See* RealView Developer Kit.

RVDS *See* RealView Developer Suite.

RVI-ME *See* RealView ICE Micro Edition.

RWPI *See* Read Write Position Independent.

Saved Processor Status Register (SPSR)
SPSR. A register that holds a copy of what was in the Current Processor Status Register before the most recent exception. Each exception mode has its own SPSR.

Scope The accessibility of a function or variable at a particular point in the application code. Symbols that have global scope are always accessible. Symbols with local or private scope are only accessible to code in the same subroutine or object.

Section A block of software code or data for an Image.

Semihosting A mechanism whereby the target communicates I/O requests made in the application code to the host system, rather attempting to support the I/O itself.

SIMD *See* Single Instruction, Multiple Data.

Single Instruction, Multiple Data (SIMD)
Single Instruction, Multiple Data (SIMD) instructions perform similar operations on four 8-bit, or two 16-bit, data items held in 32-bit registers.

Copyright © 2005 ARM Limited. All rights reserved.

Software Interrupt (SWI)
: An instruction that causes the processor to call a programmer-specified subroutine. Used by ARM to handle semihosting.

SPSR
: *See* Saved Processor Status Register.

Stack
: The portion of computer memory that is used to record the address of code that calls a subroutine. The stack can also be used for parameters and temporary variables.

SWI
: *See* Software Interrupt.

Target
: The actual target processor, (real or simulated), on which the target application is running.

  The fundamental object in any debugging session. The basis of the debugging system. The environment in which the target software will run. It is essentially a collection of real or simulated processors.

Thumb instruction
: One halfword or two halfwords that encode an operation for an ARM processor operating in Thumb state. Thumb instructions must be halfword-aligned.

Thumb state
: A processor that is executing Thumb instructions is operating in Thumb state. The processor switches to ARM state (and to recognizing ARM instructions) when directed to do so by a state-changing instruction such as `BX`, `BLX`.

  See also ARM state.

TrustZone
: ARM technology-optimized software that provides a secure execution environment to enable trusted programs and data to be separated from the operating system and applications.

UNDEFINED
: An attempt to execute an ****** Unknown value "smallcaps" for Role attribute******undefined* instruction causes an Undefined Instruction exception.

UNPREDICTABLE
: The result of an ****** Unknown value "smallcaps" for Role attribute******unpredictable* instruction cannot be relied upon. ****** Unknown value "smallcaps" for Role attribute******unpredictable* instructions or results must not represent security holes. ****** Unknown value "smallcaps" for Role attribute******unpredictable* instructions must not halt or hang the processor, or any parts of the system.

Vector Floating Point (VFP)
: A standard for floating-point coprocessors where several data values can be processed by a single instruction.

Veneer
: A small block of code used with subroutine calls when there is a requirement to change processor state or branch to an address that cannot be reached in the current processor state.

VFP
: *See* Vector Floating Point.

Copyright © 2005 ARM Limited. All rights reserved.

Word A 32-bit unit of information. Contents are taken as being an unsigned integer unless otherwise stated.

Zero initialized (ZI) R/W memory used to hold variables that do not have an initial value. The memory is normally set to zero on reset.

ZI *See* Zero Initialized.

Copyright © 2005 ARM Limited. All rights reserved.

Copyright © 2005 ARM Limited. All rights reserved.

Index

The items in this index are listed in alphabetical order, with symbols and numerics appearing at the end. The references given are to page numbers.

A

Copyright © 2005 ARM Limited. All rights reserved.

 Copyright © 2005 ARM Limited. All rights reserved.

B

C

D

Copyright © 2005 ARM Limited. All rights reserved.

E

F

 Copyright © 2005 ARM Limited. All rights reserved.

G

H

I

Copyright © 2005 ARM Limited. All rights reserved.

J

K

L

M

 Copyright © 2005 ARM Limited. All rights reserved.

N

O

P

Q

R

S

Copyright © 2005 ARM Limited. All rights reserved.

T

U

V

Copyright © 2005 ARM Limited. All rights reserved.

W

Symbols

Copyright © 2005 ARM Limited. All rights reserved.

 Copyright © 2005 ARM Limited. All rights reserved.