RealView® Compilation Tools
Version 2.1

Assembler Guide
RealView Compilation Tools
Assembler Guide

Copyright © 2002-2004 ARM Limited. All rights reserved.

Release Information

The following changes have been made to this book.

<table>
<thead>
<tr>
<th>Date</th>
<th>Issue</th>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>August 2002</td>
<td>A</td>
<td>Release 1.2</td>
</tr>
<tr>
<td>January 2003</td>
<td>B</td>
<td>Release 2.0</td>
</tr>
<tr>
<td>September 2003</td>
<td>C</td>
<td>Release 2.0.1 for RVDS v2.0</td>
</tr>
<tr>
<td>January 2004</td>
<td>D</td>
<td>Release 2.1 for RVDS v2.1</td>
</tr>
</tbody>
</table>

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 Open Access. This document has no restriction on distribution.

Product Status

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

Web Address

http://www.arm.com
Contents
RealView Compilation Tools Assembler Guide

Preface
About this book .................................................................................................. viii
Feedback ........................................................................................................... xi

Chapter 1  Introduction
1.1 About the RealView Compilation Tools assemblers ................................. 1-2

Chapter 2  Writing ARM and Thumb Assembly Language
2.1 Introduction ................................................................................................. 2-2
2.2 Overview of the ARM architecture ........................................................... 2-3
2.3 Structure of assembly language modules ................................................. 2-13
2.4 Using the C preprocessor ......................................................................... 2-21
2.5 Conditional execution .............................................................................. 2-22
2.6 Loading constants into registers ............................................................... 2-27
2.7 Loading addresses into registers .............................................................. 2-32
2.8 Load and store multiple register instructions ......................................... 2-41
2.9 Using macros ............................................................................................ 2-50
2.10 Describing data structures with MAP and FIELD directives ............... 2-53
2.11 Using frame directives ........................................................................... 2-68

Chapter 3  Assembler Reference
3.1 Command syntax ........................................................................................ 3-2
Glossary
Preface

This preface introduces the documentation for the RealView Compilation Tools (RVCT) assemblers and assembly language. It contains the following sections:

• About this book on page viii
• Feedback on page xi.
About this book

This book provides tutorial and reference information for the RVCT 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 pseudo-instructions and directives available to assembly language programmers, and the ARM®, Thumb®, and Vector Floating-point (VFP) instruction sets.

Intended audience

This book is written for all developers who are producing applications using RVCT. It assumes that you are an experienced software developer and that you are familiar with the ARM development tools as described in RealView Compilation Tools v2.1 Essentials Guide.

Using this book

This book is organized into the following chapters:

Chapter 1 Introduction
Read this chapter for an introduction to the RVCT v2.1 assemblers and assembly language.

Chapter 2 Writing ARM and Thumb 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 Instruction Reference
Read this chapter for reference material on the ARM instruction set.

Chapter 5 Thumb Instruction Reference
Read this chapter for reference material on the Thumb instruction set.

Chapter 6 Vector Floating-point Programming
Read this chapter for reference material on the VFP instruction set, and other VFP-specific assembly language information.
Chapter 7 Directives Reference

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

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.
- monospace: 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.

ARM publications

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

- RealView Compilation Tools v2.1 Essentials Guide (ARM DUI 0202)
- RealView Compilation Tools v2.1 Developer Guide (ARM DUI 0203)
Preface

- RealView Compilation Tools v2.1 Compiler and Libraries Guide (ARM DUI 0205)
- RealView Compilation Tools v2.1 Linker and Utilities Guide (ARM DUI 0206)

For general information on software interfaces and standards supported by ARM, see install_directory\Documentation\Specifications\. In addition, see the following documentation for specific information relating to ARM products:

- RealView ARMulator ISS User Guide (ARM DUI 0207)
- ARM Reference Peripheral Specification (ARM DDI 0062)
- the ARM datasheet or technical reference manual for your hardware device.

Other publications

For a comprehensive introduction to ARM architecture see:

Feedback

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

Feedback on RealView Compilation Tools

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

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

Feedback on this book

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

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

General suggestions for additions and improvements are also welcome.
Chapter 1

Introduction

This chapter introduces the assemblers provided with *RealView*® *Compilation Tools* (RVCT) v2.1. It contains the following section:

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

RVCT has:

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

The language that these assemblers take as input is basically the same. However, there are limitations on what features of the language you can use in the inline assemblers. Refer to the Mixing C, C++, and Assembly Language chapter in RealView Compilation Tools v2.1 Developer Guide for more information on the inline assemblers.

The remainder of this book relates mainly to armasm.
Chapter 2
Writing ARM and Thumb Assembly Language

This chapter provides an introduction to the general principles of writing ARM® and Thumb® assembly language. It contains the following sections:

- **Introduction** on page 2-2
- **Overview of the ARM architecture** on page 2-3
- **Structure of assembly language modules** on page 2-13
- **Using the C preprocessor** on page 2-21
- **Conditional execution** on page 2-22
- **Loading constants into registers** on page 2-27
- **Loading addresses into registers** on page 2-32
- **Load and store multiple register instructions** on page 2-41
- **Using macros** on page 2-50
- **Describing data structures with MAP and FIELD directives** on page 2-53
- **Using frame directives** on page 2-68.
2.1 Introduction

This chapter gives a basic, practical understanding of how to write ARM and Thumb 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. This information is in Chapter 4 ARM Instruction Reference, Chapter 5 Thumb Instruction Reference, and Chapter 6 Vector Floating-point Programming. More information is in the ARM Architecture Reference Manual.

2.1.1 Code examples

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

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

1. Type armasm -g 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 an ELF/DWARF2-compatible debugger with an appropriate debug target.

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

fromelf -c filename.o

See RealView Compilation Tools v2.1 Linker and Utilities Guide for details on armlink and fromelf.
2.2 Overview of the ARM architecture

This section gives a brief overview of the ARM architecture.

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

2.2.1 Architecture versions

The information and examples in this book assume that you are using a processor that implements ARM architecture v3 or above. See ARM Architecture Reference Manual for details of the various architecture versions.

All these processors have a 32-bit addressing range.

2.2.2 ARM and Thumb state

ARMv4T and above define a 16-bit instruction set called the Thumb instruction set. The functionality of the Thumb instruction set is a subset of the functionality of the 32-bit ARM instruction set. Refer to Thumb instruction set overview on page 2-10 for more information.

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 CODE16 and CODE32 directives. Refer to CODE16 and CODE32 on page 7-59 for details.

ARM processors always start executing code in ARM state.
2.2.3 Processor mode

ARM processors support up to seven processor modes, depending on the architecture version. These are:

<table>
<thead>
<tr>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b10000</td>
<td>User</td>
</tr>
<tr>
<td>0b10001</td>
<td>FIQ - Fast Interrupt Request</td>
</tr>
<tr>
<td>0b10010</td>
<td>IRQ - Interrupt Request</td>
</tr>
<tr>
<td>0b10011</td>
<td>Supervisor</td>
</tr>
<tr>
<td>0b10111</td>
<td>Abort</td>
</tr>
<tr>
<td>0b11011</td>
<td>Undefined</td>
</tr>
<tr>
<td>0b11111</td>
<td>System (ARMv4 and above)</td>
</tr>
</tbody>
</table>

All modes except User mode are referred to as **privileged** modes.

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. Refer to the *Handling Processor Exceptions* chapter in *RealView Compilation Tools v2.1 Developer Guide*, and *ARM Architecture Reference Manual* for more information.

2.2.4 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. Refer to *ARM Architecture Reference Manual* for a detailed description of how registers are banked.

The following registers are available in ARMv3 and above:

- 30 general-purpose, 32-bit registers
- The program counter (pc) on page 2-5
- The Current Program Status Register (CPSR) on page 2-5
- Five Saved Program Status Registers (SPSRs) on page 2-6.

**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 the program counter. You can also load the program counter directly using data operation instructions. For example, to return from a subroutine, you can copy the link register into the program counter using:

```asm
MOV pc,lr
```

During execution, r15 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. Refer to *Conditional execution* on page 2-22 for more information.

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

On ARMv5TE, and ARMv6 and above, the CPSR also holds the Q flag (see *The ALU status flags* on page 2-22).

On ARMv6 and above, the CPSR also holds the GE flags (see *Parallel add and subtract* on page 4-83) and the Endianness bit (see *SETEND* on page 4-117).
Five 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. Refer to the Handling Processor Exceptions chapter in RealView Compilation Tools v2.1 Developer Guide for more information.
2.2.5 ARM 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. Some instructions use the least significant bit to determine whether the code being branched to is Thumb code or ARM code.

See Chapter 4 ARM Instruction Reference for detailed information on the syntax of the ARM instruction set.

ARM instructions can be classified into a number of functional groups:
- Branch instructions
- Data processing instructions
- Single register load and store instructions
- Multiple register load and store instructions on page 2-8
- Status register access instructions on page 2-8
- Coprocessor instructions on page 2-8.

Branch instructions

These instructions are used to:
- branch backwards to form loops
- branch forward in conditional structures
- branch to subroutines
- change the processor from ARM state to Thumb state.

Data processing instructions

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

Long multiply instructions (unavailable in some architectures) 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 or an 8-bit unsigned byte. In ARMv4 and above they can also load or store a 16-bit unsigned halfword, or load and sign extend a 16-bit halfword or an 8-bit byte.
Semaphore instructions load and alter a memory semaphore.

**Multiple register load and store instructions**

These instructions load or store any subset of the general-purpose registers from or to memory. Refer to *Load and store multiple register instructions* on page 2-41 for a detailed description of these instructions.

**Status register access instructions**

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

**Coprocessor instructions**

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

The following general points apply to ARM instructions:

- **Conditional execution**
- **Register access**
- **Access to the inline barrel shifter**.

**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.

You can specify whether a data processing instruction sets the state of these 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.

Refer to *Conditional execution* on page 2-22 for a detailed description.

**Register access**

In ARM state, all instructions can access r0 to r14, and most also allow access to 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. Refer to *MRS* on page 4-113 and *MSR* on page 4-114 for more information.

**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 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.

Refer to *Loading constants into registers* on page 2-27 for more information on using the barrel-shifter to generate constants.
2.2.7 Thumb instruction set overview

The functionality of the Thumb instruction set is almost exactly a subset of the functionality of the ARM instruction set. The instruction set is optimized for production by a C or C++ compiler.

All Thumb instructions are 16 bits long and are stored halfword-aligned in memory. Because of this, the least significant bit of the address of an instruction 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.

All Thumb data processing instructions:
- operate on full 32-bit values in registers
- use full 32-bit addresses for data access and for instruction fetches.

Refer to Chapter 5 Thumb Instruction Reference for detailed information on the syntax of the Thumb instruction set, and how Thumb instructions differ from their ARM counterparts.

2.2.8 Thumb instruction capabilities

The following general points apply to Thumb instructions:
- Conditional execution
- Register access
- Access to the barrel shifter on page 2-11.

Conditional execution

The conditional branch instruction is the only Thumb instruction that can be executed conditionally on the value of the ALU status flags in the CPSR. All data processing instructions update these flags, except when one or more high registers are specified as operands to the MOV or ADD instructions. In these cases the flags cannot be updated.

You cannot have any data processing instructions between an instruction that sets a condition and a conditional branch that depends on it. Use a conditional branch over any instruction that you wish to be conditional.

Register access

In Thumb state, most instructions can access only r0 to r7. These are referred to as the low registers.

Registers r8 to r15 are limited access registers. In Thumb state these are referred to as high registers. They can be used, for example, as fast temporary storage.
Refer to Chapter 5 *Thumb Instruction Reference* for a complete list of the Thumb data processing instructions that can access the high registers.

**Access to the barrel shifter**

In Thumb state you can use the barrel shifter only in a separate operation, using an LSL, LSR, ASR, or ROR instruction.

### 2.2.9 Differences between Thumb and ARM instruction sets

The general differences between the Thumb instruction set and the ARM instruction set are dealt with under the following headings:

- **Branch instructions**
- **Data processing instructions**
- **Single register load and store instructions** on page 2-12
- **Multiple register load and store instructions** on page 2-12.

There are no Thumb coprocessor instructions, no Thumb semaphore instructions, and no Thumb instructions to access the CPSR or SPSR.

**Branch instructions**

These instructions are used to:

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

Program-relative branches, particularly conditional branches, are more limited in range than in ARM code, and branches to subroutines can only be unconditional.

**Data processing instructions**

These operate on the general-purpose registers. In many cases, the result of the operation must be put in one of the operand registers, not in a third register. There are fewer data processing operations available than in ARM state. They have limited access to registers r8 to r15.

The ALU status flags in the CPSR are always updated by these instructions except when MOV or ADD instructions access registers r8 to r15. Thumb data processing instructions that access registers r8 to r15 cannot update the flags.
Single register load and store instructions

These instructions load or store the value of a single low register from or to memory. In Thumb state they can only access registers r0 to r7.

Multiple register load and store instructions

LDM and STM load from memory and store to memory any subset of the registers in the range r0 to r7.

PUSH and POP instructions implement a full descending stack using the stack pointer (r13) as the base. In addition to transferring r0 to r7, PUSH can store the link register and POP can load the program counter.
2.3 Structure of assembly language modules

Assembly language is the language that the ARM assembler (armasm) parses and assembles to produce object code. This can be:

- ARM assembly language
- Thumb assembly language
- a mixture of both.

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.

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 program counter plus or minus an offset. This is called program-relative addressing.

Labels can be defined in a map. See Describing data structures with MAP and FIELD directives on page 2-53. You can place the origin of the map in a specified register at runtime, and references to the label use the specified register plus an offset. This is called register-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.

Refer to 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.
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.
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\RVDS\Examples\asm directory. Refer to 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

```assembly
AREA ARMex, CODE, READONLY
ENTRY start ; Name this block of code ARMex
 ; 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 may 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. Refer to the Linker chapter in RealView Compilation Tools v2.1 Linker and Utilities Guide for more information on how the linker places sections.
In an ARM assembly language 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. Refer to AREA on page 7-56 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 2-16 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 2-16 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).

Refer to the Semihosting SWIs chapter in RealView Compilation Tools v2.1 Compiler and Libraries Guide for additional information.

**The END directive**

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

To call subroutines, use a branch and link instruction. The syntax is:

```
BL destination
```

where `destination` is usually the label on the first instruction of the subroutine.

`destination` can also be a program-relative or register-relative expression. Refer to `B` and `BL` on page 4-97 for more information.

The `BL` instruction:
- places the return address in the *link register* (lr)
- sets 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. Refer to the *Using the Procedure Call Standard* chapter in *RealView Compilation Tools v2.1 Developer Guide* for more information.

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\RVDS\Examples\...\asm`. Refer to `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
```
2.3.4 An example Thumb assembly language module

Example 2-3 shows some of the core constituents of a Thumb assembly language module. It is based on subrout.s. It is supplied as thumbsub.s in the \install_directory\RVDS\Examples\...\asm directory. Refer to Code examples on page 2-2 for instructions on how to assemble, link, and execute the example.

Example 2-3

CODE32 and CODE16 directives

These directives instruct the assembler to assemble subsequent instructions as ARM (CODE32) or Thumb (CODE16) instructions. They do not assemble to an instruction to change the processor state at runtime. They only change the assembler state.

The ARM assembler, armasm, starts in ARM mode by default. You can use the -16 option in the command line if you want it to start in Thumb mode.
BX instruction

This instruction is a branch that can change processor state at runtime. The least significant bit of the target address specifies whether it is an ARM instruction (clear) or a Thumb instruction (set). In this example, this bit is set in the ADR pseudo-instruction.
2.4 Using the C preprocessor

You can include the C preprocessor command #include in your assembly language source file. If you do this, you must preprocess the file using the C preprocessor, before using armasm to assemble it. See RealView Compilation Tools v2.1 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 2-4 shows the commands you write to preprocess and assemble a file, sourcefile.s. In this example, the preprocessor outputs a file called preprocessed.s, and armasm assembles preprocessed.s.

Example 2-4 Preprocessing an assembly language source file

```
armcc -cpp -E sourcefile.s > preprocessedfile.s
armasm preprocessedfile.s
```
2.5 Conditional execution

In ARM state, each data processing instruction has an option to update ALU status flags in the Current Program Status Register (CPSR) according to the result of the operation.

Add an $S$ suffix to an ARM data processing instruction to make it update the ALU status flags in the CPSR.

Do not use the $S$ suffix with $CMP$, $CMN$, $TST$, or $TEQ$. These comparison instructions always update the flags. This is their only effect.

In Thumb state, there is no option. All data processing instructions update the ALU status flags in the CPSR, except when one or more high registers are used in MOV and ADD instructions. MOV and ADD cannot update the status flags in these cases.

Almost every ARM instruction can be executed conditionally on the state of the ALU status flags in the CPSR. Refer to Table 2-1 on page 2-23 for a list of the suffixes to add to instructions to make them conditional.

In ARM state, you can:
- update the ALU status flags in the CPSR on the result of a data operation
- execute several other data operations without updating the flags
- execute following instructions or not, according to the state of the flags updated in the first operation.

In Thumb state, most data operations always update the flags, and conditional execution can only be achieved using the conditional branch instruction ($B$). The suffixes for this instruction are the same as in ARM state. No other instruction can be conditional.

2.5.1 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.
- **Q**: ARMv5E, and ARMv6 and later. Sticky flag (see The $Q$ flag on page 4-7).

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

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

The relation of condition code suffixes to the $N$, $Z$, $C$ and $V$ flags is shown in Table 2-1.

<table>
<thead>
<tr>
<th>Suffix</th>
<th>Flags</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>EQ</td>
<td>Z set</td>
<td>Equal</td>
</tr>
<tr>
<td>NE</td>
<td>Z clear</td>
<td>Not equal</td>
</tr>
<tr>
<td>CS/HS</td>
<td>C set</td>
<td>Higher or same (unsigned $\geq$)</td>
</tr>
<tr>
<td>CC/L0</td>
<td>C clear</td>
<td>Lower (unsigned $&lt;$)</td>
</tr>
<tr>
<td>MI</td>
<td>N set</td>
<td>Negative</td>
</tr>
<tr>
<td>PL</td>
<td>N clear</td>
<td>Positive or zero</td>
</tr>
<tr>
<td>VS</td>
<td>V set</td>
<td>Overflow</td>
</tr>
<tr>
<td>VC</td>
<td>V clear</td>
<td>No overflow</td>
</tr>
<tr>
<td>HI</td>
<td>C set and Z clear</td>
<td>Higher (unsigned $&gt;$)</td>
</tr>
<tr>
<td>LS</td>
<td>C clear or Z set</td>
<td>Lower or same (unsigned $\leq$)</td>
</tr>
<tr>
<td>GE</td>
<td>N and V the same</td>
<td>Signed $\geq$</td>
</tr>
<tr>
<td>LT</td>
<td>N and V differ</td>
<td>Signed $&lt;$</td>
</tr>
<tr>
<td>GT</td>
<td>Z clear, N and V the same</td>
<td>Signed $&gt;$</td>
</tr>
<tr>
<td>LE</td>
<td>Z set, N and V differ</td>
<td>Signed $\leq$</td>
</tr>
<tr>
<td>AL</td>
<td>Any</td>
<td>Always. This suffix is normally omitted.</td>
</tr>
</tbody>
</table>

Examples

- **ADD** $r0$, $r1$, $r2$ ; $r0 = r1 + r2$, don't update flags
- **ADDS** $r0$, $r1$, $r2$ ; $r0 = r1 + r2$, and update flags
- **ADDCSS** $r0$, $r1$, $r2$ ; If C flag set then $r0 = r1 + r2$, and update flags
- **CMP** $r0$, $r1$ ; update flags based on $r0$-$r1$. 
2.5.3 Using conditional execution in ARM state

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

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

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

2.5.4 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:

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

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

```
gcd     CMP      r0, r1
BEQ      end
BLT      less
SUB      r0, r0, r1
B        gcd
SUB      r1, r1, r0
B        gcd
end
```
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 2-2 and Table 2-3 on page 2-26 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 2-2 Conditional branches only

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

Total = 13
```
Converting to Thumb

Because \( B \) is the only Thumb instruction that can be executed conditionally, the \( \text{gcd} \) algorithm must be written with conditional branches in Thumb code.

Like the ARM conditional branch implementation, the Thumb code requires seven instructions. However, because Thumb instructions are only 16 bits long, 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 Thumb instruction, whereas each ARM 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. Refer to *ARM Architecture Reference Manual* and the technical reference manuals for individual processors for full information.

---

<table>
<thead>
<tr>
<th>r0: a</th>
<th>r1: b</th>
<th>Instruction</th>
<th>Cycles (ARM7)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>CMP r0, r1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>SUBGT r0, r0, r1</td>
<td>1 (not executed)</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>SUBLT r1, r1, r0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>BNE gcd</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>CMP r0, r1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>SUBGT r0, r0, r1</td>
<td>1 (not executed)</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>SUBLT r1, r1, r0</td>
<td>1 (not executed)</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>BNE gcd</td>
<td>1 (not executed)</td>
</tr>
</tbody>
</table>

Total = 10
2.6 Loading constants into registers

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

Thumb instructions have a similar limitation.

You can load any 32-bit value into a register with a data load, but there are more direct and efficient ways to load many commonly-used constants. You can also include many commonly-used constants directly as operands within data-processing instructions, without a separate load operation at all.

The following sections describe:

- how to use the MOV and MVN instructions to load a range of immediate values, see *Direct loading with MOV and MVN* on page 2-28
- how to use the LDR pseudo-instruction to load any 32-bit constant, see *Loading with LDR Rd, =const* on page 2-29
- how to load floating-point constants, see *Loading floating-point constants* on page 2-31.
2.6.1 Direct loading with MOV and MVN

In ARM state, you can use the MOV and MVN instructions to load a range of eight-bit constant values directly into a register:

- **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. Table 2-4 shows the range of values that this provides.

- **MVN** can load the bitwise complement of these values. The numerical values are -\((n+1)\), where \(n\) are the values given in Table 2-4.

You do not need to calculate the necessary rotation. The assembler performs the calculation for you.

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 cannot be constructed, the assembler reports the error:

Immediate \(n\) out of range for this operation.

The range of values shown in Table 2-4 can also be used as one of the operands in data-processing operations. You cannot use their bitwise complements as operands, and you cannot use them as operands in multiplication operations.

<table>
<thead>
<tr>
<th>Rotate</th>
<th>Binary</th>
<th>Decimal</th>
<th>Step</th>
<th>Hexadecimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>No rotate</td>
<td>00000000000000000000000000000000xxxxxxx</td>
<td>0-255</td>
<td>1</td>
<td>0-0xFF</td>
</tr>
<tr>
<td>Right, 30 bits</td>
<td>000000000000000000000000000000000000000000</td>
<td>0-1020</td>
<td>4</td>
<td>0-0x3FC</td>
</tr>
<tr>
<td>Right, 28 bits</td>
<td>000000000000000000000000000000000000000000</td>
<td>0-4080</td>
<td>16</td>
<td>0-0x3FF0</td>
</tr>
<tr>
<td>Right, 26 bits</td>
<td>000000000000000000000000000000000000000000</td>
<td>0-16320</td>
<td>64</td>
<td>0-0x3FC0</td>
</tr>
<tr>
<td>... ... ... ...</td>
<td>... ...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Right, 8 bits</td>
<td>xxxxxxxx0000000000000000000000000000000000</td>
<td>0-255 x 2^24</td>
<td>2^24</td>
<td>0-0xFF000000</td>
</tr>
<tr>
<td>Right, 6 bits</td>
<td>xxxxxxxx00000000000000000000000000000000xx</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Right, 4 bits</td>
<td>xxxxxxxx00000000000000000000000000000000xx</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Right, 2 bits</td>
<td>xxxxxxxx00000000000000000000000000000000xx</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 2-4 ARM state immediate constants
Direct loading with MOV in Thumb state

In Thumb state you can use the MOV instruction to load constants in the range 0-255. You cannot generate constants outside this range because:

- The Thumb MOV instruction does not provide inline access to the barrel shifter. Constants cannot be right-rotated as they can in ARM state.
- The Thumb MVN instruction can act only on registers and not on constant values. Bitwise complements cannot be directly loaded as they can in ARM state.

If you attempt to use a MOV instruction with a value outside the range 0-255, the assembler reports the error:

Immediate \( n \) out of range for this operation.

2.6.2 Loading with LDR Rd, =const

The LDR Rd, =const pseudo-instruction can construct any 32-bit numeric constant in a single instruction. 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 code 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      r, [pc, #offset to literal pool]
; load register \( r \) 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. Refer to Placing literal pools on page 2-30 for more information.

Refer to LDR ARM pseudo-instruction on page 4-124 for a description of the syntax of the LDR pseudo-instruction.
Placing literal pools

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

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

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

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

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

If the next literal pool is out of range, the assembler generates an error message. In this
case you must use the LTORG directive to place an additional literal pool in the code. Place
the LTORG directive after the failed LDR pseudo-instruction, and within 4KB (ARM) or
1KB (Thumb). Refer to LTORG on page 7-14 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 2-5 shows how this works in practice. It is supplied as loadcon.s in the
install_directory\RVDS\Examples\asm directory. The instructions listed as
comments are the ARM instructions that are generated by the assembler. Refer to Code
examples on page 2-2 for instructions on how to assemble, link, and execute the
example.

Example 2-5

| AREA     | Loadcon, CODE, READONLY          |
| ENTRY    | ; Mark first instruction to execute |
| start    | BL      func1                     |
|          | BL      func2                     |
| stop     | MOV     r0, #0x18                 |
|          | LDR     r1, =0x20026              |
|          | SWI     0x123456                  |
|          | ; angel_SWIreason_ReportException |
|          | ; ADP_Stopped_ApplicationExit     |
|          | ; ARM semihosting SWI             |
|          | ; Branch to first subroutine      |
|          | ; Branch to second subroutine     |
|          | ; MOV R0, #42                     |
|          | ; LDR R1, [PC, #offset to 42]    |
2.6.3 Loading floating-point constants

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

Refer to FLDFDL pseudo-instruction on page 6-36 for details.
2.7 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 2-37.

2.7.1 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 either of the following:

- A program-relative expression, which is a label with an optional offset, where the address of the label is relative to the current pc.
- A register-relative expression, which is a label with an optional offset, where the address of the label is relative to an address held in a specified general-purpose register. Refer to Describing data structures with MAP and FIELD directives on page 2-53 for information on specifying register-relative expressions.

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 offset range is \( \pm 255 \) bytes for an offset to a non word-aligned address, and \( \pm 1020 \) bytes (255 words) for an offset to a word-aligned address. (For Thumb, the address must be word aligned, and the offset must be positive.)

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 range of an ADRL pseudo-instruction is \( \pm 64KB \) for a non word-aligned address and \( \pm 256KB \) for a word-aligned address. (There is no ADRL pseudo-instruction for Thumb.)

ADRL assembles to two instructions, if successful. The assembler generates two instructions even if the address could be loaded in a single instruction.

Refer to Loading addresses with LDR Rd, = label on page 2-37 for information on loading addresses that are outside the range of the ADRL pseudo-instruction.
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, ADR can generate word-aligned addresses only.

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

Example 2-6 shows the type of code generated by the assembler when assembling ADR and ADRL pseudo-instructions. It is supplied as adrlabel.s in the install_directory\RVDS\Examples\...\asm directory. Refer to Code examples on page 2-2 for instructions on how to assemble, link, and execute the example.

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

--- Example 2-6 ---

```
AREA    adrlabel, CODE,READONLY
ENTRY                          ; Mark first instruction to execute
Start
    BL      func                   ; Branch to subroutine
    stop    MOV     r0, #0x18              ; angel_SWI reason_ReportException
    LDR     r1, =0x20026           ; ADP_Stopped_ApplicationExit
    SWI     0x123456               ; ARM semihosting SWI
    LTORG                          ; Create a literal pool
    func        ADR     r0, Start              ; => SUB r0, PC, #offset to Start
                  ADR     r1, DataArea           ; => ADD r1, PC, #offset to DataArea
                  ; ADR   r2, DataArea+4300      ; This would fail because the offset
                  ; cannot be expressed by operand2
                  ; of an ADD
                  ADRL    r2, DataArea+4300      ; => ADD r2, PC, #offset1
                  ; ADD r2, r2, #offset2
                  MOV     pc, lr                 ; Return
DataArea    SPACE   8000                   ; Starting at the current location,
                  ; clears a 8000 byte area of memory
                  ; to zero
END
```
Implementing a jump table with ADR

Example 2-7 on page 2-35 shows ARM code that implements a jump table. It is supplied as jump.s in the install_directory\RVDS\Examples\...\asm directory. Refer to 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.

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:

\[ \text{argument1} = 0 \quad \text{Result} = \text{argument2} + \text{argument3}. \]
\[ \text{argument1} = 1 \quad \text{Result} = \text{argument2} - \text{argument3}. \]

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

**EQU**

Is an assembler directive. It is used to give a value to a symbol. In 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 program counter.
Example 2-7  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                                     ; Label the function
    MOV     r0, #0x18                ; angel_SWIreason_ReportException
    LDR     r1, =0x20026             ; ADP_Stopped_ApplicationExit
    SWI     0x123456                 ; ARM semihosting SWI
    CMP     r0, #num                 ; Treat function code as unsigned integer
    MOVHS   pc, lr                   ; If code is >= num then simply return
        ADR     r3, JumpTable            ; Load address of jump table
        LDR     pc, [r3,r0,LSL#2]          ; Jump to the appropriate routine
JumpTable
        DCD     DoAdd
        DCD     DoSub
DoAdd    ADD     r0, r1, r2               ; Operation 0
        MOV     pc, lr                   ; Return
DoSub    SUB     r0, r1, r2               ; Operation 1
        MOV     pc, lr                   ; Return
END                              ; Mark the end of this file
Converting to Thumb

Example 2-8 shows the implementation of the jump table converted to Thumb code.

Most of the Thumb version is the same as the ARM code. The differences are commented in the Thumb version.

In Thumb state, you cannot:

- increment the base register of LDR and STR instructions
- load a value into the pc using an LDR instruction
- do an inline shift of a value held in a register.

Example 2-8 Thumb code jump table

```assembly
AREA    Jump, CODE, READONLY ; Following code is Thumb code
num     EQU     2
ENTRY
start   MOV     r0, #0
        MOV     r1, #3
        MOV     r2, #2
        BL      arithfunc
stop    MOV     r0, #0x18
        LDR     r1, =0x20026
        SWI     0xAB                     ; Thumb semihosting SWI
arithfunc
        CMP     r0, #num
        BHS     exit                     ; MOV pc, lr cannot be conditional
        ADR     r3, JumpTable
        LSL     r0, r0, #2               ; 3 instructions needed to replace
        LDR     r0, [r3,r0]              ; LDR pc, [r3,r0,LSL#2]
        MOV     pc, r0
ALIGN    ; Ensure that the table is aligned on a
        ; 4-byte boundary
JumpTable
        DCD     DoAdd
        DCD     DoSub
DoAdd     ADD     r0, r1, r2
exit      MOV     pc, lr
DoSub     SUB     r0, r1, r2
        MOV     pc, lr
END
```
2.7.2 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 2-29. 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      r
  
  ; load register n with one word
  ; from the address [pc + offset]

You must ensure that there is a literal pool within range. Refer to Placing literal pools on page 2-30 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 2-9 shows how this works. It is supplied as ldrlabel.s in the install_directory\RVDS\Examples\...\asm directory. Refer to 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 2-9

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
```
LDR  r1, ={Darea + 12}  ; => LDR R1, [PC, #offset into Literal Pool 1]
LDR  r2, ={Darea + 6000} ; => LDR R2, [PC, #offset into Literal Pool 1]
MOV  pc, lr               ; Return
LTORG                        ; Literal Pool 1

func2
LDR  r3, ={Darea + 6000}  ; => LDR R3, [PC, #offset into Literal Pool 1]
; (sharing with previous literal)
; LDR  r4, ={Darea + 6004}  ; If uncommented produces an error as Literal Pool 2 is out of range
MOV  pc, lr               ; Return

Darea  SPACE 8000         ; Starting at the current location, clears a 8000 byte area of memory to zero
END                                ; Literal Pool 2 is out of range of the LDR instructions above
An LDR Rd, =label example: string copying

Example 2-10 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. Refer to DCB on page 7-18 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 2-10  String copy

```
AREA    StrCopy, CODE, READONLY
ENTRY
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
```
Converting to Thumb

There is no post-indexed addressing mode for Thumb LDR and STR instructions. Because of this, you must use an ADD instruction to increment the address register after the LDR and STR instructions. For example:

```
LDRB r2, [r1] ; load register 2
ADD  r1, #1    ; increment the address in
              ; register 1.
```
2.8 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.

Use the -checkreglist assembler command line option to check that registers in register lists are specified in increasing order. Refer to Command syntax on page 3-2 for more information.
2.8.1 ARM LDM and STM instructions

The load (or store) multiple instruction loads (stores) any subset of the 16 general-purpose registers from (to) memory, using a single instruction.

Syntax

The syntax of the LDM instructions is:

\[ \text{LDM}(\text{cond})\text{address-mode} \text{ Rn}(!),\text{reg-list}{^\wedge} \]

where:

- \textit{cond} is an optional condition code. Refer to \textit{Conditional execution} on page 2-22 for more information.

- \textit{address-mode} specifies the addressing mode of the instruction. Refer to \textit{LDM and STM addressing modes} on page 2-43 for details.

- \textit{Rn} is the base register for the load operation. The address stored in this register is the starting address for the load operation. Do not specify \textit{r15} (pc) as the base register.

- \textit{!} specifies base register write back. If this is specified, the address in the base register is updated after the transfer. It is decremented or incremented by one word for each register in the register list.

- \textit{register-list} is a comma-delimited list of symbolic register names and register ranges enclosed in braces. There must be at least one register in the list. Register ranges are specified with a dash. For example: \{r0,r1,r4-r6,pc\}

\[ \text{Do not specify writeback if the base register Rn is in register-list.} \]

- \textit{^\wedge} You must not use this option in User or System mode. For details of its use in privileged modes, see the \textit{Handling Processor Exceptions} chapter in \textit{RealView Compilation Tools v2.1 Developer Guide} and \textit{LDM and STM} on page 4-20.

The syntax of the STM instruction corresponds exactly, except for some details in the effect of the \textit{^\wedge} option.
Usage

See *Implementing stacks with LDM and STM* on page 2-44 and *Block copy with LDM and STM* on page 2-46.

### 2.8.2 LDM and STM addressing modes

There are four different addressing modes. The base register can be incremented or decremented by one word for each register in the operation, and the increment or decrement can occur before or after the operation. The suffixes for these options are:

- **IA**: Increment after.
- **IB**: Increment before.
- **DA**: Decrement after.
- **DB**: Decrement before.

There are alternative addressing mode suffixes that are easier to use for stack operations. See *Implementing stacks with LDM and STM* on page 2-44.
2.8.3 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. Refer to Table 2-5 for a list of stack-oriented suffixes.

<table>
<thead>
<tr>
<th>Stack type</th>
<th>Push</th>
<th>Pop</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full descending</td>
<td>STMFD (STMDB)</td>
<td>LDMFD (LDMIA)</td>
</tr>
<tr>
<td>Full ascending</td>
<td>STMFA (STMIB)</td>
<td>LDMFA (LDMIA)</td>
</tr>
<tr>
<td>Empty descending</td>
<td>STMED (STMDA)</td>
<td>LDMED (LDMIB)</td>
</tr>
<tr>
<td>Empty ascending</td>
<td>STMEA (STMIA)</td>
<td>LDMEA (LDMDB)</td>
</tr>
</tbody>
</table>

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.
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 the pc off the stack at exit, instead of popping lr and then moving that value into the pc. For example:

```
subroutine
STMFD sp!, {r5-r7,lr} ; Push work registers and lr
    ; code
    BL somewhere_else
    ; Code
LDMFD sp!, {r5-r7,pc} ; Pop work registers and pc
```

**Note**

Use this with care in mixed ARM and Thumb systems. In ARMv4T systems, you cannot change state by popping directly into the program counter.

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

See the *Interworking ARM and Thumb* chapter in *RealView Compilation Tools v2.1 Developer Guide* for more information on mixing ARM and Thumb.
2.8.4 Block copy with LDM and STM

Example 2-11 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\RVDS\Examples\asm directory. Refer to Code examples on page 2-2 for instructions on how to assemble, link, and execute the example.

Example 2-11 Block copy

```
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
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 2-12 on page 2-47 lists the block copy module rewritten to use LDM and STM for copying.
Example 2-12

```
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?
STMFD sp!, {r4-r11} ; Save some working registers
octcopy
LDMIA r0!, {r4-r11} ; Load 8 words from the source
STMIA r1!, {r4-r11} ; and put them at the destination
SUBS r3, r3, #1 ; Decrement the counter
BNE octcopy ; ... copy more
LDMDF sp!, {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_SW1reason_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 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
END
```
2.8.5 Thumb LDM and STM instructions

The Thumb instruction set contains the following pairs of multiple-register transfer instructions:

- LDM and STM for block memory transfers
- PUSH and POP for stack operations.

LDM and STM

These instructions can be used to load or store any subset of the low registers from or to memory. The base register is always updated at the end of the multiple register transfer instruction. You must specify the ! character. The only valid suffix for these instructions is IA.

Examples of these instructions are:

- LDMIA   r1!, {r0,r2-r7}
- STMIA   r4!, {r0-r3}

PUSH and POP

These instructions can be used to push any subset of the low registers and (optionally) the link register onto the stack, and to pop any subset of the low registers and (optionally) the pc off the stack. The base address of the stack is held in r13. Examples of these instructions are:

- PUSH   {r0-r3}
- POP    {r0-r3}
- PUSH   {r4-r7,lr}
- POP    {r4-r7,pc}

The optional addition of the lr or pc to the register list provides support for subroutine entry and exit.

The stack is always full descending.

Thumb-state block copy example

The block copy example, Example 2-11 on page 2-46, can be converted into Thumb instructions (see Example 2-13 on page 2-49).

Because the Thumb LDM and STM instructions can access only the low registers, the number of words copied per iteration is reduced from eight to four. In addition, the LDM and STM instructions can be used to carry out the single word at a time copy, because they update the base pointer after each access. If LDR and STR were used for this, separate ADD instructions would be required to update each base pointer.
Example 2-13

```
AREA    Tblock, CODE, READONLY   ; Name this block of code
num     EQU     20                       ; Set number of words to be copied
ENTRY                            ; Mark first instruction to execute
header                                   ; The first instruction to call
MOV     sp, #0x400               ; Set up stack pointer (r13)
ADR     r0, start + 1            ; Processor starts in ARM state,
BX      r0                       ; so small ARM code header used
       ; to call Thumb main program
       ; Some instructions are Thumb
CODE16                           ; Subsequent instructions are Thumb
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
blockcopy
LSR     r3,r2, #2                ; Number of four word multiples
BEQ     copywords                ; Less than four words to move?
PUSH    {r4-r7}                  ; Save some working registers
quadcopy
LDMIA   r0!, {r4-r7}             ; Load 4 words from the source
STMIA   r1!, {r4-r7}             ; and put them at the destination
SUB     r3, #1                   ; Decrement the counter
BNE     quadcopy                 ; ... copy more
POP     {r4-r7}                  ; Don't need these now-restore originals
copywords
MOV     r3, #3                   ; Bottom two bits represent number
AND     r2, r3                   ; ... of odd words left to copy
BEQ     stop                     ; No words left to copy?
wordcopy
LDMIA   r0!, {r3}                ; Load a word from the source and
STMIA   r1!, {r3}                ; store it to the destination
SUB     r2, #1                   ; Decrement the counter
BNE     wordcopy                 ; ... copy more
stop    MOV     r0, #0x18                ; angel_SWIreason_ReportException
        LDR     r1, =0x20026             ; ADP_Stopped_ApplicationExit
        SWI     0xAB                     ; Thumb semihosting SWI

END
```

```
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     0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
```

```
AREA    BlockData, DATA, READWRITE
```
2.9 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.

Refer to MACRO and MEND on page 7-27 for more details.

2.9.1 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
2.9.2 Unsigned integer division macro example

Example 2-14 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 2-14

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" <> ""
    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" <> ""
    ADC     $Div, $Div, $Div     ; Omit next instruction if $Div is null
  ENDIF
  MOV     $Temp, $Temp, LSR #1     ; Halve $Temp,
  CMP     $Temp, $Bot              ; and loop until
  BHS     %b91                     ; less than divisor
MEND
The macro checks that no two parameters use the same register. It also optimizes the code produced if only the remainder is required.

To avoid multiple definitions of labels if DivMod is used more than once in the assembler source, the macro uses local labels (90, 91). Refer to Local labels on page 2-14 for more information.

Example 2-15 shows the code that this macro produces if it is invoked as follows:

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

Example 2-15

```
ASSERT r5 <> r4                   ; Produce an error if the
ASSERT r5 <> r2                   ; registers supplied are
ASSERT r4 <> r2                   ; not all different
ASSERT r0 <> r5                   ; These three only matter if $Div
ASSERT r0 <> r4                   ; is not null ("")
ASSERT r0 <> r2                   ;

ratio
MOV     r2, r4                     ; Put divisor in $Temp
CMP     r2, r5, LSR #1             ; double it until
90      MOVLS   r2, r2, LSL #1             ; 2 * r2 > r5
CMP     r2, r5, LSR #1
BLS     %b90                       ; The b means search backwards
MOV     r0, #0                     ; Initialize quotient
91      CMP     r5, r2               ; Can we subtract r2?
SUBCS   r5, r5, r2                 ; If we can, do so
ADC     r0, r0, r0                 ; Double r0

MOV     r2, r2, LSR #1             ; Halve r2,
CMP     r2, r4                     ; and loop until
BHS     %b91                       ; less than divisor
```
2.10 Describing data structures with MAP and FIELD directives

You can use the MAP and FIELD directives to describe data structures. These directives are always used together.

Data structures defined using MAP and FIELD:
- are easily maintainable
- can be used to describe multiple instances of the same structure
- make it easy to access data efficiently.

The MAP directive specifies the base address of the data structure. Refer to MAP on page 7-15 for more information.

The FIELD directive specifies the amount of memory required for a data item, and can give the data item a label. It is repeated for each data item in the structure. Refer to FIELD on page 7-16 for more information.

Note

No space in memory is allocated when a map is defined. Use define constant directives (for example, DCD) to allocate space in memory.
2.10.1 Relative maps

To access data more than 4KB away from the current instruction, you can use a register-relative instruction, such as:

```
LDR    r4,[r9,#offset]
```

offset is limited to 4096, so r9 must already contain a value within 4KB of the address of the data.

---

**Example 2-16**

```
MAP     0
consta  FIELD   4       ; consta uses four bytes, located at offset 0
constb  FIELD   4       ; constb uses four bytes, located at offset 4
x       FIELD   8       ; x uses eight bytes, located at offset 8
y       FIELD   8       ; y uses eight bytes, located at offset 16
string  FIELD   256     ; string is up to 256 bytes long, starting at offset 24
```

Using the map in Example 2-16, you can access the data structure using the following instructions:

```
MOV     r9,#4096
LDR     r4,[r9,#constb]
```

The labels are relative to the start of the data structure. The register used to hold the start address of the map (r9 in this case) is called the base register.

There are likely to be many LDR or STR instructions accessing data in this data structure.

This map does not contain the location of the data structure. The location of the structure is determined by the value loaded into the base register at runtime.

The same map can be used to describe many instances of the data structure. These can be located anywhere in memory.

There are restrictions on what addresses can be loaded into a register using the MOV instruction. Refer to Loading addresses into registers on page 2-32 for details of how to load arbitrary addresses.

---

**Note**

r9 is the static base register (sb) in the Procedure Call Standard for the ARM Architecture. Refer to the Using the Procedure Call Standard chapter in RealView Compilation Tools v2.1 Developer Guide for more information.
### 2.10.2 Register-based maps

In many cases, you can use the same register as the base register every time you access a data structure. You can include the name of the register in the base address of the map. Example 2-17 shows such a register-based map. The labels defined in the map include the register.

**Example 2-17**

```assembly
MAP     0,r9
consta FIELD   4       ; consta uses four bytes, located at offset 0 (from r9)
constb FIELD   4       ; constb uses four bytes, located at offset 4
x       FIELD   8       ; x uses eight bytes, located at offset 8
y       FIELD   8       ; y uses eight bytes, located at offset 16
string  FIELD   256     ; string is up to 256 bytes long, starting at offset 24
```

Using the map in Example 2-17, you can access the data structure wherever it is:

```assembly
ADR     r9,datasstart
LDR     r4,constb ; => LDR r4,[r9,#4]
```

constb contains the offset of the data item from the start of the data structure, and also includes the base register. In this case the base register is r9, defined in the MAP directive.
2.10.3 Program-relative maps

You can use the program counter (r15) as the base register for a map. In this case, each STM or LDM instruction must be within 4KB of the data item it addresses, because the offset is limited to 4KB. The data structure must be in the same section as the instructions, because otherwise there is no guarantee that the data items will be within range after linking.

Example 2-18 shows a program fragment with such a map. It includes a directive which allocates space in memory for the data structure, and an instruction which accesses it.

Example 2-18

```assembly
datastruc   SPACE   280         ; reserves 280 bytes of memory for datastruc
MAP     datastruc
consta      FIELD   4
constb      FIELD   4
x           FIELD   8
y           FIELD   8
string      FIELD   256
code        LDR     r2,constb   ; => LDR r2,[pc,offset]
```

In this case, there is no need to load the base register before loading the data as the program counter already holds the correct address. (This is not actually the same as the address of the LDR instruction, because of pipelining in the processor. However, the assembler takes care of this for you.)
2.10.4 Finding the end of the allocated data

You can use the FIELD directive with an operand of 0 to label a location within a structure. The location is labeled, but the location counter is not incremented.

The size of the data structure defined in Example 2-19 depends on the values of MaxStrLen and ArrayLen. If these values are too large, the structure overruns the end of available memory.

Example 2-19 uses:
- an EQU directive to define the end of available memory
- a FIELD directive with an operand of 0 to label the end of the data structure.

An ASSERT directive checks that the end of the data structure does not overrun the available memory.

Example 2-19

StartOfData EQU 0x1000
EndOfData EQU 0x2000
MAP StartOfData
Integer FIELD 4
Integer2 FIELD 4
String FIELD MaxStrLen
Array FIELD ArrayLen=8
BitMask FIELD 4
EndOfUsedData FIELD 0
ASSERT EndOfUsedData <= EndOfData
2.10.5 Forcing correct alignment

You are likely to have problems if you include some character variables in the data structure, as in Example 2-20. This is because a lot of words are misaligned.

```
Example 2-20

StartOfData EQU 0x1000
EndOfData EQU 0x2000
   MAP StartOfData
Char FIELD 1
Char2 FIELD 1
Char3 FIELD 1
Integer FIELD 4 ; alignment = 3
Integer2 FIELD 4
String FIELD MaxStrLen
Array FIELD ArrayLen*8
BitMask FIELD 4
EndOfUsedData FIELD 0
   ASSERT EndOfUsedData <= EndOfData
```

You cannot use the ALIGN directive, because the ALIGN directive aligns the current location within memory. MAP and FIELD directives do not allocate any memory for the structures they define.

You could insert a dummy FIELD 1 after Char3 FIELD 1. However, this makes maintenance difficult if you change the number of character variables. You must recalculate the right amount of padding each time.

Example 2-21 on page 2-59 shows a better way of adjusting the padding. The example uses a FIELD directive with a 0 operand to label the end of the character data. A second FIELD directive inserts the correct amount of padding based on the value of the label. An :AND: operator is used to calculate the correct value.

The (-EndOfChars):AND:3 expression calculates the correct amount of padding:

0 if EndOfChars is 0 mod 4;
3 if EndOfChars is 1 mod 4;
2 if EndOfChars is 2 mod 4;
1 if EndOfChars is 3 mod 4.

This automatically adjusts the amount of padding used whenever character variables are added or removed.
Example 2-21

StartOfData EQU 0x1000
EndOfData EQU 0x2000
MAP StartOfData
Char FIELD 1
Char2 FIELD 1
Char3 FIELD 1
EndOfChars FIELD 0
Padding FIELD (-EndOfChars):AND:3
Integer FIELD 4
Integer2 FIELD 4
String FIELD MaxStrLen
Array FIELD ArrayLen*8
BitMask FIELD 4
EndOfUsedData FIELD 0
ASSERT EndOfUsedData <= EndOfData
2.10.6 Using register-based MAP and FIELD directives

Register-based MAP and FIELD directives define register-based symbols. The main uses for register-based symbols:

- defining structures similar to C structures
- gaining faster access to memory sections described by non register-based MAP and FIELD directives.

Defining register-based symbols

Register-based symbols can be very useful, but you must be careful when using them. As a general rule, use them only in the following ways:

- As the location for a load or store instruction to load from or store to. If `Location` is a register-based symbol based on the register `Rb` and with numeric offset, the assembler automatically translates, for example, `LDR Rn,Location` into `LDR Rn,[Rb,#offset]`.

  In an `ADR` or `ADRL` instruction, `ADR Rn,Location` is converted by the assembler into `ADD Rn,Rb,#offset`.

- Adding an ordinary numeric expression to a register-based symbol to get another register-based symbol.

- Subtracting an ordinary numeric expression from a register-based symbol to get another register-based symbol.

- Subtracting a register-based symbol from another register-based symbol to get an ordinary numeric expression. Do not do this unless the two register-based symbols are based on the same register. Otherwise, you have a combination of two registers and a numeric value. This results in an assembler error.

- As the operand of a `:BASE:` or `:INDEX:` operator. These operators are mainly of use in macros.

Other uses usually result in assembler error messages. For example, if you write `LDR Rn,=Location`, where `Location` is register-based, you are asking the assembler to load `Rn` from a memory location that always has the current value of the register `Rb` plus offset in it. It cannot do this, because there is no such memory location.

Similarly, if you write `ADD Rd,Rn,#expression`, and `expression` is register-based, you are asking for a single `ADD` instruction that adds both the base register of the expression and its offset to `Rn`. Again, the assembler cannot do this. You must use two `ADD` instructions to perform these two additions.
Setting up a C-type structure

Using structures in C requires that you:
1. Declare the fields that the structure contains.
2. Generate the structure in memory, and use it.

For example, the following `typedef` statement defines a point structure that contains three `float` fields named `x`, `y` and `z`, but it does not allocate any memory. The second statement allocates three structures of type `Point` in memory, named `origin`, `oldloc`, and `newloc`:

```c
typedef struct Point
{
    float x,y,z;
} Point;
```

Point origin, oldloc, newloc;

The following assembly language code is equivalent to the `typedef` statement above:

```
PointBase   RN      r11
MAP     0,PointBase
Point_x     FIELD   4
Point_y     FIELD   4
Point_z     FIELD   4
```

The following assembly language code allocates space in memory. This is equivalent to the last line of C code:

```
origin  SPACE   12
oldloc  SPACE   12
newloc  SPACE   12
```

You must load the base address of the data structure into the base register before you can use the labels defined in the map. For example:

```
LDR     PointBase,=origin
MOV     r0,#0
STR     r0,Point_x
MOV     r0,#2
STR     r0,Point_y
MOV     r0,#3
STR     r0,Point_z
```

is equivalent to the C code:

```c
origin.x = 0;
origin.y = 2;
origin.z = 3;
```
Making faster access possible

To gain faster access to a section of memory:
1. Describe the memory section as a structure.
2. Use a register to address the structure.

For example, consider the definitions in Example 2-22.

Example 2-22

| StartOfData | EQU | 0x1000 |
| EndOfData   | EQU | 0x2000 |
| MAP         | StartOfData |
| Integer     | FIELD | 4 |
| String      | FIELD | MaxStrLen |
| Array       | FIELD | ArrayLen*8 |
| BitMask     | FIELD | 4 |
| EndOfUsedData | FIELD | 0 |
| ASSERT      | EndOfUsedData <= EndOfData |

If you want the equivalent of the C code:

```c
Integer = 1;
String = "";
BitMask = 0xA000000A;
```

With the definitions from Example 2-22, the assembly language code can be as shown in Example 2-23.

Example 2-23

```assembly
MOV r0,#1
LDR r1,=Integer
STR r0,[r1]
MOV r0,#0
LDR r1,=String
STRB r0,[r1]
MOV r0,#0xA000000A
LDR r1,=BitMask
STRB r0,[r1]
```

Example 2-23 uses LDR pseudo-instructions. Refer to Loading with LDR Rd, =const on page 2-29 for an explanation of these.
Example 2-23 on page 2-62 contains separate LDR pseudo-instructions to load the address of each of the data items. Each LDR pseudo-instruction is converted to a separate instruction by the assembler. However, it is possible to access the entire data section with a single LDR pseudo-instruction. Example 2-24 shows how to do this. Both speed and code size are improved.

Example 2-24

```
AREA    data, DATA
StartOfData    EQU     0x1000
EndOfData      EQU     0x2000
DataAreaBase   RN      r11
MAP     0,DataAreaBase
StartOfUsedData FIELD   0
Integer       FIELD   4
String        FIELD   MaxStrLen
Array         FIELD   ArrayLen*8
BitMask       FIELD   4
EndOfUsedData  FIELD   0
UsedDataLen    EQU     EndOfUsedData - StartOfUsedData
                ASSERT UsedDataLen <= (EndOfData - StartOfData)

AREA    code, CODE
LDR     DataAreaBase,=StartOfData
MOV     r0,#1
STR     r0,Integer
MOV     r0,#0
STRB    r0,String
MOV     r0,#0xA000000A
STRB    r0,BitMask
```

--- Note ---

In this example, the MAP directive is:

```
MAP 0, DataAreaBase
```

not:

```
MAP StartOfData,DataAreaBase
```

The MAP and FIELD directives give the position of the data relative to the DataAreaBase register, not the absolute position. The LDR DataAreaBase,=StartOfData statement provides the absolute position of the entire data section.
If you use the same technique for a section of memory containing memory-mapped I/O (or whose absolute addresses must not change for other reasons), you must take care to keep the code maintainable.

One method is to add comments to the code warning maintainers to take care when modifying the definitions. A better method is to use definitions of the absolute addresses to control the register-based definitions.

Using `MAP offset, reg` followed by `label` `FIELD 0` makes `label` into a register-based symbol with register part `reg` and numeric part `offset`. Example 2-25 shows this.

```
Example 2-25

StartOfIOArea  EQU     0x1000000
SendFlag_Abs   EQU     0x1000000
SendData_Abs   EQU     0x1000004
RcvFlag_Abs    EQU     0x1000008
RcvData_Abs    EQU     0x100000C
IOAreaBase     RN      r11

MAP     (SendFlag_Abs-StartOfIOArea),IOAreaBase
SendFlag        FIELD   0
MAP     (SendData_Abs-StartOfIOArea),IOAreaBase
SendData        FIELD   0
MAP     (RcvFlag_Abs-StartOfIOArea),IOAreaBase
RcvFlag         FIELD   0
MAP     (RcvData_Abs-StartOfIOArea),IOAreaBase
RcvData         FIELD   0

Load the base address with `LDR IOAreaBase,=StartOfIOArea`. This allows the individual locations to be accessed with statements like `LDR R0,RcvFlag` and `STR R4,SendData`.
```
2.10.7 Using two register-based structures

Sometimes you need to operate on two structures of the same type at the same time. For example, if you want the equivalent of the pseudo-code:

\[
\begin{align*}
\text{newloc.x} &= \text{oldloc.x} + \text{(value in r0)}; \\
\text{newloc.y} &= \text{oldloc.y} + \text{(value in r1)}; \\
\text{newloc.z} &= \text{oldloc.z} + \text{(value in r2)};
\end{align*}
\]

The base register needs to point alternately to the oldloc structure and to the newloc one. Repeatedly changing the base register would be inefficient. Instead, use a non register-based map, and set up two pointers in two different registers as in Example 2-26.

Example 2-26

| MAP     | 0         | ; Non-register based relative map used twice, for |
| Pointx  | FIELD 4   | ; old and new data at oldloc and newloc         |
| Pointy  | FIELD 4   | ; oldloc and newloc are labels for              |
| Pointz  | FIELD 4   | ; memory allocated in other sections            |
| code    |           |                                                  |
| ADR     | r8,oldloc |                                                  |
| ADR     | r9,newloc |                                                  |
| LDR     | r3,[r8,Pointx] ; load from oldloc (r8)         |
| ADD     | r3,r3,r0  |                                                  |
| STR     | r3,[r9,Pointx] ; store to newloc (r9)          |
| LDR     | r3,[r8,Pointy] |                                                  |
| ADD     | r3,r3,r1  |                                                  |
| STR     | r3,[r9,Pointy] |                                                  |
| LDR     | r3,[r8,Pointz] |                                                  |
| ADD     | r3,r3,r2  |                                                  |
| STR     | r3,[r9,Pointz] |                                                  |
2.10.8 Avoiding problems with MAP and FIELD directives

Using MAP and FIELD directives can help you to produce maintainable data structures. However, this is only true if the order the elements are placed in memory is not important to either the programmer or the program.

You can have problems if you load or store multiple elements of a structure in a single instruction. These problems arise in operations such as:

- loading several single-byte elements into one register
- using a store multiple or load multiple instruction (STM and LDM) to store or load multiple words from or to multiple registers.

These operations require the data elements in the structure to be contiguous in memory, and to be in a specific order. If the order of the elements is changed, or a new element is added, the program is broken in a way that cannot be detected by the assembler.

There are several methods for avoiding problems such as this.

Example 2-27 shows a sample structure.

Example 2-27

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>MiscBase</td>
<td>RN</td>
<td>r10</td>
</tr>
<tr>
<td></td>
<td>MAP</td>
<td>0,MiscBase</td>
</tr>
<tr>
<td>MiscStart</td>
<td>FIELD</td>
<td>0</td>
</tr>
<tr>
<td>Misc_a</td>
<td>FIELD</td>
<td>1</td>
</tr>
<tr>
<td>Misc_b</td>
<td>FIELD</td>
<td>1</td>
</tr>
<tr>
<td>Misc_c</td>
<td>FIELD</td>
<td>1</td>
</tr>
<tr>
<td>Misc_d</td>
<td>FIELD</td>
<td>1</td>
</tr>
<tr>
<td>MiscEndOfChars</td>
<td>FIELD</td>
<td>0</td>
</tr>
<tr>
<td>MiscPadding</td>
<td>FIELD</td>
<td>(-:INDEX:MiscEndOfChars) :AND: 3</td>
</tr>
<tr>
<td>Misc_I</td>
<td>FIELD</td>
<td>4</td>
</tr>
<tr>
<td>Misc_J</td>
<td>FIELD</td>
<td>4</td>
</tr>
<tr>
<td>Misc_K</td>
<td>FIELD</td>
<td>4</td>
</tr>
<tr>
<td>Misc_data</td>
<td>FIELD</td>
<td>4+20</td>
</tr>
<tr>
<td>MiscEnd</td>
<td>FIELD</td>
<td>0</td>
</tr>
<tr>
<td>MiscLen</td>
<td>EQU</td>
<td>MiscEnd-MiscStart</td>
</tr>
</tbody>
</table>

There is no problem in using LDM and STM instructions for accessing single data elements that are larger than a word (for example, arrays). An example of this is the 20-word element Misc_data. It could be accessed as follows:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ArrayBase         RN R9</td>
<td>ADR ArrayBase, MiscBase</td>
</tr>
<tr>
<td></td>
<td>LDMIA ArrayBase, {R0-R5}</td>
</tr>
</tbody>
</table>
Example 2-27 on page 2-66 loads the first six items in the array Misc_data. The array is a single element and therefore covers contiguous memory locations. No one is likely to want to split it into separate arrays in the future.

However, for loading Misc_I, Misc_J, and Misc_K into registers r0, r1, and r2 the following code works, but might cause problems in the future:

```assembly
ArrayBase  RN  r9
    ADR     ArrayBase, Misc_I
    LDMIA   ArrayBase, {r0-r2}
```

Problems arise if the order of Misc_I, Misc_J, and Misc_K is changed, or if a new element Misc_New is added in the middle. Either of these small changes breaks the code.

If these elements are accessed separately elsewhere, you must not amalgamate them into a single array element. In this case, you must amend the code. The first remedy is to comment the structure to prevent changes affecting this section:

```assembly
Misc_I      FIELD   4    ;  ==} Do not split/reorder
Misc_J      FIELD   4    ;    } these 3 elements, STM
Misc_K      FIELD   4    ;  ==} and LDM instructions used.
```

If the code is strongly commented, no deliberate changes are likely to be made that affect the workings of the program. Unfortunately, mistakes can occur. A second method of catching these problems is to add ASSERT directives just before the STM and LDM instructions to check that the labels are consecutive and in the correct order:

```assembly
ArrayBase  RN  R9
    ; Check that the structure elements
    ; are correctly ordered for LDM
    ASSERT (((Misc_J-Misc_I) = 4) :LAND: ((Misc_K-Misc_J) = 4))
    ADR     ArrayBase, Misc_I
    LDMIA   ArrayBase, {r0-r2}
```

This ASSERT directive stops assembly at this point if the structure is not in the correct order to be loaded with an LDM. Remember that the element with the lowest address is always loaded from, or stored to, the lowest numbered register.
2.11 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.

Refer to Frame directives on page 7-34 for details of these directives.

The assembler uses these directives to insert DWARF2 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. Refer to the Using the Procedure Call Standard chapter in RealView Compilation Tools v2.1 Developer Guide for more information about stack unwinding.

Frame directives do not affect the code produced by armasm.
Chapter 3
Assembler Reference

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

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

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

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

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

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

Invoke the ARM assembler using this command:

\texttt{armasm options inputfile}

where \texttt{options} can be any combination of the following, separated by spaces:

- \texttt{--16} instructs the assembler to interpret instructions as Thumb® instructions. This is equivalent to a \texttt{CODE16} directive at the head of the source file.

- \texttt{--32} instructs the assembler to interpret instructions as ARM instructions. This is the default.

- \texttt{--apcs [qualifiers]} This option specifies whether you are using the \textit{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.

- \texttt{--bige}:\texttt{nd} instructs the assembler to assemble code suitable for a big-endian ARM. The default is \texttt{-littleend}.

- \texttt{--littleend} instructs the assembler to assemble code suitable for a little-endian ARM.

- \texttt{--checkr list} instructs the assembler to check RL\textit{ist}, \texttt{LDM}, and \texttt{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.

- \texttt{--cpu name} sets the target CPU. See \textit{CPU names} on page 3-8.

- \texttt{--depend dependfile} instructs the assembler to save source file dependency lists to \texttt{dependfile}. These are suitable for use with make utilities.

- \texttt{--diag_error tag[, tag, ...]} See \textit{Controlling the output of diagnostic messages} on page 3-11.

- \texttt{--diag_remark tag[, tag, ...]} See \textit{Controlling the output of diagnostic messages} on page 3-11.
--diag_warning tag[, tag, ...]

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

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

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

--diag_style {arm|ide}

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

-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.

-g

instructs the assembler to generate DWARF2 debug tables. For backwards compatibility, the following command line option is permitted, but not required:
    -dwarf2

-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 7-66).

--keep

instructs the assembler to keep local labels in the symbol table of the object file, for use by the debugger (see KEEP on page 7-69).
--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.

--nocache

turns off source caching. By default the assembler caches source files on the first pass and reads them from memory on the second pass.

--noesc

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

--noexceptions

See Controlling exception table generation on page 3-13.

--noexceptions_unwind

See Controlling exception table generation on page 3-13.

--noregs

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.

--nowarn

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.

--unsafe

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 Compilation Tools v2.1 Compiler and Libraries Guide.
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

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 RealView Compilation Tools v2.1 Developer Guide for more information about the AAPCS.

```
--apcs /none
```

specifies that `inputfile` does not use AAPCS. AAPCS registers are not set up. Qualifiers are not allowed.

**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:

```
/interwork
```

specifies that the code in `inputfile` is suitable for ARM/Thumb interworking. See RealView Compilation Tools v2.1 Developer Guide for information on interworking.

```
/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`.

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

/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:

```bash
--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 run-time. 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 3, 4T, or 5TE, or part numbers such as ARM7TDMI®. See ARM Architecture Reference Manual for information about the architectures. The default is ARM7TDMI. See also Obtaining a list of valid FPU names on page 3-9.

See the C and C++ Compilers chapter in RealView Compilation Tools v2.1 Compiler and Libraries Guide for details of the effect on software library selection at link time.

Obtaining a list of valid CPU names

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

```
armasm --cpu list
```

3.1.5 FPU names

There is an option to specify the FPU name:

```
--fpu name
```

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:

<table>
<thead>
<tr>
<th>name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>Selects no floating-point option. This makes your assembled object file compatible with any other object file.</td>
</tr>
<tr>
<td>vfp</td>
<td>This is a synonym for vfpv2.</td>
</tr>
<tr>
<td>vfpv2</td>
<td>Selects hardware vector floating-point unit conforming to architecture VFPv2.</td>
</tr>
<tr>
<td>softvfp</td>
<td>Selects software floating-point library (FPLib) with pure-endian doubles. This is the default if no <code>-fpu</code> option is specified.</td>
</tr>
</tbody>
</table>
softvfp+vfp
Selects hardware Vector Floating Point unit.
To armasm, this is identical to -fpu vfpv2.

softvfp+vfpv2
Selects hardware Vector Floating Point unit.
To armasm, this is identical to -fpu vfpv2.

fpa
Selects hardware Floating Point Accelerator (but see note below).

softfpa
Selects software floating-point library with mixed-endian doubles
(but see note below).

Note
The use of fpa and softfpa is deprecated. Support will be removed in future releases of
RVDS.

See the C and C++ Compilers chapter in RealView Compilation Tools v2.1 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

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 allow aligned loads and saves of bytes, halfwords and words.
attributes modify the default. They can be any one of the following:

+L41       Allow unaligned LDRs.
-L22       Disallow halfword loads.
-S22       Disallow halfword stores.
-L22-S22   Disallow halfword loads and stores.
3.1.7 Pre-executing a SET directive

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

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

You must enclose `directive` in quotes. See `SETA, SETL, and SETS` on page 7-7.

The assembler also executes a corresponding `GBLL`, `GBLS`, or `GBLA` directive to define the variable before setting its value.

The variable name is case-sensitive.

**Note**

The command line interface of your system might require you to enter special character combinations, such as `\"`, to include strings in `directive`. Alternatively, you can use `--via file` to include a `--predefine` argument. The command line interface does not alter arguments from `--via` files.

3.1.8 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.

**Note**

Avoiding large multiple register transfers increases code size and decreases performance slightly.

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

3.1.9 Listing output to a file

Use the following option to list output to a file:

```
--list [listingfile]
```

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

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

```
--noterse      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 six options that control the output of diagnostic messages:

```
--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.
--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
Allows 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 3-35).

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:

armcc --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>
<thead>
<tr>
<th>Severity</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Catastrophic error</td>
<td>Catastrophic errors indicate problems that cause the assembly to stop. These errors include command-line errors, internal errors, and missing include files. If multiple source files are being assembled, then no further source files are assembled.</td>
</tr>
<tr>
<td>Error</td>
<td>Errors indicate violations in the syntactic or semantic rules of assembler language. Assembly continues, but object code is not generated.</td>
</tr>
<tr>
<td>Warning</td>
<td>Warnings indicate unusual conditions in your code that might indicate a problem. Assembly continues, and object code is generated unless any problems with an Error severity are detected.</td>
</tr>
<tr>
<td>Remark</td>
<td>Remarks indicate common, but not recommended, use of assembler language. These diagnostics are not issued by default. Assembly continues, and object code is generated unless any problems with an Error severity are detected.</td>
</tr>
</tbody>
</table>
3.1.11 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.

--noexceptions
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.

--noexceptions_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 and FRAME UNWIND OFF on page 7-44.

Unwind tables

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

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

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

The assembler can generate nounwind table entries for all functions and non-functions. The assembler can generate an unwind table for a function only if the function contains sufficient FRAME directives to describe the use of the stack within the function. Functions must conform to the conditions set out in the ehabi document EABI-2001-0003 section 9.1 Constraints on Use. If the assembler cannot generate an unwind table it generates a nounwind table.
3.2 Format of source lines

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

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

All three sections of the source line are optional.

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

You can write directives in all 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).
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:

- f0-f7 and F0-F7 (FPA registers, deprecated)
- s0-s31 and S0-S31 (VFP single-precision registers)
- d0-d15 and D0-D15 (VFP double-precision registers).

3.3.4 Predeclared coprocessor names

The following coprocessor names and coprocessor register names are predeclared:

- p0-p15 (coprocessors 0-15)
- c0-c15 (coprocessor registers 0-15).
3.4 Built-in variables and constants

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

Table 3-2 Built-in variables

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ARCHITECTURE}</td>
<td>Holds the name of the selected ARM architecture.</td>
</tr>
<tr>
<td>{AREANAME}</td>
<td>Holds the name of the current AREA.</td>
</tr>
<tr>
<td>{ARMASM_VERSION}</td>
<td>Holds an integer that increases with each version. See also Determining the armasm version at assembly time on page 3-17</td>
</tr>
<tr>
<td>ads$vversion</td>
<td>Has the same value as {ARMASM_VERSION}, see above.</td>
</tr>
<tr>
<td>{CODESIZE}</td>
<td>Is a synonym for {CONFIG}.</td>
</tr>
<tr>
<td>{COMMANDLINE}</td>
<td>Holds the contents of the command line.</td>
</tr>
<tr>
<td>{CONFIG}</td>
<td>Has the value 32 if the assembler is assembling ARM code, or 16 if it is assembling Thumb code.</td>
</tr>
<tr>
<td>{CPU}</td>
<td>Holds the name of the selected cpu. The default is ARM7TDMI. If an architecture was specified in the command line --cpu option, {CPU} holds the value &quot;Generic ARM&quot;.</td>
</tr>
<tr>
<td>{ENDIAN}</td>
<td>Has the value big if the assembler is in big-endian mode, or little if it is in little-endian mode.</td>
</tr>
<tr>
<td>{FPU}</td>
<td>Holds the name of the selected fpu. The default is SoftVFP.</td>
</tr>
<tr>
<td>{INPUTFILE}</td>
<td>Holds the name of the current source file.</td>
</tr>
<tr>
<td>{INTER}</td>
<td>Has the value True if /inter is set. The default is False.</td>
</tr>
<tr>
<td>{LINENUM}</td>
<td>Holds an integer indicating the line number in the current source file.</td>
</tr>
<tr>
<td>{NOSWST}</td>
<td>Has the value True if /noswst is set. The default is False.</td>
</tr>
<tr>
<td>{OPT}</td>
<td>Value of the currently-set listing option. The OPT directive can be used to save the current listing option, force a change in it, or restore its original value.</td>
</tr>
<tr>
<td>{PC} or .</td>
<td>Address of current instruction.</td>
</tr>
<tr>
<td>{PCSTOREOFFSET}</td>
<td>Is the offset between the address of the STR pc,[,]... or STM Rb,[,]... pc instruction and the value of pc stored out. This varies depending on the CPU or architecture specified.</td>
</tr>
<tr>
<td>{ROPI}</td>
<td>Has the value True if /ropi is set. The default is False.</td>
</tr>
<tr>
<td>{RWPI}</td>
<td>Has the value True if /ropi is set. The default is False.</td>
</tr>
<tr>
<td>{SWST}</td>
<td>Has the value True if /swst is set. The default is False.</td>
</tr>
<tr>
<td>{VAR} or @</td>
<td>Current value of the storage area location counter.</td>
</tr>
</tbody>
</table>
Built-in variables cannot be set using the SETA, SETL, or SETS directives. They can be used in expressions or conditions, for example:

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

|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

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

#### 3.4.1 Determining the armasm version at assembly time

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

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

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

You can use symbols to represent variables, addresses, and numeric constants. Symbols representing addresses are also called labels. See:
- 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.
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 7-4
- LCLA, LCLL, and LCLS on page 7-6
- SETA, SETL, and SETS on page 7-7.

3.5.3 Numeric constants

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

Use the EQU directive to define constants (see EQU on page 7-62). 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.
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$b000000E
s2 SETS "abc"
fixup SETS "|xy$s2.z|" ; fixup now has value |xyabcz|
|CSS$code| MOV r4,#16 ; but the label here is CSS$code
3.5.5 Labels

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

Program-relative labels

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

- DCB on page 7-18
- DCD and DCDU on page 7-19
- DCFD and DCFDU on page 7-21
- DCFS and DCFSU on page 7-22
- DCI on page 7-23
- DCQ and DCQU on page 7-24
- DCW and DCWU on page 7-25.

Register-relative labels

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

- MAP on page 7-15
- SPACE on page 7-17
- DCDO on page 7-20
- EQU on page 7-62.

Absolute addresses

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

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

Local labels are typically used for loops and conditional code within a routine, or for
small subroutines that are only used locally. They are particularly useful in macros (see
MACRO and MEND on page 7-27).

Use the ROUT directive to limit the scope of local labels (see ROUT on page 7-74). 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.

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.
3.6 Expressions, literals, and operators

This section contains the following subsections:

- String expressions on page 3-25
- String literals on page 3-25
- Numeric expressions on page 3-26
- Numeric literals on page 3-27
- Floating-point literals on page 3-28
- Register-relative and program-relative expressions on page 3-29
- Logical expressions on page 3-29
- Logical literals on page 3-29
- Operator precedence on page 3-30
- Unary operators on page 3-32
- Binary operators on page 3-35.
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 3-35
- SETA, SETL, and SETS on page 7-7.

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

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 allowed, unless -noesc is specified (see Command syntax on page 3-2).

Examples

abc SETS "this string contains only one " double quote"
def SETS "this string contains only one $$ dollar symbol"
3.6.3 Numeric expressions

Numeric expressions consist of combinations of numeric constants, numeric variables, ordinary numeric literals, binary operators, and parentheses. See:
- Numeric constants on page 3-19
- Variables on page 3-19
- Numeric literals on page 3-27
- Binary operators on page 3-35
- SETA, SETL, and SETS on page 7-7.

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

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

**Example**

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

Numeric literals can take any of the following forms:

- decimal-digits
- 0xhexadecimal-digits
- &hexadecimal-digits
- n_base-n-digits
- 'character'

where

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

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

Examples

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

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

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

where

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 &FF0000000000000 ; Minus infinity
3.6.6 Register-relative and program-relative expressions

A register-relative expression evaluates to a named register plus or minus a numeric constant (see MAP on page 7-15).

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 3-38).

Relations consist of combinations of variables, literals, constants, or expressions with appropriate relational operators (see Relational operators on page 3-37).

3.6.8 Logical literals

The logical literals are:

- \{TRUE\}
- \{FALSE\}.
3.6.9 Operator precedence

The assembler includes an extensive set of operators for use in expressions. Many of the operators resemble their counterparts in high-level languages such as C (see Unary operators on page 3-32 and Binary operators on page 3-35).

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>
<thead>
<tr>
<th>armasm precedence</th>
<th>equivalent C operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>unary operators</td>
<td>unary operators</td>
</tr>
<tr>
<td>* / :MOD:</td>
<td>* / %</td>
</tr>
<tr>
<td>string manipulation</td>
<td>n/a</td>
</tr>
<tr>
<td>:SHL: :SHR: :ROR: :ROL:</td>
<td>&lt;&lt; &gt;&gt;</td>
</tr>
</tbody>
</table>
The highest precedence operators are at the top of the list.
The highest precedence operators are evaluated first.
Operators of equal precedence are evaluated from left to right.

### Table 3-5 Operator precedence in armasm

<table>
<thead>
<tr>
<th>armasm precedence</th>
<th>equivalent C operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>= &gt; &gt;= &lt; &lt;= /= &lt;&gt;</td>
<td>== &gt; &gt;= &lt; &lt;= !=</td>
</tr>
</tbody>
</table>

### Table 3-4 Operator precedence in C

<table>
<thead>
<tr>
<th>C precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>unary operators</td>
</tr>
<tr>
<td>* / %</td>
</tr>
<tr>
<td>+ - (as binary operators)</td>
</tr>
<tr>
<td>&lt;&lt; &gt;&gt;</td>
</tr>
<tr>
<td>&lt;&lt;= == !=</td>
</tr>
<tr>
<td>&amp;</td>
</tr>
<tr>
<td>^</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>&amp;&amp;</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
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>
<thead>
<tr>
<th>Operator</th>
<th>Usage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>:CHR:</td>
<td>:CHR:A</td>
<td>Returns the character with ASCII code A.</td>
</tr>
<tr>
<td>:LOWERCASE:</td>
<td>:LOWERCASE:string</td>
<td>Returns the given string, with all uppercase characters converted to lowercase.</td>
</tr>
<tr>
<td>:REVERSE_CC:</td>
<td>:REVERSE_CC:cond_code</td>
<td>Returns the inverse of the condition code in cond_code.</td>
</tr>
<tr>
<td>:STR:</td>
<td>:STR:A</td>
<td>Returns an 8-digit hexadecimal string corresponding to a numeric expression, or the string &quot;T&quot; or &quot;F&quot; if used on a logical expression.</td>
</tr>
<tr>
<td>:UPPERCASE:</td>
<td>:UPPERCASE:string</td>
<td>Returns the given string, with all lowercase characters converted to uppercase.</td>
</tr>
</tbody>
</table>
Table 3-7 lists the unary operators that return numeric values.

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

<table>
<thead>
<tr>
<th>Operator</th>
<th>Alias</th>
<th>Usage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>?</td>
<td>?A</td>
<td>Number of bytes of executable code generated by line defining</td>
<td>Symbol A.</td>
</tr>
<tr>
<td>+ and -</td>
<td></td>
<td></td>
<td>Unary plus. Unary minus. + and – can act on numeric and program-relative expressions.</td>
</tr>
<tr>
<td>:BASE:</td>
<td>:BASE:A</td>
<td>If A is a pc-relative or register-relative expression, :BASE: returns</td>
<td>the number of its register component. :BASE: is most useful in macros.</td>
</tr>
<tr>
<td>:CC_ENCODING:</td>
<td>:CC_ENCODING:cond_code</td>
<td>Returns the numeric value of the condition code in cond_code.</td>
<td></td>
</tr>
<tr>
<td>:DEF:</td>
<td>:DEF:A</td>
<td>{TRUE} if A is defined, otherwise {FALSE}.</td>
<td></td>
</tr>
<tr>
<td>:INDEX:</td>
<td>:INDEX:A</td>
<td>If A is a register-relative expression, :INDEX: returns the offset</td>
<td>from that base register. :INDEX: is most useful in macros.</td>
</tr>
<tr>
<td>:LEN:</td>
<td>:LEN:A</td>
<td>Length of string A.</td>
<td></td>
</tr>
<tr>
<td>:LNOT:</td>
<td>:LNOT:A</td>
<td>Logical complement of A.</td>
<td></td>
</tr>
<tr>
<td>:NOT:</td>
<td>~ :NOT:A</td>
<td>Bitwise complement of A.</td>
<td></td>
</tr>
<tr>
<td>:RCONST:</td>
<td>:RCONST:Rn</td>
<td>Number of register, 0-15 corresponding to r0-r15.</td>
<td></td>
</tr>
<tr>
<td>:SB_OFFSET_11_0:</td>
<td>:SB_OFFSET_11_0: label</td>
<td>Least-significant 12 bytes of (label – sb).</td>
<td></td>
</tr>
</tbody>
</table>
Example of use of :SB_OFFSET_19_12: and :SB_OFFSET_11_0

MyIndex EQU 0
    AREA area1, CODE
    LDR IP, [SB, #0]
    LDR IP, [IP, #MyIndex]
    ADD IP, IP, # :SB_OFFSET_19_12: label
    LDR PC, [IP, # :SB_OFFSET_11_0: label]

    AREA area2, DATA
    label
    IMPORT FunctionAddress
    DCD FunctionAddress
    END

These operators can only be used in ADD and LDR instructions. They can only be used in the way shown.
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>
<thead>
<tr>
<th>Operator</th>
<th>Alias</th>
<th>Usage</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>A*B</td>
<td>Multiply</td>
<td></td>
</tr>
<tr>
<td>/</td>
<td>A/B</td>
<td>Divide</td>
<td></td>
</tr>
<tr>
<td>%</td>
<td>A:MOD:B</td>
<td>A modulo B</td>
<td></td>
</tr>
</tbody>
</table>

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

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

Table 3-10 shows the shift operators.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Alias</th>
<th>Usage</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>:ROL:</td>
<td>A:ROL:B</td>
<td>Rotate A left by B bits</td>
<td></td>
</tr>
<tr>
<td>:ROR:</td>
<td>A:ROR:B</td>
<td>Rotate A right by B bits</td>
<td></td>
</tr>
<tr>
<td>:SHL:</td>
<td>&lt;&lt;</td>
<td>A:SHL:B</td>
<td>Shift A left by B bits</td>
</tr>
<tr>
<td>:SHR:</td>
<td>&gt;&gt;</td>
<td>A:SHR:B</td>
<td>Shift A right by B bits</td>
</tr>
</tbody>
</table>

--- Note ---

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

Addition, subtraction, and logical operators

Addition and subtraction operators act on numeric expressions.

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

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

<table>
<thead>
<tr>
<th>Operator</th>
<th>Alias</th>
<th>Usage</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>A+B</td>
<td>Add A to B</td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>A-B</td>
<td>Subtract B from A</td>
<td></td>
</tr>
<tr>
<td>:EOR:</td>
<td>^</td>
<td>A:EOR:B</td>
<td>Bitwise Exclusive OR of A and B</td>
</tr>
<tr>
<td>:OR:</td>
<td></td>
<td></td>
<td>A:OR:B</td>
</tr>
</tbody>
</table>
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>
<thead>
<tr>
<th>Operator</th>
<th>Aliases</th>
<th>Usage</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>==</td>
<td>A=B</td>
<td>A equal to B</td>
</tr>
<tr>
<td>&gt;</td>
<td></td>
<td>A&gt;B</td>
<td>A greater than B</td>
</tr>
<tr>
<td>&gt;=</td>
<td></td>
<td>A&gt;=B</td>
<td>A greater than or equal to B</td>
</tr>
<tr>
<td>&lt;</td>
<td></td>
<td>A&lt;B</td>
<td>A less than B</td>
</tr>
<tr>
<td>&lt;=</td>
<td></td>
<td>A&lt;=B</td>
<td>A less than or equal to B</td>
</tr>
<tr>
<td>/=</td>
<td>&lt;&gt;</td>
<td>A/=B</td>
<td>A not equal to B</td>
</tr>
<tr>
<td>!=</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Boolean operators

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

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

Table 3-13 shows the Boolean operators.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Usage</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>:LAND:</td>
<td>A:LAND:B</td>
<td>Logical AND of A and B</td>
</tr>
<tr>
<td>:LEOR:</td>
<td>A:LEOR:B</td>
<td>Logical Exclusive OR of A and B</td>
</tr>
<tr>
<td>:LOR:</td>
<td>A:LOR:B</td>
<td>Logical OR of A and B</td>
</tr>
</tbody>
</table>
Chapter 4
ARM Instruction Reference

This chapter describes the ARM® instructions that are supported by the ARM assembler. It contains the following sections:

- Conditional execution on page 4-6
- ARM Memory access instructions on page 4-8
- ARM general data processing instructions on page 4-32
- ARM multiply instructions on page 4-51
- ARM saturating instructions on page 4-77
- ARM parallel instructions on page 4-82
- ARM packing and unpacking instructions on page 4-90
- ARM branch instructions on page 4-96
- Coprocessor instructions on page 4-102
- Miscellaneous ARM instructions on page 4-111
- ARM pseudo-instructions on page 4-120.

See to Table 4-1 on page 4-2 to locate individual instructions and pseudo-instructions.
Table 4-1 Location of ARM instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Brief description</th>
<th>Page</th>
<th>Architecturea</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC, ADD</td>
<td>Add with carry, Add</td>
<td>page 4-36</td>
<td>All</td>
</tr>
<tr>
<td>ADR pseudo-instruction</td>
<td>Load program-relative or register-relative address (short range)</td>
<td>page 4-121</td>
<td>All</td>
</tr>
<tr>
<td>ADRL pseudo-instruction</td>
<td>Load program-relative or register-relative address into a register (medium range)</td>
<td>page 4-122</td>
<td>All</td>
</tr>
<tr>
<td>AND</td>
<td>Logical AND</td>
<td>page 4-39</td>
<td>All</td>
</tr>
<tr>
<td>B</td>
<td>Branch</td>
<td>page 4-97</td>
<td>All</td>
</tr>
<tr>
<td>BIC</td>
<td>Bit clear</td>
<td>page 4-39</td>
<td>All</td>
</tr>
<tr>
<td>BKPT</td>
<td>Breakpoint</td>
<td>page 4-118</td>
<td>5</td>
</tr>
<tr>
<td>BL</td>
<td>Branch with link</td>
<td>page 4-97</td>
<td>All</td>
</tr>
<tr>
<td>BLX</td>
<td>Branch, link and exchange</td>
<td>page 4-99</td>
<td>STb</td>
</tr>
<tr>
<td>BX</td>
<td>Branch and exchange</td>
<td>page 4-98</td>
<td>4Tb</td>
</tr>
<tr>
<td>CDP, CDP2</td>
<td>Coprocessor data operation</td>
<td>page 4-103</td>
<td>2, 5</td>
</tr>
<tr>
<td>CLZ</td>
<td>Count leading zeroes</td>
<td>page 4-47</td>
<td>5</td>
</tr>
<tr>
<td>CMN, CMP</td>
<td>Compare negative, Compare</td>
<td>page 4-43</td>
<td>All</td>
</tr>
<tr>
<td>CPS</td>
<td>Change processor state</td>
<td>page 4-115</td>
<td>6</td>
</tr>
<tr>
<td>CPY</td>
<td>Copy</td>
<td>page 4-41</td>
<td>6</td>
</tr>
<tr>
<td>EOR</td>
<td>Exclusive OR</td>
<td>page 4-39</td>
<td>All</td>
</tr>
<tr>
<td>LDC, LDC2</td>
<td>Load coprocessor</td>
<td>page 4-107</td>
<td>2, 5</td>
</tr>
<tr>
<td>LDM</td>
<td>Load multiple registers</td>
<td>page 4-20</td>
<td>All</td>
</tr>
<tr>
<td>LDR</td>
<td>Load register</td>
<td>page 4-8</td>
<td>All</td>
</tr>
<tr>
<td>LDR pseudo-instruction</td>
<td>Load register pseudo-instruction</td>
<td>page 4-120</td>
<td>All</td>
</tr>
<tr>
<td>LDREX</td>
<td>Load register exclusive</td>
<td>page 4-28</td>
<td>6</td>
</tr>
<tr>
<td>MAR</td>
<td>Move from registers to 40-bit accumulator</td>
<td>page 4-119</td>
<td>XScalec</td>
</tr>
<tr>
<td>MCR, MCR2, MCRR, MCRR2</td>
<td>Move from register(s) to coprocessor</td>
<td>page 4-104</td>
<td>2, 5, 5Ea, 6</td>
</tr>
<tr>
<td>MIA, MIAPH, MIaxy</td>
<td>Multiply with internal 40-bit accumulate</td>
<td>page 4-75</td>
<td>XScale</td>
</tr>
</tbody>
</table>
### Table 4-1 Location of ARM instructions (continued)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Brief description</th>
<th>Page</th>
<th>Architecture&lt;sup&gt;a&lt;/sup&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>MLA</td>
<td>Multiply accumulate</td>
<td>page 4-53</td>
<td>2</td>
</tr>
<tr>
<td>MOV</td>
<td>Move</td>
<td>page 4-41</td>
<td>All</td>
</tr>
<tr>
<td>MRA</td>
<td>Move from 40-bit accumulator to registers</td>
<td>page 4-119</td>
<td>XScale</td>
</tr>
<tr>
<td>MRC, MRC2</td>
<td>Move from coprocessor to register</td>
<td>page 4-105</td>
<td>2, 5</td>
</tr>
<tr>
<td>MRR, MRR2</td>
<td>Move from coprocessor to 2 registers</td>
<td>page 4-106</td>
<td>SE&lt;sup&gt;d&lt;/sup&gt;, 6</td>
</tr>
<tr>
<td>MRS</td>
<td>Move from PSR to register</td>
<td>page 4-113</td>
<td>3</td>
</tr>
<tr>
<td>MSR</td>
<td>Move from register to PSR</td>
<td>page 4-114</td>
<td>3</td>
</tr>
<tr>
<td>MUL</td>
<td>Multiply</td>
<td>page 4-53</td>
<td>2</td>
</tr>
<tr>
<td>MVN</td>
<td>Move not</td>
<td>page 4-41</td>
<td>All</td>
</tr>
<tr>
<td>NOP</td>
<td>pseudo-instruction Generates the preferred no-operation code.</td>
<td>page 4-126</td>
<td>All</td>
</tr>
<tr>
<td>ORR</td>
<td>Logical OR</td>
<td>page 4-39</td>
<td>All</td>
</tr>
<tr>
<td>PKHBT, PKHTB</td>
<td>Pack halfwords</td>
<td>page 4-94</td>
<td>6</td>
</tr>
<tr>
<td>PLD</td>
<td>Cache preload</td>
<td>page 4-22</td>
<td>SE&lt;sup&gt;d&lt;/sup&gt;</td>
</tr>
<tr>
<td>QA00, QA000, QDSUB, QSUB</td>
<td>Saturating arithmetic</td>
<td>page 4-78</td>
<td>SE&lt;sup&gt;e&lt;/sup&gt;</td>
</tr>
<tr>
<td>QA008, QA0016, QA00SUB, QSUB8, QSUB16, QSUBADDX</td>
<td>Byte-wise and halfword-wise parallel signed saturating arithmetic</td>
<td>page 4-83</td>
<td>6</td>
</tr>
<tr>
<td>REV, REV16, REVSH</td>
<td>Reverse byte order</td>
<td>page 4-50</td>
<td>6</td>
</tr>
<tr>
<td>RFE</td>
<td>Return from exception</td>
<td>page 4-26</td>
<td>6</td>
</tr>
<tr>
<td>RSB, RSC, SBC</td>
<td>Reverse sub, Reverse sub with carry, Sub with carry</td>
<td>page 4-36</td>
<td>All</td>
</tr>
<tr>
<td>SA008, SA0016, SA00SUBX</td>
<td>Byte-wise and halfword-wise parallel signed arithmetic</td>
<td>page 4-83</td>
<td>6</td>
</tr>
<tr>
<td>SEL</td>
<td>Select bytes according to CPSR GE flags</td>
<td>page 4-48</td>
<td>6</td>
</tr>
<tr>
<td>SETEND</td>
<td>Set endianness for memory accesses</td>
<td>page 4-117</td>
<td>6</td>
</tr>
<tr>
<td>SHADD8, SHADD16, SHADDUBX, SHSUB8, SHSUB16, SHSUBADDX</td>
<td>Byte-wise and halfword-wise parallel signed halving arithmetic</td>
<td>page 4-83</td>
<td>6</td>
</tr>
<tr>
<td>SMLAD</td>
<td>Dual signed multiply-accumulate (32 &lt;= 32 + 16 x 16 + 16 x 16)</td>
<td>page 4-68</td>
<td>6</td>
</tr>
<tr>
<td>Mnemonic</td>
<td>Brief description</td>
<td>Page</td>
<td>Architecturea</td>
</tr>
<tr>
<td>------------</td>
<td>--------------------------------------------------------</td>
<td>--------</td>
<td>---------------</td>
</tr>
<tr>
<td>SMLAL</td>
<td>Signed multiply-accumulate (64 &lt;= 64 + 32 x 32)</td>
<td>4-55</td>
<td>Mf</td>
</tr>
<tr>
<td>SMLALD</td>
<td>Dual signed multiply-accumulate long</td>
<td>4-72</td>
<td>6</td>
</tr>
<tr>
<td>SMLALxy</td>
<td>Signed multiply-accumulate (64 &lt;= 64 + 16 x 16 + 16 x 16)</td>
<td>4-63</td>
<td>5ExPc</td>
</tr>
<tr>
<td>SMLAy</td>
<td>Signed multiply-accumulate (32 &lt;= 32 + 32 x 16)</td>
<td>4-61</td>
<td>5ExPc</td>
</tr>
<tr>
<td>SMLAxy</td>
<td>Signed multiply-accumulate (32 &lt;= 32 + 16 x 16)</td>
<td>4-58</td>
<td>5ExPc</td>
</tr>
<tr>
<td>SMLSD</td>
<td>Dual signed multiply-subtract-accumulate</td>
<td>4-68</td>
<td>6</td>
</tr>
<tr>
<td>SMLSLD</td>
<td>Dual signed multiply-subtract-accumulate long</td>
<td>4-72</td>
<td>6</td>
</tr>
<tr>
<td>SMULL</td>
<td>Signed multiply (64 &lt;= 32 x 32)</td>
<td>4-55</td>
<td>Mf</td>
</tr>
<tr>
<td>SMULWy</td>
<td>Signed multiply (32 &lt;= 32 x 16)</td>
<td>4-60</td>
<td>5ExPc</td>
</tr>
<tr>
<td>SMULxy</td>
<td>Signed multiply (32 &lt;= 16 x 16)</td>
<td>4-57</td>
<td>5ExPc</td>
</tr>
<tr>
<td>SMMLA</td>
<td>Signed top word multiply-accumulate</td>
<td>4-70</td>
<td>6</td>
</tr>
<tr>
<td>SMMLS</td>
<td>Signed top word multiply-subtract</td>
<td>4-70</td>
<td>6</td>
</tr>
<tr>
<td>SMMUL</td>
<td>Signed top word multiply (32 &lt;= TopWord(32 x 32))</td>
<td>4-67</td>
<td>6</td>
</tr>
<tr>
<td>SMUAD, SMUSD</td>
<td>Dual signed multiply, and add or subtract products</td>
<td>4-65</td>
<td>6</td>
</tr>
<tr>
<td>SRS</td>
<td>Store return state</td>
<td>4-24</td>
<td>6</td>
</tr>
<tr>
<td>SSAT</td>
<td>Signed saturate</td>
<td>4-80</td>
<td>6</td>
</tr>
<tr>
<td>SSAT16</td>
<td>Signed saturate, parallel halfwords</td>
<td>4-88</td>
<td>6</td>
</tr>
<tr>
<td>SSUB8, SSUB16, SSUBADDX</td>
<td>Byte-wise and halfword-wise parallel signed arithmetic</td>
<td>4-83</td>
<td>6</td>
</tr>
<tr>
<td>STC, STC2</td>
<td>Store coprocessor</td>
<td>4-107</td>
<td>2, 5ExPc</td>
</tr>
<tr>
<td>STM</td>
<td>Store multiple registers</td>
<td>4-20</td>
<td>All</td>
</tr>
<tr>
<td>STR</td>
<td>Store register</td>
<td>4-8</td>
<td>All</td>
</tr>
<tr>
<td>STREX</td>
<td>Store register exclusive</td>
<td>4-28</td>
<td>6</td>
</tr>
<tr>
<td>Mnemonic</td>
<td>Brief description</td>
<td>Page</td>
<td>Architecture&lt;sup&gt;a&lt;/sup&gt;</td>
</tr>
<tr>
<td>--------------------------</td>
<td>------------------------------------------------------------------------------------</td>
<td>-----------</td>
<td>---------------------------</td>
</tr>
<tr>
<td>SUB</td>
<td>Subtract</td>
<td>page 4-36</td>
<td>All</td>
</tr>
<tr>
<td>SWI</td>
<td>Software interrupt</td>
<td>page 4-112</td>
<td>All</td>
</tr>
<tr>
<td>SWP</td>
<td>Swap registers and memory</td>
<td>page 4-31</td>
<td>3</td>
</tr>
<tr>
<td>SXT, SXTA</td>
<td>Signed extend, with optional addition</td>
<td>page 4-91</td>
<td>6</td>
</tr>
<tr>
<td>TEQ, TST</td>
<td>Test equivalence, Test</td>
<td>page 4-45</td>
<td>All</td>
</tr>
<tr>
<td>UADD8, UADD16, UADDSUBX</td>
<td>Byte-wise and halfword-wise parallel unsigned arithmetic</td>
<td>page 4-83</td>
<td>6</td>
</tr>
<tr>
<td>UHADD8, UHADD16, UHADDSUBX, UHSUB8, UHSUB16, UHSUBADDX</td>
<td>Byte-wise and halfword-wise parallel unsigned halving arithmetic</td>
<td>page 4-83</td>
<td>6</td>
</tr>
<tr>
<td>UMAAL</td>
<td>Unsigned multiply accumulate accumulate long (64 &lt;= 32 + 32 x 32)</td>
<td>page 4-74</td>
<td>6</td>
</tr>
<tr>
<td>UMLAL, UMULL</td>
<td>Unsigned multiply-acumulate, multiply (64 &lt;= 32 x 32 + 64), (64 &lt;= 32 x 32)</td>
<td>page 4-55</td>
<td>M&lt;sup&gt;f&lt;/sup&gt;</td>
</tr>
<tr>
<td>UQADD8, UQADD16, UQADDSUBX, UQSUB8, UQSUB16, UQSUBADDX</td>
<td>Byte-wise and halfword-wise parallel unsigned saturating arithmetic</td>
<td>page 4-83</td>
<td>6</td>
</tr>
<tr>
<td>USAD8</td>
<td>Unsigned sum of absolute differences</td>
<td>page 4-86</td>
<td>6</td>
</tr>
<tr>
<td>USADA8</td>
<td>Accumulate unsigned sum of absolute differences</td>
<td>page 4-86</td>
<td>6</td>
</tr>
<tr>
<td>USAT</td>
<td>Unsigned saturate</td>
<td>page 4-80</td>
<td>6</td>
</tr>
<tr>
<td>USAT16</td>
<td>Unsigned saturate, parallel halfwords</td>
<td>page 4-88</td>
<td>6</td>
</tr>
<tr>
<td>USUB8, USUB16, USUBADDX</td>
<td>Byte-wise and halfword-wise parallel unsigned arithmetic</td>
<td>page 4-83</td>
<td>6</td>
</tr>
<tr>
<td>UXT, UXTA</td>
<td>Unsigned extend, with optional addition</td>
<td>page 4-91</td>
<td>6</td>
</tr>
</tbody>
</table>

<sup>a</sup> n : available in architecture version n and above  
<sup>b</sup> nT : available in T variants of architecture version n and above  
<sup>c</sup> XScale: XScale coprocessor instructions  
<sup>d</sup> 5E : available in ARMv5E, except ExP variants, and ARMv6 and above  
<sup>e</sup> 5ExP : available in ARMv5E, including ExP variants, and ARMv6 and above  
<sup>f</sup> M : available in ARMv3M, and ARMv4 and above, except xM versions
4.1 Conditional execution

Almost all ARM instructions can include an optional condition code. This is shown in syntax descriptions as \{cond\}. 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 4-2.

### Table 4-2 ARM condition codes

<table>
<thead>
<tr>
<th>Suffix</th>
<th>Flags</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>EQ</td>
<td>Z set</td>
<td>Equal</td>
</tr>
<tr>
<td>NE</td>
<td>Z clear</td>
<td>Not equal</td>
</tr>
<tr>
<td>CS/HS</td>
<td>C set</td>
<td>Higher or same (unsigned (\geq))</td>
</tr>
<tr>
<td>CC/LO</td>
<td>C clear</td>
<td>Lower (unsigned (&lt;))</td>
</tr>
<tr>
<td>MI</td>
<td>N set</td>
<td>Negative</td>
</tr>
<tr>
<td>PL</td>
<td>N clear</td>
<td>Positive or zero</td>
</tr>
<tr>
<td>VS</td>
<td>V set</td>
<td>Overflow</td>
</tr>
<tr>
<td>VC</td>
<td>V clear</td>
<td>No overflow</td>
</tr>
<tr>
<td>HI</td>
<td>C set and Z clear</td>
<td>Higher (unsigned (\leq))</td>
</tr>
<tr>
<td>LS</td>
<td>C clear or Z set</td>
<td>Lower or same (unsigned (\leq))</td>
</tr>
<tr>
<td>GE</td>
<td>N and V the same</td>
<td>Signed (\geq)</td>
</tr>
<tr>
<td>LT</td>
<td>N and V different</td>
<td>Signed (&lt;)</td>
</tr>
<tr>
<td>GT</td>
<td>Z clear, and N and V the same</td>
<td>Signed (&gt;)</td>
</tr>
<tr>
<td>LE</td>
<td>Z set, or N and V different</td>
<td>Signed (\leq)</td>
</tr>
<tr>
<td>AL</td>
<td>Any</td>
<td>Always (usually omitted)</td>
</tr>
</tbody>
</table>

Almost all ARM data processing instructions can optionally update the condition code flags according to the result. To make an instruction update the flags, include the S suffix as shown in the syntax description for the instruction.

Some instructions (CMP, CMN, TST and TEQ) do not require the S suffix. Their only function is to update the flags. They always update the flags.

Flags are preserved until updated. A conditional instruction which 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.

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.

For more information, see *Conditional execution* on page 2-22.

### 4.1.1 The Q flag

The Q flag only exists in ARM architecture v5TE, and ARMv6 and above. It is used to detect saturation in special saturating arithmetic instructions (see *QADD, QSUB, QDADD, and QDSUB* on page 4-78), or overflow in certain multiply instructions (see *SMLA* on page 4-58 and *SMLAW* on page 4-61).

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 4-114).

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 4-113).
4.2 ARM Memory access instructions

This section contains the following subsections:

- **LDR and STR, words and unsigned bytes** on page 4-9
  Load register and store register, 32-bit word or 8-bit unsigned byte.

- **LDR and STR, halfwords and signed bytes** on page 4-14
  Load register, signed 8-bit bytes and signed and unsigned 16-bit halfwords.
  Store register, 16-bit halfwords.

- **LDR and STR, doublewords** on page 4-17
  Load two consecutive registers and store two consecutive registers.

- **LDM and STM** on page 4-20
  Load and store multiple registers.

- **PLD** on page 4-22
  Cache preload.

- **SRS** on page 4-24
  Store return state.

- **RFE** on page 4-26
  Return from exception.

- **LDREX and STREX** on page 4-28
  Load and store register exclusive.

- **SWP and SWPB** on page 4-31
  Swap data between registers and memory.

--- Note ---

There is also an LDR pseudo-instruction (see *LDR ARM pseudo-instruction* on page 4-124). This pseudo-instruction either assembles to an LDR instruction, or to a MOV or MVN instruction.
4.2.1 LDR and STR, words and unsigned bytes

Load register and store register, 32-bit word or 8-bit unsigned byte. Byte loads are zero-extended to 32 bits.

--- Note ---

Also, see ARM pseudo-instructions on page 4-120.

Syntax

Both LDR and STR have four possible forms:

- zero offset
- pre-indexed offset
- program-relative
- post-indexed offset.

The syntax of the four forms, in the same order, are:

\[
\begin{align*}
& \text{op} \{ \text{cond} \} \{ \text{B} \} \{ \text{T} \} \ Rd, [\ Rn \] \\
& \text{op} \{ \text{cond} \} \{ \text{B} \} \ Rd, [\ Rn, \text{FlexOffset}]\{!\} \\
& \text{op} \{ \text{cond} \} \{ \text{B} \} \ Rd, \text{label} \\
& \text{op} \{ \text{cond} \} \{ \text{T} \} \ Rd, [\ Rn], \text{FlexOffset}
\end{align*}
\]

where:

- \( \text{op} \) is either LDR (Load Register) or STR (Store Register).
- \( \text{cond} \) is an optional condition code (see Conditional execution on page 4-6).
- \( \text{B} \) is an optional suffix. If \( \text{B} \) is present, the least significant byte of \( \text{Rd} \) is transferred. If \( \text{op} \) is LDR, the other bytes of \( \text{Rd} \) are cleared. Otherwise, a 32-bit word is transferred.
- \( \text{T} \) is an optional suffix. If \( \text{T} \) 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). \( \text{T} \) has no effect in User mode. You cannot use \( \text{T} \) with a pre-indexed offset.
- \( \text{Rd} \) is the ARM register to load or save.
- \( \text{Rn} \) is the register on which the memory address is based.
- \( \text{Rn} \) must not be the same as \( \text{Rd} \), if the instruction:
  - is pre-indexed with writeback (the \! suffix)
FlexOffset is a flexible offset applied to the value in \( R_n \) (see Flexible offset syntax on page 4-11).

\( label \) is a program-relative expression. See Register-relative and program-relative expressions on page 3-29 for more information. \( label \) must be within \( \pm 4KB \) of the current instruction.

\( ! \) is an optional suffix. If \( ! \) is present, the address including the offset is written back into \( R_n \). You cannot use the \( ! \) suffix if \( R_n \) is \( r15 \).

**Zero offset**

The value in \( R_n \) is used as the address for the transfer.

**Pre-indexed offset**

The offset is applied to the value in \( R_n \) before the data transfer takes place. The result is used as the memory address for the transfer. If the \( ! \) suffix is used, the result is written back into \( R_n \). \( R_n \) must not be \( r15 \) if the \( ! \) suffix is used.

**Program-relative**

This is an alternative version of the pre-indexed form. The assembler calculates the offset from the PC for you, and generates a pre-indexed instruction with the PC as \( R_n \).

You cannot use the \( ! \) suffix.

**Post-indexed offset**

The value in \( R_n \) is used as the memory address for the transfer. The offset is applied to the value in \( R_n \) after the data transfer takes place. The result is written back into \( R_n \). \( R_n \) must not be \( r15 \).
Flexible offset syntax

Both pre-indexed and post-indexed offsets can be either of the following:

\[ \text{#expr} \]

\[ \{-\}Rm\{, \text{shift}\} \]

where:

- is an optional minus sign. If - is present, the offset is subtracted from \( Rn \). Otherwise, the offset is added to \( Rn \).

\( \text{expr} \) is an expression evaluating to an integer in the range -4095 to +4095. This is often a numeric constant (see examples below).

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

\( \text{shift} \) is an optional shift to be applied to \( Rm \). It can be any one of:

- ASR \( \#n \) arithmetic shift right \( n \) bits. \( 1 \leq n \leq 32 \).
- LSL \( \#n \) logical shift left \( n \) bits. \( 0 \leq n \leq 31 \).
- LSR \( \#n \) logical shift right \( n \) bits. \( 1 \leq n \leq 32 \).
- ROR \( \#n \) rotate right \( n \) bits. \( 1 \leq n \leq 31 \).
- RRX rotate right one bit, with extend.
Address alignment for word transfers

In most circumstances, you must ensure that addresses for 32-bit transfers are 32-bit word-aligned.

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.

Loading to r15

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

Bits[1:0] of the value loaded:
- are ignored in ARMv3 and below
- must be zero in ARMv4.

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 B or T suffixes when loading to r15.
Saving from r15

In general, 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 instructions are available in all versions of the ARM architecture.

In T variants of ARMv5 and above, a load to r15 causes a change to executing Thumb instructions if bit[0] of the value loaded is set.

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.
STRB    r0,[r3,-r8,ASR #2]  ; stores the least significant byte from
                        ; r0 to a byte at an address equal to
                        ; contents(r3) minus contents(r8)/4.
                        ; r3 and r8 are not altered.
STR     r5,[r7],#-8         ; stores a word from r5 to the address
                        ; in r7, and then decrements r7 by 8.
LDR     r0,localdata        ; loads a word located at label localdata
```
4.2.2 LDR and STR, halfwords and signed bytes

Load register, signed 8-bit bytes and signed and unsigned 16-bit halfwords.

Store register, 16-bit halfwords.

Signed loads are sign-extended to 32 bits. Unsigned halfword loads are zero-extended to 32 bits.

Syntax

These instructions have four possible forms:

- zero offset
- pre-indexed offset
- program-relative
- post-indexed offset.

The syntax of the four forms, in the same order, are:

- \( \text{op}\{\text{cond}\}\text{type } \text{Rd}, [\text{Rn}] \)
- \( \text{op}\{\text{cond}\}\text{type } \text{Rd}, [\text{Rn}, \text{Offset}]\}
- \( \text{op}\{\text{cond}\}\text{type } \text{Rd}, \text{label}\)
- \( \text{op}\{\text{cond}\}\text{type } \text{Rd}, [\text{Rn}], \text{Offset} \)

where:

- \( \text{op} \) is either LDR or STR.
- \( \text{cond} \) is an optional condition code (see Conditional execution on page 4-6).
- \( \text{type} \) must be one of:
  - \( \text{SH} \) for Signed Halfword (LDR only)
  - \( \text{H} \) for unsigned Halfword
  - \( \text{SB} \) for Signed Byte (LDR only).
- \( \text{Rd} \) is the ARM register to load or save.
- \( \text{Rn} \) is the register on which the memory address is based.

\( \text{Rn} \) must not be the same as \( \text{Rd} \), if the instruction is either:

- pre-indexed with writeback
- post-indexed.
label is a program-relative expression. See Register-relative and program-relative expressions on page 3-29 for more information. label must be within ±255 bytes of the current instruction.

Offset is an offset applied to the value in \( R_n \) (see Offset syntax).

! is an optional suffix. If ! is present, the address including the offset is written back into \( R_n \). You cannot use the ! suffix if \( R_n \) is r15.

**Zero offset**

The value in \( R_n \) is used as the address for the transfer.

**Pre-indexed offset**

The offset is applied to the value in \( R_n \) before the transfer takes place. The result is used as the memory address for the transfer. If the ! suffix is used, the result is written back into \( R_n \).

**Program-relative**

This is an alternative version of the pre-indexed form. The assembler calculates the offset from the PC for you, and generates a pre-indexed instruction with the PC as \( R_n \).

You cannot use the ! suffix.

**Post-indexed offset**

The value in \( R_n \) is used as the memory address for the transfer. The offset is applied to the value in \( R_n \) after the transfer takes place. The result is written back into \( R_n \).

**Offset syntax**

Both pre-indexed and post-indexed offsets can be either of the following:

\[ \#expr \]

\[ \{-\}Rm \]

where:

- is an optional minus sign. If - is present, the offset is subtracted from \( R_n \). Otherwise, the offset is added to \( R_n \).

expr is an expression evaluating to an integer in the range -255 to +255. This is often a numeric constant (see examples below).
$Rm$ is a register containing a value to be used as the offset. $Rm$ must not be r15.

The offset syntax is the same for LDR and STR, doublewords on page 4-17.

**Address alignment for halfword transfers**

The address must be even for halfword transfers.

If your system has a system coprocessor (cp15), you can enable alignment checking. Non halfword-aligned 16-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:

- a non halfword-aligned 16-bit load corrupts $Rd$
- a non halfword-aligned 16-bit save corrupts two bytes at [address] and [address–1].

**Loading to r15**

You cannot load halfwords or bytes to r15.

**Architectures**

These instructions are available in ARMv4 and above.

**Examples**

- LDRQSH r11,[r6] ; (conditionally) loads r11 with a 16-bit halfword from the address in r6. Sign extends to 32 bits.
- LDRH  r1,[r0,#22] ; load r1 with a 16 bit halfword from 22 bytes above the address in r0. Zero extend to 32 bits.
- STRH  r4,[r0,r1]! ; store the least significant halfword from r4 to two bytes at an address equal to contents(r0) plus contents(r1). Write address back into r0.
- LDRSB r6,constf ; load a byte located at label constf. Sign extend.

**Incorrect example**

- LDRSB r1,[r6],r3,LSL#4 ; This format is only available for word and unsigned byte transfers.
4.2.3 LDR and STR, doublewords

Load two consecutive registers and store two consecutive registers, 64-bit doubleword.

Syntax

These instructions have four possible forms:

- zero offset
- pre-indexed offset
- program-relative
- post-indexed offset.

The syntax of the four forms are, in the same order:

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

\[
\text{op}\{\text{cond}\}\text{D}\ Rd, \ [Rn, \ Offset]}
\]

\[
\text{op}\{\text{cond}\}\text{D}\ Rd, \ label
\]

\[
\text{op}\{\text{cond}\}\text{D}\ Rd, \ [Rn], \ Offset
\]

where:

- \(\text{op}\) is either LDR or STR.
- \(\text{cond}\) is an optional condition code (see Conditional execution on page 4-6).
- \(Rd\) is one of the ARM registers to load or save. The other one is \(R(d+1)\). \(Rd\) must be an even numbered register, and not \(r14\).
- \(Rn\) is the register on which the memory address is based.
- \(Rn\) must not be the same as \(Rd\) or \(R(d+1)\), unless the instruction is either:
  - zero offset
  - pre-indexed without writeback.
- \(Offset\) is an offset applied to the value in \(Rn\) (see Offset syntax on page 4-18).
- \(label\) is a program-relative expression. See Register-relative and program-relative expressions on page 3-29 for more information.
- \(label\) must be within \(±252\) bytes of the current instruction.
- \(!\) is an optional suffix. If \(!\) is present, the final address including the offset is written back into \(Rn\).
Zero offset

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

Pre-indexed offset

The offset is applied to the value in $Rn$ before the transfers take place. The result is used as the memory address for the transfers. If the ! suffix is used, the address is written back into $Rn$.

Program-relative

This is an alternative version of the pre-indexed form. The assembler calculates the offset from the PC for you, and generates a pre-indexed instruction with the PC as $Rn$.

You cannot use the ! suffix.

Post-indexed offset

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

Offset syntax

Both pre-indexed and post-indexed offsets can be either of the following:

$\#expr$

{\{-}Rm

where:

- is an optional minus sign. If - is present, the offset is subtracted from $Rn$. Otherwise, the offset is added to $Rn$.

$expr$ is an expression evaluating to an integer in the range –255 to +255. This is often a numeric constant (see examples below).

$Rm$ is a register containing a value to be used as the offset. For loads, $Rm$ must not be the same as $Rd$ or $R(d+1)$, and must not be r15.

This is the same offset syntax as for LDR and STR, halfwords and signed bytes on page 4-14.
Address alignment

Depending on your architecture and system configuration, an address for doubleword transfers might be:

- required to be aligned on an 8-byte boundary
- required to be aligned on a 4-byte boundary.

If your system has a system coprocessor, you can enable alignment checking. Incorrectly aligned doubleword transfers cause an alignment exception if alignment checking is enabled.

Architectures

These instructions are available in ARMv6 and above, and E variants of ARMv5.

Examples

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDRD</td>
<td>r6,[r11]</td>
</tr>
<tr>
<td>LDRMID</td>
<td>r4,[r7],r2</td>
</tr>
<tr>
<td>STRD</td>
<td>r4,[r9,#24]</td>
</tr>
<tr>
<td>STRD</td>
<td>r0,[r9,-r2]!</td>
</tr>
<tr>
<td>LDREQD</td>
<td>r8,abc4</td>
</tr>
</tbody>
</table>

Incorrect examples

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDRD</td>
<td>r1,[r6]                  ; Rd must be even.</td>
</tr>
<tr>
<td>STRD</td>
<td>r14,[r9,#36]             ; Rd must not be r14.</td>
</tr>
<tr>
<td>STRD</td>
<td>r2,[r3],r6              ; Rn must not be Rd or R(d+1).</td>
</tr>
</tbody>
</table>
4.2.4 LDM and STM

Load and store multiple registers. Any combination of registers r0 to r15 can be transferred.

Syntax

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

where:

- \(\text{op}\) is either LDM or STM.
- \(\text{cond}\) is an optional condition code (see Conditional execution on page 4-6).
- \(\text{addr\_mode}\) is any one of the following:
  - IA: increment address after each transfer
  - IB: increment address before each transfer
  - DA: decrement address after each transfer
  - DB: decrement address before each transfer
  - FD: full descending stack
  - ED: empty descending stack
  - FA: full ascending stack
  - EA: empty ascending stack.
- \(Rn\) is the base register, the ARM register containing 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 \(Rn\).
- \(\text{reglist}\) is a list of 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 4-21).
- \(^\wedge\) is an optional suffix. You must not use it in User mode or System mode. It has the following purposes:
  - If \(\text{op}\) is LDM and \(\text{reglist}\) contains the pc (r15), in addition to the normal multiple register transfer, the SPSR is copied into the CPSR. This is for returning from exception handlers. Use this only from exception modes.
  - Otherwise, data is transferred into or out of the User mode registers instead of the current mode registers.
Non word-aligned addresses

These instructions ignore bits [1:0] of the address. (On a system with a system coprocessor, if alignment checking is enabled, nonzero values in these bits cause an alignment exception.)

Loading to r15

A load to r15 (the program counter) causes a branch to the instruction at the address loaded. In T variants of ARMv5 and above, a load to r15 causes a change to executing Thumb instructions if bit 0 of the value loaded is set.

Loading or storing the base register, with writeback

If \( Rn \) is in \( reglist \), and writeback is specified with the ! suffix:
- if \( op \) is STM and \( Rn \) is the lowest-numbered register in \( reglist \), the initial value of \( Rn \) is stored
- otherwise, the loaded or stored value of \( Rn \) is UNPREDICTABLE.

Architectures

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

In T variants of ARMv5 and above, a load to r15 causes a change to executing Thumb instructions if bit 0 of the value loaded is set.

Examples

- LDMIA \( r8,\{r0,r2,r9\} \)
- STMDB \( r1!,\{r3-r6,r11,r12\} \)
- STMFD \( r13!,\{r0,r4-r7,LR\} \)  ; Push registers including the stack pointer
- LDMFD \( r13!,\{r0,r4-r7,PC\} \)  ; Pop the same registers and return from subroutine

Incorrect examples

- STMIA \( r5!,\{r5,r4,r9\} \)  ; value stored for r5 unpredictable
- LDMDA \( r2,\{\} \)  ; must be at least one register in list
4.2.5 PLD

Cache preload.

Syntax

\texttt{PLD [Rn{, FlexOffset}]}

where:

- \texttt{Rn} is the register on which the memory address is based.
- \texttt{FlexOffset} is an optional flexible offset applied to the value in \texttt{Rn}.

\texttt{FlexOffset} can be either of the following:

\begin{itemize}
  \item \texttt{#expr}
  \item \texttt{-}\texttt{Rm{, shift}}
\end{itemize}

where:

- \texttt{-} is an optional minus sign. If \texttt{-} is present, the offset is subtracted from \texttt{Rn}. Otherwise, the offset is added to \texttt{Rn}.
- \texttt{expr} is an expression evaluating to an integer in the range \(-4095\) to \(+4095\). This is often a numeric constant.
- \texttt{Rm} is a register containing a value to be used as the offset.
- \texttt{shift} is an optional shift to be applied to \texttt{Rm}. It can be any one of:
  \begin{itemize}
    \item \texttt{ASR \#n} arithmetic shift right \(n\) bits. \(1 \leq n \leq 32\).
    \item \texttt{LSL \#n} logical shift left \(n\) bits. \(0 \leq n \leq 31\).
    \item \texttt{LSR \#n} logical shift right \(n\) bits. \(1 \leq n \leq 32\).
    \item \texttt{ROR \#n} rotate right \(n\) bits. \(1 \leq n \leq 31\).
    \item \texttt{RRX} rotate right one bit, with extend.
  \end{itemize}

This is the same offset syntax as for \texttt{LDR} and \texttt{STR}, \textit{words and unsigned bytes} on page 4-9.

Usage

Use \texttt{PLD} to hint to the memory system that there is likely to be a load from the specified address within the next few instructions. If possible, the memory system uses this to speed up later memory accesses, otherwise \texttt{PLD} has no effect.
**Alignment**

There are no alignment restrictions on the address. If a system control coprocessor (cp15) is present then it will not generate an alignment exception for any PLD instruction.

**Architectures**

PLD is available in ARMv6 and above, and E variants of ARMv5.

**Examples**

```
PLD [r2]
PLD [r15,#280]
PLD [r9,#-2481]
PLD [r0,#av*4] ; av * 4 must evaluate, at assembly time, to
               ; an integer in the range -4095 to +4095
PLD [r0,r2]
PLD [r5,r8,LSL #2]
```
4.2.6 SRS

Store return state.

Syntax

SRS addr_mode #mode{!}

where:

- **addr_mode**
  - IA: increment address after each transfer
  - IB: increment address before each transfer
  - DA: decrement address after each transfer
  - DB: decrement address before each transfer
  - FD: full descending stack
  - ED: empty descending stack
  - FA: full ascending stack
  - EA: empty ascending stack.

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

- !
  - An optional suffix. If ! is present, the final address is written back into the banked r13 used as the base register.

Operation

SRS stores the r14 and the SPSR of the current mode, at the address contained in r13 of the mode specified by mode, and the following address. Optionally updates r13 of the mode specified by mode. This is compatible with the normal use of the STM instruction for stack accesses, see *LDM and STM* on page 4-20.

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

Architectures

SRS is available in ARMv6 and above.

Example

```
R13_usr EQU 16

SRSFD #R13_usr
```
Incorrect examples

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

Return from exception.

Syntax

RFE addr_mode  Rn{!}

where:

addr_mode is any one of the following:
IA    increment address after each transfer
IB    increment address before each transfer
DA    decrement address after each transfer
dB    decrement address before each transfer
FD    full descending stack
ED    empty descending stack
FA    full ascending stack
EA    empty ascending stack.

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

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

Operation

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

You can use RFE to return from an exception if you previously saved the return state using the SRS instruction (see SRS on page 4-24).

Architectures

RFE is available in ARMv6 and above.

Example

RFEFD  r13!
Incorrect examples

RFENEFD r13! ; RFE is always unconditional
RFEFD r15   ; do not use r15
4.2.8 LDREX and STREX

Load register exclusive and store register exclusive.

Syntax

LDREX{cond} Rd, [Rn]
STREX{cond} Rd, Rm, [Rn]

where:

cond is an optional condition code (see Conditional execution on page 4-6).

Rd is the destination register. After the instruction, this will contain:
- for LDREX, the word loaded from memory
- for STREX, either:
  - 0 if the instruction succeeds
  - 1 if the instruction is locked out.
Rd must not be r15.

Rm is the source register containing the word to store to memory. Rm must be distinct from Rd, and must not be r15.

Rn is the register containing the memory address. Rn must not be r15, and for STREX it must be distinct from Rd.

LDREX

LDREX loads a word from memory.

- If the physical address has the Shared TLB attribute, LDREX tags the physical address as exclusive access for the current processor, and clears any exclusive access tag for this processor for any other physical address.
- Otherwise, it tags the fact that the executing processor has an outstanding tagged physical address.
STREX

STREX performs a conditional store to memory. The conditions are as follows:

- If the physical address does not have the Shared TLB attribute, and the executing processor has an outstanding tagged physical address, the store takes place and the tag is cleared.

- If the physical address does not have the Shared TLB attribute, and the executing processor does not have an outstanding tagged physical address, the store does not take place.

- If the physical address has the Shared TLB attribute, and the physical address is tagged as exclusive access for the executing processor, the store takes place and the tag is cleared.

- If the physical address has the Shared TLB attribute, and the physical address is not tagged as exclusive access for the executing processor, the store does not take place.

Usage

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

For reasons of performance, keep the number of instructions between corresponding LDREX and STREX instruction to a minimum.

Note

The address used in an STREX instruction must be the same as the address in the most recently executed LDREX instruction. The result of executing an STREX instruction to a different address is UNPREDICTABLE.

Architectures

These instructions are available in ARMv6 and above.
Example

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

Incorrect examples

LDREX   r0, [r2,#96]!   ; no offset allowed
STREX   r11, r10, [r15] ; use of r15 not allowed
LDREXGT r6, r4, [r10]   ; LDREX has only two registers in its syntax
STREX r3, r3,  [r7]    ; Rd must be distinct from Rn (and from Rm)
4.2.9 SWP and SWPB

Swap data between registers and memory.

You can use SWP to implement semaphores. See also LDREX and STREX on page 4-28 for instructions to implement more sophisticated semaphores in ARMv6 and above.

**Syntax**

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

where:

cond is an optional condition code (see Conditional execution on page 4-6).

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.

**Non word-aligned addresses**

Non word-aligned addresses are handled in exactly the same way as an LDR and an STR instruction (see Address alignment for word transfers on page 4-12).

**Architectures**

SWP is available in ARMv2a, ARMv3, and above.
4.3 ARM general data processing instructions

This section contains the following subsections:

- Flexible second operand on page 4-33
- ADD, SUB, RSB, ADC, SBC, and RSC on page 4-36
  Add, subtract, and reverse subtract, each with or without carry.
- AND, ORR, EOR, and BIC on page 4-39
  Logical AND, OR, Exclusive OR and Bit Clear.
- MOV, CPY and MVN on page 4-41
  Move and Move Not.
- CMP and CMN on page 4-43
  Compare and Compare Negative.
- TST and TEQ on page 4-45
  Test and Test Equivalence.
- CLZ on page 4-47
  Count Leading Zeroes.
- SEL on page 4-48
  Select bytes from each operand according to the state of the CPSR GE flags.
- REV, REV16, and REVSH on page 4-50
  Reverse byte order in a word or halfword. Reverse bytes in a halfword and sign extend.
4.3.1 Flexible second operand

Most ARM general data processing instructions have a flexible second operand. This is shown as \textit{Operand2} in the descriptions of the syntax of each instruction.

Syntax

\textit{Operand2} has the following possible forms:

- \#immed\_8\textit{r}
- \textit{Rm[}, shift\textit{]}

where:

- \textit{immed\_8\textit{r}} 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 (but see \textit{Instruction substitution} on page 4-35).
- \textit{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.
- \textit{shift} is an optional shift to be applied to \textit{Rm}. It can be any one of:
  - \textit{ASR #n} arithmetic shift right \(n\) bits. \(1 \leq n \leq 32\).
  - \textit{LSL #n} logical shift left \(n\) bits. \(0 \leq n \leq 31\).
  - \textit{LSR #n} logical shift right \(n\) bits. \(1 \leq n \leq 32\).
  - \textit{ROR #n} rotate right \(n\) bits. \(1 \leq n \leq 31\).
  - \textit{RRX} rotate right one bit, with extend.
  - \textit{type Rs} where:
    - \textit{type} is one of ASR, LSL, LSR, ROR.
    - \textit{Rs} is an ARM register supplying the shift amount. Only the least significant byte is used.

\textit{Note}

The result of the shift operation is used as \textit{Operand2} in the instruction, but \textit{Rm} itself is not altered.

\textbf{ASR}

Arithmetic shift right by \(n\) bits divides the value contained in \textit{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).

![Figure 4-1 ROR](image)

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 on page 4-35).

![Figure 4-2 RRX](image)
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, 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 immed_8r.

If a value of immed_8r cannot be expressed as a rotated 8-bit pattern, but its logical inverse or negation could be, the assembler substitutes the other instruction of the pair and inverts or negates immed_8r.

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

Examples

ADD r3,r7,#1020 ; immed_8r. 1020 is 0xFF rotated right by 30 bits.
AND r0,r5,r2 ; r2 contains the data for Operand2.
SUB r11,r12,r3,ASR #5 ; Operand2 is the contents of r3 divided by 32.
MOVS r4,r4,LSR #32 ; Updates the C flag to r4 bit 31. Clears r4 to 0.

Incorrect examples

ADD r3,r7,#1023 ; 1023 (0x3FF) is not a rotated 8-bit pattern.
SUB r11,r12,r3,LSL #32 ; #32 is out of range for LSL.
MOVS r4,r4,RRX #3 ; Do not specify a shift amount for RRX. RRX is always a one-bit shift.
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 4-83.

**Syntax**

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

where:

- \( \text{op} \) is one of ADD, SUB, RSB, ADC, SBC, or RSC.
- \( \text{cond} \) is an optional condition code (see Conditional execution on page 4-6).
- \( S \) 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 4-6).
- \( Rd \) is the destination register.
- \( Rn \) is the register holding the first operand.
- \( \text{Operand2} \) is a flexible second operand. See Flexible second operand on page 4-33 for details of the options.

**Usage**

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

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

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

You can use ADC, SBC, and RSC to synthesize multiword arithmetic (see Multiword arithmetic examples on page 4-38).

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

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

The RSC (Reverse Subtract with Carry) instruction subtracts the value in \( Rn \) from the value of \( \text{Operand2} \). If the carry flag is clear, the result is reduced by one.
In certain circumstances, the assembler can substitute one instruction for another. Be aware of this when reading disassembly listings. See Instruction substitution on page 4-35 for details.

**Condition flags**

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

**Use of r15**

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

If you use r15 as Rd:

- Execution branches to the address corresponding to the result.
- If you use the S suffix, the SPSR of the current mode is copied to the CPSR. You can use this to return from exceptions (see the Handling Processor Exceptions chapter in RealView Compilation Tools v2.1 Developer Guide).

--- Caution ---

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

---

You cannot use r15 for Rd or any operand in any data processing instruction that has a register-controlled shift (see Flexible second operand on page 4-33).

**Architectures**

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

**Examples**

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

**Incorrect example**

- `RSCLES r0, r15, r0, LSL r4` ; r15 not allowed 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.

```
ADD   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:

```
SUB   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:

```
SUB   r6,r6,r9
SBCS  r9,r2,r1
SBC   r2,r8,r11
```
4.3.3 AND, ORR, EOR, and BIC

Logical AND, OR, Exclusive OR and Bit Clear.

Syntax

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

where:

- \( op \) is one of AND, ORR, EOR, or BIC.
- \( cond \) is an optional condition code (see Conditional execution on page 4-6).
- \( S \) 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 4-6).
- \( Rd \) is the destination register.
- \( Rn \) is the register holding the first operand.
- \( Operand2 \) is a flexible second operand. See Flexible second operand on page 4-33 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 4-35 for details.

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 4-33)
- do not affect the V flag.
Use of r15

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

If you use r15 as \( Rd \):
- Execution branches to the address corresponding to the result.
- If you use the S suffix, the SPSR of the current mode is copied to the CPSR. You can use this to return from exceptions (see the Handling Processor Exceptions chapter in RealView Compilation Tools v2.1 Developer Guide).

Caution

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

You cannot use r15 for \( Rd \) or any operand in any data processing instruction that has a register-controlled shift (see Flexible second operand on page 4-33).

Architectures

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

Examples

\[
\begin{align*}
&\text{AND} \quad r9, r2, \#0xFF00 \\
&\text{ORREQ} \quad r2, r0, r5 \\
&\text{EORS} \quad r0, r0, r3, \text{ROR} r6 \\
&\text{BICNE} \quad r8, r10, r0, \text{RRX}
\end{align*}
\]

Incorrect example

\[
\begin{align*}
&\text{EORS} \quad r0, r15, r3, \text{ROR} r6 \quad ; \text{r15 not allowed with register controlled shift}
\end{align*}
\]
4.3.4 MOV, CPY and MVN

Move, Copy and Move Not.

Syntax

\[
\begin{align*}
\text{MOV}\{\text{cond}\}\{S\} & : \text{Rd}, \text{Operand2} \\
\text{CPY}\{\text{cond}\} & : \text{Rd}, \text{Rm} \\
\text{MVN}\{\text{cond}\}\{S\} & : \text{Rd}, \text{Operand2}
\end{align*}
\]

where:

- \text{cond} is an optional condition code (see Conditional execution on page 4-6).
- \text{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 4-6).
- \text{Rd} is the destination register.
- \text{Operand2} is a flexible second operand. See Flexible second operand on page 4-33 for details of the options.
- \text{Rm} is the source register.

Usage

The MOV instruction copies the value of \text{Operand2} into \text{Rd}.

The MVN instruction takes the value of \text{Operand2}, performs a bitwise logical NOT operation on the value, and places the result into \text{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 4-35 for details.

The CPY instruction is a synonym for MOV without the S suffix, and with \text{Operand2} an unshifted register. It copies the unshifted value of \text{Rm} into \text{Rd}.
**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 $\text{Operand2}$ (see *Flexible second operand* on page 4-33)
- do not affect the V flag.

You cannot use the $S$ suffix with CPY, and the flags are not updated.

**Use of r15**

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

If you use r15 as $Rd$:

- Execution branches to the address corresponding to the result.
- If you use the $S$ suffix, the SPSR of the current mode is copied to the CPSR. You can use this to return from exceptions (see the *Handling Processor Exceptions* chapter in *RealView Compilation Tools v2.1 Developer Guide*).

**Caution**

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

You cannot use r15 for $Rd$ or any operand in any data processing instruction that has a register-controlled shift (see *Flexible second operand* on page 4-33).

**Architectures**

MOV and MVN are available in all versions of the ARM architecture.

CPY is available in ARMv6 and above.

**Example**

MOVNE r11,#0xF000000B

Incorrect examples

MOV r15,r3,ASR r0  ; r15 not allowed with register controlled shift
CPYS r3,r8         ; S suffix not allowed with CPY
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 4-6).
- `Rn` is the ARM register holding the first operand.
- `Operand2` is a flexible second operand. See Flexible second operand on page 4-33 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 4-35 for details.

Condition flags

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

Use of r15

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 4-33).
Architectures

These instructions are available in all versions 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 allowed with register
; controlled shift
4.3.6 TST and TEQ

Test and Test Equivalence.

Syntax

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

where:

- \text{cond} is an optional condition code (see Conditional execution on page 4-6).
- \text{Rn} is the ARM register holding the first operand.
- \text{Operand2} is a flexible second operand. See Flexible second operand on page 4-33 for details of the options.

Usage

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

The TST instruction performs a bitwise AND operation on the value in \text{Rn} and the value of \text{Operand2}. This is the same as a ANDS instruction, except that the result is discarded.

The TEQ instruction performs a bitwise Exclusive OR operation on the value in \text{Rn} and the value of \text{Operand2}. This is the same as a EORS instruction, except that the result is discarded.

Condition flags

These instructions:

- update the N and Z flags according to the result
- can update the C flag during the calculation of \text{Operand2} (see Flexible second operand on page 4-33)
- do not affect the V flag.

Use of r15

If you use r15 as \text{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 4-33).
Architectures

These instructions are available in all versions 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 allowed with register
      ; controlled shift
4.3.7 CLZ

Count Leading Zeroes.

Syntax

\texttt{CLZ}\{cond\} \texttt{Rd, Rm}

where:

\texttt{cond} is an optional condition code (see \textit{Conditional execution} on page 4-6).
\texttt{Rd} is the destination register. \texttt{Rd} must not be r15.
\texttt{Rm} is the operand register. \texttt{Rm} must not be r15.

Usage

The CLZ instruction counts the number of leading zeroes in the value in \texttt{Rm} and returns the result in \texttt{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 affect the flags.

Architectures

CLZ is available in ARMv5 and above.

Examples

\begin{verbatim}
CLZ r4,r9
CLZNE r2,r3
\end{verbatim}
4.3.8  SEL

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

Syntax

SEL{cond}  Rd,  Rn,  Rm

where:

cond        is an optional condition code (see Conditional execution on page 4-6).
Rd           is the destination register.
Rn           is the register containing the first operand.
Rm           is the register containing the second operand.

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

Usage

Use the SEL instruction after one of the signed parallel instructions, see Parallel add and subtract on page 4-83. You can use this to select maximum or minimum values in multiple byte or halfword data.

Operation

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

- if GE[0] is set, Rd[7:0] come from Rn[7:0], otherwise from Rm[7:0]
- if GE[1] is set, Rd[15:8] come from Rn[15:8], otherwise from Rm[15:8]
- if GE[3] is set, Rd[31:24] come from Rn[31:24], otherwise from Rm[31:24]

Condition flags

This instruction does not affect the flags.

Architectures

SEL is available in ARMv6 and above.
Examples

SEL r0, r4, r5
SELLT r4, r0, r4
4.3.9 REV, REV16, and REVSH

Reverse byte order in a word or halfword. Reverse bytes in a halfword and sign extend.

Syntax

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

where:

- \( \text{op} \) is any one of the following:
  - REV: Reverses byte order in a word.
  - REV16: Reverses byte order in each halfword of \( \text{Rm} \).
  - REVSH: Reverses byte order in the bottom halfword of \( \text{Rm} \), and sign extends to 32 bits.
- \( \text{cond} \) is an optional condition code (see Conditional execution on page 4-6).
- \( \text{Rd} \) is the destination register.
- \( \text{Rm} \) is the register containing the second operand.

Do not use \( r15 \) for \( \text{Rd} \) or \( \text{Rm} \).

Condition flags

This instruction does not affect the flags.

Architectures

These instructions are available in ARMv6 and above.

Examples

- REV \( r3, r7 \)
- REV16 \( r0, r0 \)
- REVSH \( r0, r5 \); Reverse Signed Halfword
- REVHS \( r3, r7 \); Reverse with Higher or Same condition
4.4 ARM multiply instructions

This section contains the following subsections:

- **MUL and MLA** on page 4-53
  Multiply and multiply-accumulate (32-bit by 32-bit, bottom 32-bit result).

- **UMULL, UMLAL, SMULL and SMLAL** on page 4-55
  Unsigned and signed long multiply and multiply accumulate (32-bit by 32-bit, 64-bit result or 64-bit accumulator).

- **SMULxy** on page 4-57
  Signed multiply (16-bit by 16-bit, 32-bit result).

- **SMLAxy** on page 4-58
  Signed multiply-accumulate (16-bit by 16-bit, 32-bit accumulate).

- **SMULWy** on page 4-60
  Signed multiply (32-bit by 16-bit, top 32-bit result).

- **SMLAWy** on page 4-61
  Signed multiply-accumulate (32-bit by 16-bit, top 32-bit accumulate).

- **SMLALxy** on page 4-63
  Signed multiply-accumulate (16-bit by 16-bit, 64-bit accumulate).

- **SMUAD and SMUSD** on page 4-65
  Dual 16-bit signed multiply with addition or subtraction of products.

- **SMMUL** on page 4-67
  32-bit by 32-bit signed multiply, top 32-bit result.

- **SMLAD and SMLSD** on page 4-68
  Dual 16-bit signed multiply, 32-bit accumulation of sum or difference of 32-bit products.

- **SMMLA and SMMLS** on page 4-70
  32-bit by 32-bit signed multiply, 32-bit accumulation of top 32 bits of product.
  32-bit by 32-bit signed multiply, subtract top 32 bits of product from 32-bit value.

- **SMLALD and SMLSLD** on page 4-72
  Dual 16-bit signed multiply, 64-bit accumulation of sum or difference of 32-bit products.
- **UMAAL on page 4-74**  
  Unsigned multiply accumulate accumulate long.

- **MIA, MIAPH, and MIAxy on page 4-75**  
  XScale coprocessor 0 instructions.  
  Multiply with internal accumulate (32-bit by 32-bit, 40-bit accumulate).  
  Multiply with internal accumulate, packed halfwords (16-bit by 16-bit twice, 40-bit accumulate).  
  Multiply with internal accumulate (16-bit by 16-bit, 40-bit accumulate).
4.4.1 MUL and MLA

Multiply and multiply-accumulate (32-bit by 32-bit, bottom 32-bit result).

Syntax

\[
\begin{align*}
\text{MUL}\{\text{cond}\}\{S\} & \quad \text{Rd}, \text{Rm}, \text{Rs} \\
\text{MLA}\{\text{cond}\}\{S\} & \quad \text{Rd}, \text{Rm}, \text{Rs}, \text{Rn}
\end{align*}
\]

where:

- \textit{cond} is an optional condition code (see \textit{Conditional execution} on page 4-6).
- \textit{S} is an optional suffix. If \textit{S} is specified, the condition code flags are updated on the result of the operation (see \textit{Conditional execution} on page 4-6).
- \textit{Rd} is the destination register.
- \textit{Rm}, \textit{Rs}, \textit{Rn} are registers holding the operands.

Do not use r15 for \textit{Rd}, \textit{Rm}, \textit{Rs}, or \textit{Rn}.

\textit{Rd} cannot be the same as \textit{Rm}.

Usage

The \texttt{MUL} instruction multiplies the values from \textit{Rm} and \textit{Rs}, and places the least significant 32 bits of the result in \textit{Rd}.

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

Condition flags

If \textit{S} is specified, these instructions:

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

Architectures

These instructions are available in ARMv2 and above.
Examples

MUL     r10, r2, r5
MLA     r10, r2, r1, r5
MULS    r0, r2, r2
MULLT   r2, r3, r2
MLAVCS  r8, r6, r3, r8

Incorrect examples

MUL     r15, r0, r3 ; use of r15 not allowed
MLA     r1, r1, r6 ; Rd cannot be the same as Rm
4.4.2 UMULL, UMLAL, SMULL and SMLAL

Unsigned and signed long multiply and multiply accumulate (32-bit by 32-bit, 64-bit accumulate or result).

Syntax

\[ \text{Op}\{\text{cond}\}\{\text{S}\} \text{ RdLo, RdHi, Rm, Rs} \]

where:

- \( \text{Op} \) is one of UMULL, UMLAL, SMULL, or SMLAL.
- \( \text{cond} \) is an optional condition code (see Conditional execution on page 4-6).
- \( \text{S} \) is an optional suffix. If \( \text{S} \) is specified, the condition code flags are updated on the result of the operation (see Conditional execution on page 4-6).
- \( \text{RdLo}, \text{RdHi} \) are the destination registers. For UMLAL and SMLAL they also hold the accumulating value.
- \( \text{Rm}, \text{Rs} \) are ARM registers holding the operands.

Do not use r15 for \( \text{RdHi}, \text{RdLo}, \text{Rm}, \text{or Rs} \).

\( \text{RdLo}, \text{RdHi}, \text{and Rm} \) must all be different registers.

Usage

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

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

The SMULL instruction interprets the values from \( \text{Rm} \) and \( \text{Rs} \) as two’s complement signed integers. It multiplies these integers and places the least significant 32 bits of the result in \( \text{RdLo} \), and the most significant 32 bits of the result in \( \text{RdHi} \).

The SMLAL instruction interprets the values from \( \text{Rm} \) and \( \text{Rs} \) as two’s complement signed integers. It multiplies these integers, and adds the 64-bit result to the 64-bit signed integer contained in \( \text{RdHi} \) and \( \text{RdLo} \).
**Condition flags**

If S is specified, these instructions:

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

**Architectures**

These instructions are available in ARMv3M, and ARMv4 and above except xM variants.

**Examples**

```
UMULL       r0,r4,r5,r6
UMLALS      r4,r5,r3,r8
SMLALLES    r8,r9,r7,r6
SMULLNE     r0,r1,r9,r0 ; Rs can be the same as other
               ; registers
```

**Incorrect examples**

```
UMULL       r1,r15,r10,r2 ; use of r15 not allowed
SMULLLE     r0,r1,r0,r5 ; RdLo, RdHi and Rm must all be
                   ; different registers
```
4.4.3 SMULxy

Signed multiply (16-bit by 16-bit, 32-bit result).

Syntax

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

where:

- `<x>` is either B or T. B means use the bottom end (bits [15:0]) of Rm, T means use the top end (bits [31:16]) of Rm.
- `<y>` is either B or T. B means use the bottom end (bits [15:0]) of Rs, T means use the top end (bits [31:16]) of Rs.
- `cond` is an optional condition code (see Conditional execution on page 4-6).
- `Rd` is the destination register.
- `Rm`, `Rs` are the registers holding the values to be multiplied.

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

Any combination of `Rd`, `Rm`, and `Rs` can use the same registers.

Usage

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

Condition flags

SMULxy does not affect any flags.

Architectures

SMULxy is available in ARMv6 and above, and E variants of ARMv5.

Example

```
SMULTBEQ r8, r7, r9
```

Incorrect examples

```
SMULBT r15, r2, r0 ; use of r15 not allowed
SMULTTS r0, r6, r2 ; use of S suffix not allowed
```
4.4.4 SMLAxy

Signed multiply-accumulate (16-bit by 16-bit, 32-bit accumulate).

**Syntax**

\[ \text{SMLA}<x><y><\{\text{cond}\}) \text{ Rd, Rm, Rs, Rn} \]

where:

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

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

\(\text{cond}\) is an optional condition code (see *Conditional execution* on page 4-6).

\(Rd\) 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.

Do not use r15 for \(Rd, Rm, Rs,\) or \(Rn\).

Any combination of \(Rd, Rm, Rs,\) and \(Rn\) can use the same registers.

**Usage**

The SMLAxy instruction 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**

SMLAxy does not affect the N, Z, C, or V flags.

If overflow occurs in the accumulation, SMLAxy sets the Q flag. To read the state of the Q flag, use an MRS instruction (see *MRS* on page 4-113).

---

**Note**

SMLAxy never clears the Q flag. To clear the Q flag, use an MSR instruction (see *MSR* on page 4-114).
Architectures

SMLAxy is available in ARMv6 and above, and E variants of ARMv5.

Examples

SMLATT r8, r1, r0, r8  
SMLABBNE r0, r2, r1, r10  
SMLABT r0, r0, r3, r5

Incorrect examples

SMLATB r0, r7, r8, r15 ; use of r15 not allowed  
SMLATTS r0, r6, r2 ; use of S suffix not allowed
4.4.5 SMULWy

Signed multiply (32-bit by 16-bit, top 32-bit result).

Syntax

\[
\text{SMULWy} \{\text{cond}\} \ Rd, \ Rm, \ Rs
\]

where:

\(<y>\) is either \( \text{B} \) or \( \text{T} \). \( \text{B} \) means use the bottom end (bits \([15:0]\)) of \( Rs \), \( \text{T} \) means use the top end (bits \([31:16]\)) of \( Rs \).

\text{cond} is an optional condition code (see Conditional execution on page 4-6).

\( Rd \) is the destination register.

\( Rm, \ Rs \) are the registers holding the operands.

Do not use \( r15 \) for \( Rd, Rm, \) or \( Rs \).

Any combination of \( Rd, Rm, \) and \( Rs \) can use the same registers.

Usage

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 \( Rd \).

Condition flags

SMULWy does not affect any flags.

Architectures

SMULWy is available in ARMv6 and above, and E variants of ARMv5.

Examples

\[
\text{SMULWB} \quad r2, r4, r7 \\
\text{SMULWT} \quad r0, r0, r9
\]

Incorrect examples

\[
\text{SMULWT} \quad r15, r9, r3 \quad ; \text{use of } r15 \text{ not allowed} \\
\text{SMULWBS} \quad r0, r4, r5 \quad ; \text{use of } S \text{ suffix not allowed}
\]
4.4.6 SMLAWy

Signed multiply-accumulate (32-bit by 16-bit, top 32-bit accumulate).

Syntax

\[ \text{SMLAW}<y>\{\text{cond}\} \text{ Rd, Rm, Rs, Rn} \]

where:

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

\(\text{cond}\) is an optional condition code (see Conditional execution on page 4-6).

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

\(\text{Rm}, \text{ Rs}\) are the registers holding the values to be multiplied.

\(\text{Rn}\) is the register holding the value to be added.

Do not use r15 for \(\text{Rd}, \text{ Rm}, \text{ Rs}, \text{ or Rn}\).

Any combination of \(\text{Rd}, \text{ Rm}, \text{ Rs}, \text{ and Rn}\) can use the same registers.

Usage

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

SMLAWy does not affect the N, Z, C or V flags.

If overflow occurs in the accumulation, SMLAWy sets the Q flag. To read the state of the Q flag, use an MRS instruction (see MRS on page 4-113).

Note

SMLAWy never clears the Q flag. To clear the Q flag, use an MSR instruction (see MSR on page 4-114).

Architectures

SMLAWy is available in ARMv6 and above, and E variants of ARMv5.
Examples

SMLAWB      r2, r4, r7, r1
SMLAWT      r0, r0, r9, r2

Incorrect examples

SMLAWT      r15, r9, r3, r1 ; use of r15 not allowed
SMLAWBS     r0, r4, r5, r1 ; use of S suffix not allowed
4.4.7 SMLALxy

Signed multiply-accumulate (16-bit by 16-bit, 64-bit accumulate).

**Syntax**

`SMLAL<x><y>{cond} RdLo, RdHi, Rm, Rs`

where:

- `<x>` is either B or T. B means use the bottom end (bits [15:0]) of `Rm`, T means use the top end (bits [31:16]) of `Rm`.
- `<y>` is either B or T. B means use the bottom end (bits [15:0]) of `Rs`, T means use the top end (bits [31:16]) of `Rs`.
- `cond` is an optional condition code (see Conditional execution on page 4-6).

`RdHi, RdLo` are the destination registers. They also hold the accumulate value.

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

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

Any combination of `RdHi, RdLo, Rm, or Rs` can use the same registers.

**Usage**

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**

SMLALxy does not affect any flags.

---

**Note**

SMLALxy cannot raise an exception. If overflow occurs on this instruction, the result wraps round without any warning.

---

**Architectures**

SMLALxy is available in ARMv6 and above, and E variants of ARMv5.
Examples

  SMLALTB     r2, r3, r7, r1
  SMLALBTVS    r0, r1, r9, r2

Incorrect examples

  SMLALTT     r8, r9, r3, r15  ; use of r15 not allowed
  SMLALBBS    r0, r1, r5, r2  ; use of S suffix not allowed
4.4.8 SMUAD and SMUSD

Dual 16-bit signed multiply with addition or subtraction of products.

**Syntax**

\[
\text{op}\{X\}\{\text{cond}\} \; \text{Rd}, \; \text{Rm}, \; \text{Rs}
\]

where:

- \textit{op} is one of:
  - \text{SMUAD} \quad \text{Dual multiply, add products.}
  - \text{SMUSD} \quad \text{Dual multiply, subtract products.}

- \textit{X} is an optional parameter. If \textit{X} is present, the most and least significant halfwords of the second operand are exchanged before the multiplications occur.

- \textit{cond} is an optional condition code (see *Conditional execution* on page 4-6).

- \textit{Rd} is the destination register.

- \textit{Rm} is the register holding the first operand.

- \textit{Rs} is the register holding the second operand.

Do not use r15 for any of \textit{Rd}, \textit{Rm}, or \textit{Rs}.

**Operation**

\text{SMUAD} multiplies the bottom halfword of \textit{Rm} with the bottom halfword of the second operand, and the top halfword of \textit{Rm} with the top halfword of the second operand. It then adds the products and stores the sum to \textit{Rd}.

\text{SMUSD} multiplies the bottom halfword of \textit{Rm} with the bottom halfword of the second operand, and the top halfword of \textit{Rm} with the top halfword of the second operand. It then subtracts the second product from the first, and stores the difference to \textit{Rd}.

**Condition flags**

These instructions do not affect any flags.

**Architectures**

These instructions are available in ARMv6 and above.
Examples

SMUAD  r2, r3, r2
SMUSDXNE  r0, r1, r2
4.4.9  SMMUL

32-bit by 32-bit signed multiply, producing only the most significant 32-bits of the result.

**Syntax**

\[
\text{SMMUL}\{R\}\{\text{cond}\}\ Rd,\ Rm,\ Rs
\]

where:

- **R** is an optional parameter. If it is present, the result is rounded, otherwise it is truncated.
- **cond** is an optional condition code (see *Conditional execution* on page 4-6).
- **Rd** is the destination register.
- **Rm** is the register holding the first operand.
- **Rs** is the register holding the second operand.

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

**Operation**

SMMUL multiplies the values from **Rm** and **Rs**, and stores the most significant 32 bits of the 64-bit result to **Rd**.

If the optional **R** parameter is specified, \(0x80000000\) is added before extracting the most significant 32 bits. This has the effect of rounding the result.

**Condition flags**

SMMUL does not affect any flags.

**Architectures**

SMMUL is available in ARMv6 and above.

**Examples**

- SMMULGE r6, r4, r3
- SMMULR r2, r2, r2
4.4.10 SMLAD and SMLSD

Dual 16-bit signed multiply with addition or subtraction of products and 32-bit accumulation.

Syntax

\( \text{op}\{X\}\{\text{cond}\}: R_d, R_m, R_s, R_n \)

where:

- \( \text{op} \) is one of:
  - SMLAD: Dual multiply, accumulate sum of products.
  - SMLSD: Dual multiply, accumulate difference of products.

- \( \text{cond} \) is an optional condition code (see Conditional execution on page 4-6).

- \( X \) is an optional parameter. If \( X \) is present, the most and least significant halfwords of the second operand are exchanged before the multiplications occur.

- \( R_d \) is the destination register.

- \( R_m \) is the register holding the first operand.

- \( R_s \) is the register holding the second operand.

- \( R_n \) is the register holding the accumulate operand.

Do not use \( r15 \) for any of \( R_d, R_m, R_s, \) or \( R_n \).

Operation

SMLAD multiplies the bottom halfword of \( R_m \) with the bottom halfword of the second operand, and the top halfword of \( R_m \) with the top halfword of the second operand. It then adds both products to the value in \( R_n \) and stores the sum to \( R_d \).

SMLSD multiplies the bottom halfword of \( R_m \) with the bottom halfword of the second operand, and the top halfword of \( R_m \) with the top halfword of the second operand. It then subtracts the second product from the first, adds the difference to the value in \( R_n \), and stores the result to \( R_d \).

Condition flags

These instructions do not affect any flags.
Architectures

These instructions are available in ARMv6 and above.

Examples

SMLSD     r1, r2, r0, r7
SMLSDX    r11, r10, r2, r3
SMLADLT   r1, r2, r4, r1
### 4.4.11 SMMLA and SMMLS

32-bit by 32-bit signed multiply, top 32-bit result, with 32-bit accumulation.

32-bit by 32-bit signed multiply, top 32-bit result, subtract from 32-bit value.

#### Syntax

```plaintext
op{R}{cond} Rd, Rm, Rs, Rn
```

where:

- **op** is one of:
  - **SMMLA** Multiply, accumulate, and truncate or round.
  - **SMMLS** Multiply, subtract from \( Rn \) and truncate or round.
- **R** is an optional parameter. If \( R \) is present, the result is rounded, otherwise it is truncated.
- **cond** is an optional condition code (see *Conditional execution* on page 4-6).
- **Rd** is the destination register.
- **Rm** is the register holding the first operand.
- **Rs** is the register holding the second operand.
- **Rn** is the register holding the accumulate operand.

Do not use r15 for any of \( Rd, Rm, Rs, \) or \( Rn \).

#### Operation

**SMMLA** multiplies the values from \( Rm \) and \( Rs \), adds the value in \( Rn \) to the most significant 32 bits of the product, and stores the result in \( Rd \).

**SMMLS** multiplies the values from \( Rm \) and \( Rs \), subtracts the product from the value in \( Rn \) shifted left by 32 bits, and stores the most significant 32 bits of the result in \( Rd \).

If the optional \( R \) parameter is specified, \( 0x80000000 \) is added before extracting the most significant 32 bits. This has the effect of rounding the result.

#### Condition flags

These instructions do not affect any flags.
Architectures

These instructions are available in ARMv6 and above.

Examples

SMMLAREQ r0, r3, r7, r5
SMMLS r2, r1, r5, r3
4.4.12 SMLALD and SMLSLD

Dual 16-bit signed multiply with addition or subtraction of products and 64-bit accumulation.

Syntax

\[ \text{op}\{X\}\{\text{cond}\} \text{ RdLo, RdHi, Rm, Rs } \]

where:

- \textit{op} is one of:
  - \texttt{SMLALD} Dual multiply, accumulate sum of products.
  - \texttt{SMLSLD} Dual multiply, accumulate difference of products.

- \textit{X} is an optional parameter. If \textit{X} is present, the most and least significant halfwords of the second operand are exchanged before the multiplications occur.

- \textit{cond} is an optional condition code (see \textit{Conditional execution} on page 4-6).

- \textit{RdLo, RdHi} are the destination registers for the 64-bit result. They also hold the 64-bit accumulate operand.

- \textit{Rm} is the register holding the first operand.

- \textit{Rs} is the register holding the second operand.

Do not use r15 for any of \textit{RdLo, RdHi, Rm, or Rs}.

Operation

\texttt{SMLALD} multiplies the bottom halfword of \textit{Rm} with the bottom halfword of the second operand, and the top halfword of \textit{Rm} with the top halfword of the second operand. It then adds both products to the value in \textit{RdLo, RdHi} and stores the sum to \textit{RdLo, RdHi}.

\texttt{SMLSLD} multiplies the bottom halfword of \textit{Rm} with the bottom halfword of the second operand, and the top halfword of \textit{Rm} with the top halfword of the second operand. It then subtracts the second product from the first, adds the difference to the value in \textit{RdLo, RdHi}, and stores the result to \textit{RdLo, RdHi}.

Condition flags

These instructions do not affect any flags.
Architectures

These instructions are available in ARMv6 and above.

Examples

```
SMLALD      r10, r11, r5, r1
SMLSLD      r3, r0, r5, r1
```
4.4.13 UMAAL

Unsigned multiply accumulate accumulate long.

Syntax

UMAAL{cond} RdLo, RdHi, Rm, Rs

where:

cond is an optional condition code (see Conditional execution on page 4-6).

RdLo, RdHi are the destination registers for the 64-bit result. They also hold the two
32-bit accumulate operands.

Rm is the register holding the first multiply operand.

Rs is the register holding the second multiply operand.

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

Operation

The UMAAL instruction multiplies the 32-bit values in Rm and Rs, adds the two 32-bit
values in RdHi and RdLo, and stores the 64-bit result to RdLo, RdHi.

Condition flags

UMAAL does not affect any flags.

Architectures

UMAAL is available in ARMv6 and above.

Examples

UMAAL r8, r9, r2, r3
UMAALGE r2, r0, r5, r3
4.4.14 MIA, MIAPH, and MIAxy

XScale coprocessor 0 instructions.

Multiply with internal accumulate (32-bit by 32-bit, 40-bit accumulate).

Multiply with internal accumulate, packed halfwords (16-bit by 16-bit twice, 40-bit accumulate).

Multiply with internal accumulate (16-bit by 16-bit, 40-bit accumulate).

Syntax

MIA\{cond\} Acc, Rm, Rs
MIAPH\{cond\} Acc, Rm, Rs
MIA<x><y>\{cond\} Acc, Rm, Rs

where:

\(cond\) is an optional condition code (see Conditional execution on page 4-6).

Acc is the internal accumulator. The standard name is acc\(x\), where \(x\) is an integer in the range 0-\(n\). The value of \(n\) depends on the processor. It is 0 in current processors.

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

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

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

Do not use r15 for either Rm or Rs.

Usage

The MIA instruction multiplies the signed integers from Rs and Rm, and adds the result to the 40-bit value in Acc.

The MIAPH instruction multiplies the signed integers from the lower halves of Rs and Rm, multiplies the signed integers from the upper halves of Rs and Rm, and adds the two 32-bit results to the 40-bit value in Acc.

The MIAxy instruction multiplies the signed integer from the selected half of Rs by the signed integer from the selected half of Rm, and adds the 32-bit result to the 40-bit value in Acc.
Condition flags

These instructions do not affect any flags.

--- Note ---

These instructions cannot raise an exception. If overflow occurs on these instructions, the result wraps round without any warning.

Architectures

These instructions are only available in XScale processors.

Examples

MIA    acc0,r5,r0
MIALE  acc0,r1,r9
MIAPH  acc0,r0,r7
MIAPHNE acc0,r11,r10
MIABB  acc0,r8,r9
MIABT  acc0,r8,r8
MIATB  acc0,r5,r3
MIATT  acc0,r0,r6
MIABTGT acc0,r2,r5
4.5 ARM saturating instructions

This section contains the following subsections:

- What saturating means
- QADD, QSUB, QDADD, and QDSUB on page 4-78
- SSAT and USAT on page 4-80.

Some of the parallel instructions are also saturating, see ARM parallel instructions on page 4-82.

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 4-114).

---

The Q flag can also be set by two other instructions (see SMLAxy on page 4-58 and SMLAWy on page 4-61), but these instructions do not saturate.
4.5.2 QADD, QSUB, QDADD, and QDSUB

Signed add, subtract, double and add, double and subtract, saturating the result to the
signed range \(-2^{31} \leq x \leq 2^{31} - 1\).

See also Parallel add and subtract on page 4-83.

Syntax

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

where:

- \text{op} is one of QADD, QSUB, QDADD, or QDSUB.
- \text{cond} is an optional condition code (see Conditional execution on page 4-6).
- \text{Rd} is the destination register.
- \text{Rm, Rn} are the registers holding the operands.

Do not use r15 for \text{Rd, Rm, or Rn}.

Usage

The QADD instruction adds the values in \text{Rm} and \text{Rn}.

The QSUB instruction subtracts the value in \text{Rn} from the value in \text{Rm}.

The QDADD instruction calculates \(\text{SAT(Rm + SAT(Rn \times 2))}\). Saturation can occur on the
doubling operation, on the addition, or on both. If saturation occurs on the doubling but
not on the addition, the Q flag is set but the final result is unsaturated.

The QDSUB instruction calculates \(\text{SAT(Rm - SAT(Rn \times 2))}\). Saturation can occur on the
doubling operation, on the subtraction, or on both. If saturation occurs on the doubling
but not on the subtraction, the Q flag is set but the final result is unsaturated.

——— Note ————

All values are treated as two's complement signed integers by these instructions.

See also Parallel add and subtract on page 4-83 for similar parallel instructions,
available in ARMv6 and above only.
**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 4-113).

No other flags are affected.

**Architectures**

These instructions are available in ARMv6 and above, and E variants of ARMv5.

**Examples**

```
QADD    r0, r1, r9
QOSUBLT r9, r0, r1
```

**Incorrect examples**

```
QSUBS   r3, r4, r2 ; use of S suffix not allowed
QADD    r11, r15, r0 ; use of r15 not allowed
```
4.5.3 SSAT and USAT

Signed saturate and unsigned saturate to any bit position, with optional shift before saturating.

See also SSAT16 and USAT16 on page 4-88.

Syntax

\[ \text{op}\{\text{cond}\} \, \text{Rd}, \, \#\text{sat\_imm}, \, \text{Rm}\{, \, \text{shift}\} \]

where:

- \text{op} is either SSAT or USAT.
- \text{cond} is an optional condition code (see Conditional execution on page 4-6).
- \text{Rd} is the destination register.
- \text{sat\_imm} specifies the bit position to saturate to, and is in the range 0 to 31.
- \text{Rm} is the ARM registers holding the operand.
- \text{shift} is one of:
  - \text{LSL} \, \#n where \( n \) is in the range 0 to 31
  - \text{ASR} \, \#n where \( n \) is in the range 1 to 32.

Do not use r15 for \text{Rd} or \text{Rm}.

Operation

The SSAT instruction applies the specified shift, then saturates to the signed range
\[-2^{\text{sat\_imm}} - 1 \leq x \leq 2^{\text{sat\_imm}} - 1 - 1.\]

The USAT instruction applies the specified shift, then saturates to the unsigned range
\[0 \leq x \leq 2^{\text{sat\_imm}} - 1.\]

Condition flags

If saturation occurs, these instructions set the Q flag. To read the state of the Q flag, use an MRS instruction (see MRS on page 4-113).

Architectures

These instructions are available in ARMv6 and above.
Examples

SSAT r7, #16, r7, LSL #4
USATNE r0, #7, r5

Incorrect example

USATGT r0, #7, r15, ASR #16 ; use of r15 not allowed


4.6 ARM parallel instructions

This section contains the following subsections:

- **Parallel add and subtract** on page 4-83
  Various byte-wise and halfword-wise additions and subtractions.

- **USAD8 and USADA8** on page 4-86
  Unsigned sum of absolute differences, and accumulate unsigned sum of absolute differences.

- **SSAT16 and USAT16** on page 4-88
  Parallel halfword saturating instructions.

There are also some parallel unpacking instructions, see **SXT, SXTA, UXT, and UXTA** on page 4-91.
4.6.1 Parallel add and subtract

Various byte-wise and halfword-wise additions and subtractions.

**Syntax**

\(<prefix>op\{cond\} \text{ Rd, Rn, Rm}\)

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.
  - ADDSUBX Exchange halfwords of \(Rm\), then add top halfwords and subtract bottom halfwords.
  - SUBADDX Exchange halfwords of \(Rm\), then subtract top halfwords and add bottom halfwords.

- \(cond\) is an optional condition code (see Conditional execution on page 4-6).

- \(Rd\) 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.
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 $-2^{15} \leq x \leq 2^{15} - 1$ or $-2^7 \leq x \leq 2^7 - 1$. The Q flag is not affected even if these operations saturate.

- Unsigned saturating arithmetic to one of the unsigned ranges $0 \leq x \leq 2^{16} - 1$ or $0 \leq x \leq 2^8 - 1$. The Q flag is not affected even if these operations saturate.

- Signed or unsigned arithmetic, halving the results. This cannot cause overflow.

**Condition flags**

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

The Q, S!, UQ, and UH prefix variants of these instructions do not affect any 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

- For halfword-wise operations, the GE flags are used in the same way as the C (Carry) flag for normal word-wise SUB and ADD instructions:
  
  GE[1:0] for bits[15:0] of the result

You can use these flags to control a following SEL instruction, see SEL on page 4-48.

**Note**

For halfword-wise operations, GE[1:0] are set or cleared together, and GE[3:2] are set or cleared together.

**Architectures**

These instructions are available in ARMv6 and above.
Examples

SHADD8      r4, r3, r9
USUBADDXNE  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 allowed
SUBADDX     r10, r8, r5 ; Must have a prefix.
4.6.2 USAD8 and USADA8

Unsigned sum of absolute differences, and accumulate unsigned sum of absolute differences.

Syntax

USAD8{cond} Rd, Rm, Rs
USADA8{cond} Rd, Rm, Rs, Rn

where:

cond is an optional condition code (see Conditional execution on page 4-6).

Rd is the destination register.

Rm is the register holding the first operand.

Rs is the register holding the second operand.

Rn is the register holding the accumulate operand.

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

Operation

The USAD8 instruction finds the four differences between the unsigned values in corresponding bytes of Rm and Rs. It adds the absolute values of the four differences, and saves the result to Rd.

The USADA8 instruction adds the absolute values of the four differences to the value in Rn, and saves the result to Rd.

Condition flags

These instructions do not set any flags.

Architectures

These instructions are available in ARMv6 and above.

Examples

USAD8 r2, r4, r6
USADA8 r0, r3, r5, r2
USADA8VS r0, r4, r0, r1
Incorrect examples

USADA8  r2, r4, r6  ; USADA8 needs four registers
USAD8CC r0, r3, r15 ; use of r15 not allowed
USADA16 r0, r4, r0, r1 ; no such instruction
4.6.3 SSAT16 and USAT16

Parallel halfword saturating instructions.

Syntax

\[ \text{op}\{\text{cond}\} R_d, \#\text{sat_imm}, R_m \]

where:

\[ \text{op} \] is one of:

- \text{SSAT16} Signed saturation.
- \text{USAT16} Unsigned saturation.

\[ \text{cond} \] is an optional condition code (see \textit{Conditional execution} on page 4-6).

\[ R_d \] is the destination register.

\[ \text{sat_imm} \] specifies the bit position to saturate to, and is in the range 1 to 16 for \text{SSAT16}, or 0 to 15 for \text{USAT16}.

\[ R_m \] is the register holding the operand.

Do not use r15 for \( R_d \) or \( R_m \).

Operation

Halfword-wise signed and unsigned saturation to any bit position.

The \text{SSAT16} instruction saturates each halfword to the signed range

\[-2^{\text{sat_imm}} - 1 \leq x \leq 2^{\text{sat_imm}} - 1 - 1.\]

The \text{USAT16} instruction saturates each halfword to the unsigned range

\[0 \leq x \leq 2^{\text{sat_imm}} - 1.\]

Condition flags

If saturation occurs on either halfword, these instructions set the Q flag. To read the state of the Q flag, use an \textit{MRS} instruction (see \textit{MRS} on page 4-113).

Architectures

These instructions are available in ARMv6 and above.
Examples

SSAT16   r7, #12, r7
USAT16   r0, #7, r5

Incorrect examples

SSAT16   r1, #16, r2, LSL #4 ; shifts not allowed with halfword saturations
USAT16   r0, #11, r15        ; use of r15 not allowed
4.7 ARM packing and unpacking instructions

This section contains the following subsections:

- **SXT, SXTA, UXT, and UXTA** on page 4-91
  Sign extend or zero extend instructions, with optional add.

- **PKHBT and PKHTB** on page 4-94
  Halfword packing instructions.
4.7.1  SXT, SXTA, UXT, and UXTA

Sign extend or zero extend, with optional add.

These instructions do any one of the following:

- sign or zero extend an 8-bit value to 32 bits, and optionally add a 32-bit value
- sign or zero extend a 16-bit value to 32 bits, and optionally add a 32-bit value
- sign or zero extend two 8-bit values to two 16-bit values, and optionally add two 16-bit values.

Syntax

\[ \text{op}\{\text{extend}\}{\text{cond}} \text{ Rd, Rm\{, rotation\}} \]
\[ \text{opA}\{\text{extend}\}{\text{cond}} \text{ Rd, Rn, Rm\{, rotation\}} \]

where:

- \(\text{op}\) is one of:
  - SXT  sign extend.
  - UXT  zero extend.
- \(\text{A}\) indicates that the instruction is extend and add.
- \(\{\text{extend}\}\) is one of:
  - B16 extends two 8-bit values to two 16-bit values.
  - B extends an 8-bit value to a 32-bit value.
  - H extends a 16-bit value to a 32-bit value.
- \(\text{cond}\) is an optional condition code (see Conditional execution on page 4-6).
- \(\text{Rd}\) is the destination register. Must not be r15.
- \(\text{Rn}\) is the register holding the first operand. Must not be r15.
- \(\text{Rm}\) is the register holding the second operand. Must not be r15.
- \(\text{rotation}\) is one of:
  - ROR \#8 the value from \(\text{Rm}\) is rotated right 8 bits.
  - ROR \#16 the value from \(\text{Rm}\) is rotated right 16 bits.
  - ROR \#24 the value from \(\text{Rm}\) is rotated right 24 bits.

If \(\text{rotation}\) is omitted, no rotation is performed.
Operation

These instructions do the following:

1. Rotate the value from Rm right by 0, 8, 16 or 24 bits.
2. Do one of the following to the value obtained:
   - Extract bits[7:0], sign or zero extend to 32 bits. If the instruction is extend and add, add the value from Rn.
   - Extract bits[15:0], sign or zero extend to 32 bits. If the instruction is extend and add, add the value from Rn.
   - Extract bits[23:16] and bits[7:0] and sign or zero extend them to 16 bits. If the instruction is extend and add, add them to bits[31:16] and bits[15:0] respectively of Rn to form bits[31:16] and bits[15:0] of the result.

Condition flags

These instructions do not affect any flags.

Architectures

These instructions are available in ARMv6 and above.

Examples

<table>
<thead>
<tr>
<th>Instruction</th>
<th>R3, R9, R4</th>
</tr>
</thead>
<tbody>
<tr>
<td>SXTH</td>
<td>r3, r9, r4</td>
</tr>
<tr>
<td>UXTAB16EQ</td>
<td>r0, r0, r4, ROR #16</td>
</tr>
</tbody>
</table>

Incorrect examples

<table>
<thead>
<tr>
<th>Instruction</th>
<th>R0, R2, R15</th>
</tr>
</thead>
<tbody>
<tr>
<td>UXTAB</td>
<td>r0, r2, r15        ; use of r15 not allowed</td>
</tr>
<tr>
<td>SXTH</td>
<td>r9, r3, r2, ROR #12; rotation must be by 0, 8, 16, or 24.</td>
</tr>
</tbody>
</table>
Synonyms

There are synonyms for these instructions. The use of these pseudonyms is deprecated. The synonyms are shown in Table 4-3.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Synonym</th>
</tr>
</thead>
<tbody>
<tr>
<td>SXTAH</td>
<td>SADD16TO32</td>
</tr>
<tr>
<td>SXTAB</td>
<td>SADD8TO32</td>
</tr>
<tr>
<td>SXTAB16</td>
<td>SADD8TO16</td>
</tr>
<tr>
<td>SXTH</td>
<td>SUNPK16TO32</td>
</tr>
<tr>
<td>SXTB</td>
<td>SUNPK8TO32</td>
</tr>
<tr>
<td>SXTB16</td>
<td>SUNPK8TO16</td>
</tr>
<tr>
<td>UXTAH</td>
<td>UADD16TO32</td>
</tr>
<tr>
<td>UXTAB</td>
<td>UADD8TO32</td>
</tr>
<tr>
<td>UXTAB16</td>
<td>UADD8TO16</td>
</tr>
<tr>
<td>UXTH</td>
<td>UUNPK16TO32</td>
</tr>
<tr>
<td>UXTB</td>
<td>UUNPK8TO32</td>
</tr>
<tr>
<td>UXTB16</td>
<td>UUNPK8TO16</td>
</tr>
</tbody>
</table>
4.7.2 PKHBT and PKHTB

Halfword packing instructions.

Combine a halfword from one register with a halfword from another register. One of the operands can be shifted before extraction of the halfword.

Syntax

\[ op\{cond\} \text{Rd, Rn, Rm\{, shift\} \]

where:

- **op** is one of:
  - \text{PKHBT} combines bits[15:0] of \text{Rn} with bits[31:16] of the shifted value from \text{Rm}.
  - \text{PKHTB} combines bits[31:16] of \text{Rn} with bits[15:0] of the shifted value from \text{Rm}.

- **cond** is an optional condition code (see Conditional execution on page 4-6).

- **Rd** is the destination register.

- **Rn** is the register holding the first operand.

- **Rm** is the register holding the second operand.

- **shift** is one of:
  - \text{LSL \#n} where \( n \) is in the range 0 to 31. Only available for PKHBT.
  - \text{ASR \#n} where \( n \) is in the range 1 to 32. Only available for PKHTB.

Do not use r15 for \text{Rd, Rm, or Rn}.

**Condition flags**

These instructions do not affect any flags.

**Architectures**

These instructions are available in ARMv6 and above.
Examples

PKHBT r0, r3, r5          ; combine the bottom halfword of r3 with the top halfword of r5
PKHBT r0, r3, r5, LSL #16; combine the bottom halfword of r3 with the bottom halfword of r5
PKHTB r0, r3, r5, ASR #16; combine the top halfword of r3 with the top halfword of r5

You can also scale the second operand by using different values of shift.

Incorrect examples

PKHBT r4, r15, r1         ; use of r15 not allowed
PKHBTEQ r4, r5, r1, ASR #8 ; ASR not allowed with PKHBTEQ
4.8 ARM branch instructions

This section contains the following subsections:

- *B and BL* on page 4-97
  Branch, and Branch with Link.

- *BX* on page 4-98
  Branch and exchange instruction set.

- *BLX* on page 4-99
  Branch with Link and exchange instruction set.

- *BXJ* on page 4-101
  Branch and change instruction set to Java.
4.8.1 B and BL

Branch, and Branch with Link.

Syntax

\[ \text{B}\{\text{cond}\} \, \text{label} \]
\[ \text{BL}\{\text{cond}\} \, \text{label} \]

where:

\text{cond} is an optional condition code (see Conditional execution on page 4-6).

\text{label} is a program-relative expression. See Register-relative and program-relative expressions on page 3-29 for more information.

Usage

The B instruction causes a branch to \text{label}.

The BL instruction copies the address of the next instruction into r14 (lr, the link register), and causes a branch to \text{label}.

Machine-level B and BL instructions have a range of ±32Mb from the address of the current instruction. However, you can use these instructions even if \text{label} is out of range. Often you do not know where \text{label} is placed by the linker. When necessary, the ARM linker adds code to allow longer branches (see The ARM linker chapter in RealView Compilation Tools v2.1 Linker and Utilities Guide). The added code is called a veneer.

Architectures

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

Examples

\[ \text{B} \, \text{loopA} \]
\[ \text{BLE} \, \text{ng+8} \]
\[ \text{BL} \, \text{subC} \]
\[ \text{BLT} \, \text{rtX} \]
4.8.2 BX

Branch, and optionally exchange instruction set.

Syntax

BX{cond} Rm

where:

cond is an optional condition code (see Conditional execution on page 4-6).
Rm is an ARM register containing the address to branch to.

Bit 0 of Rm is not used as part of the address.
If bit 0 of Rm is set, the instruction sets the T flag in the CPSR, and the code at the destination is interpreted as Thumb code.
If bit 0 of Rm is clear, bit 1 must not be set.

Usage

The BX instruction causes a branch to the address held in Rm, and changes instruction set to Thumb if bit 0 of Rm is set.

Architectures

This instruction is available in all T variants of the ARM architecture, and ARMv5 and above.

Examples

BX r7
BXVS r0
4.8.3 BLX

Branch with Link, and optionally exchange instruction set. This instruction has the following alternative forms:
- an unconditional branch with link to a program-relative address
- a conditional branch with link to an absolute address held in a register.

Syntax

\[
\text{BLX}\{\text{cond}\} \ Rm \\
\text{BLX } \text{label}
\]

where:

- \text{cond} is an optional condition code (see Conditional execution on page 4-6).
- \text{Rm} is an ARM register containing the address to branch to.
  
  Bit 0 of \text{Rm} is not used as part of the address.
  
  If bit 0 of \text{Rm} is set, the instruction sets the T flag in the CPSR, and the code at the destination is interpreted as Thumb code.
  
  If bit 0 of \text{Rm} is clear, bit 1 must not be set.

- \text{label} is a program-relative expression. See Register-relative and program-relative expressions on page 3-29 for more information.

\begin{itemize}
  \item Note
\end{itemize}

BLX \text{label} cannot be conditional. BLX \text{label} always causes a change to Thumb state.

Usage

The BLX instruction:
- copies the address of the next instruction into r14 (lr, the link register)
- causes a branch to \text{label}, or to the address held in \text{Rm}
- changes instruction set to Thumb if either:
  
  \begin{itemize}
    \item bit 0 of \text{Rm} is set
    \item the BLX \text{label} form is used.
  \end{itemize}

The machine-level BLX \text{label} instruction cannot branch to an address outside ±32Mb of the current instruction. When necessary, the ARM linker adds code to allow longer branches (see The ARM linker chapter in RealView Compilation Tools v2.1 Linker and Utilities Guide). The added code is called a veneer.
Architectures

This instruction is available in all T variants of ARMv5 and above.

Examples

BLX r2
BLXNE r0
BLX thumbsub

Incorrect example

BLXMI thumbsub ; BLX label cannot be conditional
4.8.4 BXJ

Change instruction set to Java, or operate in the same way as the equivalent BX instruction if Java state is not available.

Syntax

BXJ{cond} Rm

where:

cond is an optional condition code (see Conditional execution on page 4-6).

Rm is an ARM register containing the address to branch to if Java state is not available. Do not use r15 for Rm.

If Java state is not available:

- Bit 0 of Rm is not used as part of the address.
- If bit 0 of Rm is set, the instruction sets the T flag in the CPSR, and the code at the destination is interpreted as Thumb code.
- If bit 0 of Rm is clear, bit 1 must not be set.

Usage

The BXJ instruction causes a change to Java state if possible. Otherwise, it causes a branch to the address held in Rm, and changes instruction set to Thumb if bit 0 of Rm is set.

Architectures

This instruction is available in all J variants of the ARM architecture, and ARMv6 and above.

Examples

    BXJ r7
    BXJGE r0
4.9 Coprocessor instructions

This section does not describe Vector Floating-point instructions (see Chapter 6 Vector Floating-point Programming).

It contains the following sections:

- **CDP, CDP2** on page 4-103
  Coprocessor data operations

- **MCR, MCR2, MCRR, and MCRR2** on page 4-104
  Move to coprocessor from ARM registers, possibly with coprocessor operations

- **MRC, MRC2** on page 4-105
  Move to ARM register from coprocessor, possibly with coprocessor operations

- **MRRC and MRRC2** on page 4-106
  Move to two ARM registers from coprocessor, possibly with coprocessor operations

- **LDC, STC** on page 4-107
  Transfer data between memory and coprocessor.
4.9.1 CDP, CDP2

Coprocessor data operations.

Syntax

CDP\{\textit{cond}\} \textit{coproc}, \textit{opcode1}, \textit{CRd}, \textit{CRn}, \textit{CRm}\{, \textit{opcode2}\}

CDP2 \textit{coproc}, \textit{opcode1}, \textit{CRd}, \textit{CRn}, \textit{CRm}\{, \textit{opcode2}\}

where:

\textit{cond} is an optional condition code (see \textit{Conditional execution} on page 4-6).

\textit{coproc} is the name of the coprocessor the instruction is for. The standard name is \textit{pn}, where \textit{n} is an integer in the range 0-15.

\textit{opcode1} is a coprocessor-specific opcode.

\textit{CRd}, \textit{CRn}, \textit{CRm} are coprocessor registers.

\textit{opcode2} is an optional coprocessor-specific opcode.

Usage

The use of these instructions depends on the coprocessor. See the coprocessor documentation for details.

\begin{center} \textbf{Note} \end{center}

CDP2 is always unconditional.

\begin{center} \textbf{Architectures} \end{center}

CDP is available in ARMv2 and above.

CDP2 is available in ARMv5 and above.
4.9.2 MCR, MCR2, MCRR, and MCRR2

Move to coprocessor from ARM registers. Depending on the coprocessor, you might be able to specify various operations in addition.

Syntax

\[
\text{MCR}\{\text{cond}\} \coproc, \text{opcode1}, \text{rd}, \text{crn}, \text{crm}, \{, \text{opcode2}\}
\]

\[
\text{MCR2} \coproc, \text{opcode1}, \text{rd}, \text{crn}, \text{crm}
\]

\[
\text{MCRR}\{\text{cond}\} \coproc, \text{opcode1}, \text{rd}, \text{rn}, \text{crm}
\]

\[
\text{MCRR2} \coproc, \text{opcode1}, \text{rd}, \text{rn}, \text{crm}
\]

where:

- \text{cond} is an optional condition code (see Conditional execution on page 4-6).
- \text{coproc} is the name of the coprocessor the instruction is for. The standard name is \text{p}\text{n}, where \text{n} is an integer in the range 0-15.
- \text{opcode1} is a coprocessor-specific opcode.
- \text{rd, rn} are ARM source registers. Do not use \text{r15} for \text{rd} or \text{rn}.
- \text{crn,crm} are coprocessor registers.
- \text{opcode2} is an optional coprocessor-specific opcode.

Usage

The use of these instructions depends on the coprocessor. See the coprocessor documentation for details.

--- **Note** ---

\text{MCR2} and \text{MCRR2} are always unconditional.

---

Architectures

\text{MCR} is available in ARMv2 and above.

\text{MCR2} is available in ARMv5 and above.

\text{MCRR} is available in ARMv6 and above, and E variants of ARMv5 excluding xP variants.

\text{MCRR2} is available in ARMv6 and above.
4.9.3 MRC, MRC2

Move to ARM register from coprocessor. Depending on the coprocessor, you might be able to specify various operations in addition.

**Syntax**

\[
\text{MRC}\{\text{cond}\} \coproc, \text{opcode1}, \text{Rd}, \text{CRn}, \text{CRm}\{, \text{opcode2}\} \\
\text{MRC2} \coproc, \text{opcode1}, \text{Rd}, \text{CRn}, \text{CRm}\{, \text{opcode2}\}
\]

where:

- \(\text{cond}\) is an optional condition code (see *Conditional execution* on page 4-6).
- \(\coproc\) is the name of the coprocessor the instruction is for. The standard name is \(p_n\), where \(n\) is an integer in the range 0-15.
- \(\text{opcode1}\) is a coprocessor-specific opcode.
- \(\text{Rd}\) is the ARM destination register. If \(\text{Rd}\) is \(r15\), only the flags field is affected.
- \(\text{CRn}, \text{CRm}\) are coprocessor registers.
- \(\text{opcode2}\) is an optional coprocessor-specific opcode.

**Usage**

The use of these instructions depends on the coprocessor. See the coprocessor documentation for details.

--- **Note** ---

\(\text{MRC2}\) is always unconditional.

**Architectures**

- \(\text{MRC}\) is available in ARMv2 and above.
- \(\text{MRC2}\) is available in ARMv5 and above.
4.9.4 MRRC and MRRC2

Move to two ARM registers from coprocessor. Depending on the coprocessor, you might be able to specify various operations in addition.

**Syntax**

\[
\text{MRRC}\{\text{cond}\} \coproc, \text{opcode}, \ Rd, \ Rn, \ CRm
\]

MRRC2 \coproc, \text{opcode}, \ Rd, \ Rn, \ CRm

where:

- \textit{cond} is an optional condition code (see *Conditional execution* on page 4-6).
- \textit{coproc} is the name of the coprocessor the instruction is for. The standard name is \textit{pn}, where \textit{n} is an integer in the range 0-15.
- \textit{opcode} is a coprocessor-specific opcode.
- \textit{Rd}, \textit{Rn} are ARM destination registers. Do not use r15 for \textit{Rd} or \textit{Rn}.
- \textit{CRm} is the coprocessor source register.

**Usage**

The use of these instructions depends on the coprocessor. See the coprocessor documentation for details.

---

**Note**

MRRC2 is always unconditional.

---

**Architectures**

MRRC is available in ARMv6 and above, and E variants of ARMv5 excluding xP variants.

MRRC2 is available in ARMv6 and above.
4.9.5 LDC, 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:

\[ \text{op}(\text{cond})\{\text{L}\} \coproc, \ CRd, \ [Rn] \]
\[ \text{op}(\text{cond})\{\text{L}\} \coproc, \ CRd, \ [Rn, \ #{-}\text{offset}]\{!\} \]
\[ \text{op}(\text{cond})\{\text{L}\} \coproc, \ CRd, \ [Rn], \ #{-}\text{offset} \]

where:

- \text{op} is either LDC or STC.
- \text{cond} is an optional condition code (see Conditional execution on page 4-6).
- \text{L} is an optional suffix specifying a long transfer.
- \coproc is the name of the coprocessor the instruction is for. The standard name is \( \text{p}n \), where \( n \) is an integer in the range 0-15.
- \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.
- \text{-} is an optional minus sign. If \text{-} is present, the offset is subtracted from \Rn. Otherwise, the offset is added to \Rn.
- \text{offset} is an expression evaluating to a multiple of 4, in the range 0-1020.
- \text{!} is an optional suffix. If \text{!} is present, the address including the offset is written back into \Rn.

Usage

The use of this instruction depends on the coprocessor. See the coprocessor documentation for details.
Architectures

LDC and STC are available in ARMv2 and above.
4.9.6  LDC2, 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:

\[ \text{op \ coproc, \ CRd, \ [Rn]} \]
\[ \text{op \ coproc, \ CRd, \ [Rn, \ #{-}offset]{!}} \]
\[ \text{op \ coproc, \ CRd, \ [Rn], \ #{-}offset} \]

where:

- \( \text{op} \) is either LDC2 or STC2.
- \( \text{coproc} \) is the name of the coprocessor the instruction is for. The standard name is \( pn \), where \( n \) is an integer in the range 0-15.
- \( \text{CRd} \) 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 \( Rn \).
- \( \text{offset} \) is an expression evaluating to a multiple of 4, in the range 0-1020.
- \( ! \) is an optional suffix. If ! is present, the address including the offset is written back into \( Rn \).

Usage

The use of this instruction depends on the coprocessor. See the coprocessor documentation for details.

--- Note ---

LDC2 and STC2 are always unconditional.
Architectures

LDC2 and STC2 are available in ARMv5 and above.
4.10 Miscellaneous ARM instructions

This section contains the following subsections:

- **SWI on page 4-112**
  Software interrupt.

- **MRS on page 4-113**
  Move the contents of the CPSR or SPSR to a general-purpose register.

- **MSR on page 4-114**
  Load specified fields of the CPSR or SPSR with an immediate constant, or from the contents of a general-purpose register.

- **CPS on page 4-115**
  Change processor state.

- **SETEND on page 4-117**
  Set the endianness bit in the CPSR.

- **BKPT on page 4-118**
  Breakpoint.

- **MAR, MRA on page 4-119**
  XScale coprocessor 0 instructions.
  Transfer between two general-purpose registers and a 40-bit internal accumulator.
4.10.1 SWI

Software interrupt.

Syntax

SWI{cond} immed_24

where:

cond is an optional condition code (see Conditional execution on page 4-6).

immed_24 is an expression evaluating to an integer in the range 0-2^{24}–1 (a 24-bit integer).

Usage

The SWI instruction causes a SWI exception. This means that the processor mode changes to Supervisor, the CPSR is saved to the Supervisor mode SPSR, and execution branches to the SWI vector (see the Handling Processor Exceptions chapter in RealView Compilation Tools v2.1 Developer Guide).

Condition flags

This instruction does not affect the flags.

Architectures

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

Example

SWI 0x123456
4.10.2 MRS

Move the contents of the CPSR or SPSR to a general-purpose register.

Syntax

MRS\{cond\} Rd, psr

where:

\textit{cond} is an optional condition code (see \textit{Conditional execution} on page 4-6).
\textit{Rd} is the destination register. \textit{Rd} must not be r15.
\textit{psr} is either CPSR or SPSR.

Usage

Use \texttt{MRS} in combination with \texttt{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.

\begin{verse}
\textbf{Caution}

You must not attempt to access the SPSR when the processor is in User or System mode. This is your responsibility. The assembler cannot warn you about this because it does not know in what processor mode the code will be executed.

If you do this, the result is UNPREDICTABLE.
\end{verse}

Condition flags

This instruction does not affect the flags.

Architectures

\texttt{MRS} is available in ARMv3 and above.

Example

MRS r3, SPSR
4.10.3 MSR

Load specified fields of the CPSR or SPSR with an immediate constant, or from the contents of a general-purpose register.

Syntax

\[
\text{MSR}\{\text{cond}\} \; <\text{psr}>_<\text{fields}>, \ #\text{immed}_8r \\
\text{MSR}\{\text{cond}\} \; <\text{psr}>_<\text{fields}>, \ Rm
\]

where:

- \(\text{cond}\) is an optional condition code (see Conditional execution on page 4-6).
- \(<\text{psr}>\) is either CPSR or SPSR.
- \(<\text{fields}>\) specifies the field or fields to be moved. \(<\text{fields}>\) 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].
- \(\text{immed}_8r\) 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 4-113.

Condition flags

This instruction updates the flags explicitly if the \(f\) field is specified.

Architectures

MSR is available in ARMv3 and above.

Example

\[
\text{MSR} \; \text{CPSR}_f, \ r5
\]
4.10.4 CPS

Change processor state.

Note

CPS has no effect in User mode.
CPS cannot be conditional.

Syntax

CPS\(\text{effect iflags\{, \#mode\}}\)  
CPS \#mode

where:

effect \quad \text{is one of the following:}
  \begin{align*}
  \text{IE} & \quad \text{Interrupt enable.} \\
  \text{ID} & \quad \text{Interrupt disable.}
  \end{align*}

iflags \quad \text{is a sequence of one or more of the following:}
  \begin{align*}
  a & \quad \text{Enables or disables imprecise aborts.} \\
  i & \quad \text{Enables or disables IRQ interrupts.} \\
  f & \quad \text{Enables or disables FIQ interrupts.}
  \end{align*}

mode \quad \text{specifies the number of the mode to change to.}

Operation

CPS makes the changes specified, without affecting any other bits in the CPSR.

Condition flags

CPS does not affect any condition flags.

Architectures

CPS is available in ARMv6 and above.
Examples

CPSIE if ; enable interrupts and fast interrupts
CPSID A ; disable imprecise aborts
CPSID ai, #17 ; disable imprecise aborts and interrupts, and enter FIQ mode
CPS #16 ; enter User mode

Incorrect example

CPSEQ #19 ; CPS cannot be conditional
4.10.5 SETEND

Set the endianness bit in the CPSR.

Note

SETEND cannot be conditional.

Syntax

SETEND specifier

where:

specifier is one of the following:

BE     Big endian.
LE     Little endian.

Usage

Use SETEND to access data of different endianness, for example to access several big-endian DMA-formatted data fields by an otherwise little-endian application.

Architectures

SETEND is available in ARMv6 and above.

Example

SETEND BE          ; Set the CPSR E bit for big-endian accesses
LDR r0, [r2, #header]
LDR r1, [r2, #CRC32]
SETEND le          ; Set the CPSR E bit for little-endian accesses for the
; rest of the application
4.10.6 BKPT

Breakpoint.

Syntax

BKPT  immed_16

where:

immed_16 is an expression evaluating to an integer in the range 0-65535 (a 16-bit integer). immed_16 is ignored by ARM hardware, but can be used by a debugger to store additional information about the breakpoint.

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.

Architectures

BKPT is available in ARMv5 and above.

Examples

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>BKPT</td>
<td>0xF02C</td>
</tr>
<tr>
<td>BKPT</td>
<td>640</td>
</tr>
</tbody>
</table>
4.10.7 MAR, MRA

XScale coprocessor 0 instructions.
Transfer between two general-purpose registers and a 40-bit internal accumulator.

**Syntax**

\[
\begin{align*}
\text{MAR}\{\text{cond}\} & \text{ Acc, RdLo, RdHi} \\
\text{MRA}\{\text{cond}\} & \text{ RdLo, RdHi, Acc}
\end{align*}
\]

where:

- **cond** is an optional condition code (see *Conditional execution* on page 4-6).
- **Acc** is the internal accumulator. The standard name is acc\_x, where x is an integer in the range 0-\(n\). The value of \(n\) depends on the processor. It is 0 for current processors.
- **RdLo, RdHi** are general-purpose registers.

**Usage**

The MAR instruction copies the contents of \(RdLo\) to \(\text{bits}[31:0]\) of \(Acc\), and the least significant byte of \(RdHi\) to \(\text{bits}[39:32]\) of \(Acc\).

The MRA instruction:
- copies \(\text{bits}[31:0]\) of \(Acc\) to \(RdLo\)
- copies \(\text{bits}[39:32]\) of \(Acc\) to \(RdHi\)
- sign extends the value by copying \(\text{bit}[39]\) of \(Acc\) to \(\text{bits}[31:8]\) of \(RdHi\).

**Architectures**

These instructions are only available in XScale processors.

**Examples**

\[
\begin{align*}
\text{MAR} & \text{ acc0, r0, r1} \\
\text{MRA} & \text{ r4, r5, acc0} \\
\text{MARNE} & \text{ acc0, r9, r2} \\
\text{MRACT} & \text{ r4, r8, acc0}
\end{align*}
\]
4.11 ARM pseudo-instructions

The ARM assembler supports a number of pseudo-instructions that are translated into the appropriate combination of ARM or Thumb instructions at assembly time.

The pseudo-instructions available in ARM state are described in the following sections:

- **ADR ARM pseudo-instruction** on page 4-121
  Load a program-relative or register-relative address (short range)

- **ADRL ARM pseudo-instruction** on page 4-122
  Load a program-relative or register-relative address into a register (medium range)

- **LDR ARM pseudo-instruction** on page 4-124
  Load a register with a 32-bit constant value or an address (unlimited range)

- **NOP ARM pseudo-instruction** on page 4-126
  Generate the preferred ARM no-operation code.
4.11.1 ADR ARM pseudo-instruction

Load a program-relative or register-relative address into a register.

**Syntax**

ADR{cond}: register, expr

where:

- **cond** is an optional condition code.
- **register** is the register to load.
- **expr** is a program-relative or register-relative expression that evaluates to:
  - a non word-aligned address within ±255 bytes
  - a word-aligned address within ±1020 bytes.

More distant addresses can be used if the alignment is 16 bytes or more.

The address can be either before or after the address of the instruction or the base register (see *Register-relative and program-relative expressions* on page 3-29).

--- **Note**

For program-relative expressions, the given range is relative to a point two words after the address of the current instruction.

---

**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 expr is program-relative, it must evaluate to an address in the same assembler area as the ADR pseudo-instruction, see *AREA* on page 7-56.

**Example**

```plaintext
start   MOV     r0,#10
ADR     r4,start    ; => SUB r4,pc,#0xc
```
4.11.2 ADRL ARM 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 Thumb instructions. Use it only in ARM code.

Syntax

ADR\{cond\}L register,expr

where:

- \(cond\) is an optional condition code.
- \(register\) is the register to load.
- \(expr\) is a program-relative or register-relative expression that evaluates to:
  - a non word-aligned address within 64KB
  - a word-aligned address within 256KB.

More distant addresses can be used if the alignment is 16 bytes or more. The address can be either before or after the address of the instruction or the base register (see Register-relative and program-relative expressions on page 3-29).

--- Note ---
For program-relative expressions, the given range is relative to a point two words after the address of the current instruction.

Usage

ADRL always assembles to two 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 ARM pseudo-instruction on page 4-124 for information on loading a wider range of addresses (see also Loading constants into registers on page 2-27).

ADRL produces position-independent code, because the address is program-relative or register-relative.
If `expr` is program-relative, it must evaluate to an address in the same assembler area as the `ADRL` pseudo-instruction, see `AREA` on page 7-56. Otherwise, it might be out of range after linking.

**Example**

```
start   MOV     r0,#10
        ADRL    r4,start + 60000     ; => ADD r4,pc,#0xe800
;    ADD r4,r4,#0x254
```
4.11.3 LDR ARM pseudo-instruction

Load a register with either:

- a 32-bit constant value
- an address.

--- Note ---
This section describes the LDR pseudo-instruction only. See ARM Memory access instructions on page 4-8 for information on the LDR instruction.

Also, see Loading with LDR Rd, =const on page 2-29, for information on loading constants with the LDR pseudo-instruction.

--- Syntax ---

```
LDR{cond} register,=[expr | label-expr]
```

where:

- `cond` is an optional condition code.
- `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.
Usage

The main purposes of the LDR pseudo-instruction are:

- To generate literal constants when an immediate value cannot be moved into a register because it is out of range of the MOV and MVN instructions.
- To load a program-relative or external address into a register. The address remains valid regardless of where the linker places the ELF section containing the LDR.

**Note**

An address loaded in this way is fixed at link time, so the code is not position-independent.

The offset from the PC to the value in the literal pool must be less than 4KB. You are responsible for ensuring that there is a literal pool within range. See LTORG on page 7-14 for more information.

See Loading constants into registers on page 2-27 for a more detailed explanation of how to use LDR, and for more information on MOV and MVN.

Example

```
LDR  r3,=0xff0  ; loads 0xff0 into r3
       =>  MOV r3,#0xff0
LDR  r1,=0xffff  ; loads 0xffff into r1
       =>  LDR r1,[pc,offset_to_litpool]
            ; ...
            ; litpool DCD 0xffff
LDR  r2,=place  ; loads the address of
                ; place into r2
       =>  LDR r2,[pc,offset_to_litpool]
            ; ...
            ; litpool DCD place
```
4.11.4 NOP ARM pseudo-instruction

NOP generates the preferred ARM no-operation code.

The following instruction might be used, but this is not guaranteed:

MOV r0, r0

Syntax

NOP

Usage

NOP cannot be used conditionally. Not executing a no-operation is the same as executing it, so conditional execution is not required.

ALU status flags are unaltered by NOP.
Chapter 5
Thumb Instruction Reference

This chapter describes the Thumb® instructions that are provided by the ARM® assembler and the inline assemblers in the ARM C and C++ compilers. It contains the following sections:

- Thumb memory access instructions on page 5-4
- Thumb arithmetic instructions on page 5-15
- Thumb general data processing instructions on page 5-22
- Thumb branch instructions on page 5-35
- Thumb miscellaneous instructions on page 5-42
- Thumb pseudo-instructions on page 5-47.

See Table 5-1 on page 5-2 to locate individual directives or pseudo-instructions.
### Table 5-1 Location of Thumb instructions and pseudo-instructions

<table>
<thead>
<tr>
<th>Instruction mnemonic</th>
<th>Brief description</th>
<th>Page</th>
<th>Architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC</td>
<td>Add with carry</td>
<td>page 5-21</td>
<td>4T</td>
</tr>
<tr>
<td>ADD</td>
<td>Add</td>
<td>page 5-15</td>
<td>4T</td>
</tr>
<tr>
<td>ADR</td>
<td>Load address (pseudo-instruction)</td>
<td>page 5-48</td>
<td></td>
</tr>
<tr>
<td>AND</td>
<td>Logical AND</td>
<td>page 5-23</td>
<td>4T</td>
</tr>
<tr>
<td>ASR</td>
<td>Arithmetic shift right</td>
<td>page 5-24</td>
<td>4T</td>
</tr>
<tr>
<td>B</td>
<td>Branch</td>
<td>page 5-36</td>
<td>4T</td>
</tr>
<tr>
<td>BIC</td>
<td>Bit clear</td>
<td>page 5-23</td>
<td>4T</td>
</tr>
<tr>
<td>BKPT</td>
<td>breakpoint</td>
<td>page 5-46</td>
<td>5T</td>
</tr>
<tr>
<td>BL</td>
<td>Branch with link</td>
<td>page 5-38</td>
<td>4T</td>
</tr>
<tr>
<td>BLX</td>
<td>Branch with link and exchange instruction sets</td>
<td>page 5-40</td>
<td>5T</td>
</tr>
<tr>
<td>BX</td>
<td>Branch and exchange instruction sets</td>
<td>page 5-39</td>
<td>4T</td>
</tr>
<tr>
<td>CMN, CMP</td>
<td>Compare negative, Compare</td>
<td>page 5-26</td>
<td>4T</td>
</tr>
<tr>
<td>CPS</td>
<td>Change processor state</td>
<td>page 5-44</td>
<td>6T</td>
</tr>
<tr>
<td>CPY</td>
<td>Copy</td>
<td>page 5-28</td>
<td>6T</td>
</tr>
<tr>
<td>EOR</td>
<td>Logical exclusive OR</td>
<td>page 5-23</td>
<td>4T</td>
</tr>
<tr>
<td>LDMIA</td>
<td>Load multiple registers, increment after</td>
<td>page 5-13</td>
<td>4T</td>
</tr>
<tr>
<td>LDR</td>
<td>Load register, immediate offset</td>
<td>page 5-5</td>
<td>4T</td>
</tr>
<tr>
<td>LDR</td>
<td>Load register, register offset</td>
<td>page 5-7</td>
<td>4T</td>
</tr>
<tr>
<td>LDR</td>
<td>Load register, pc or sp relative</td>
<td>page 5-9</td>
<td>4T</td>
</tr>
<tr>
<td>LDR</td>
<td>Load register (pseudo-instruction)</td>
<td>page 5-49</td>
<td></td>
</tr>
<tr>
<td>LSL, LSR</td>
<td>Logical shift left, Logical shift right</td>
<td>page 5-24</td>
<td>4T</td>
</tr>
<tr>
<td>MOV</td>
<td>Move</td>
<td>page 5-30</td>
<td>4T</td>
</tr>
<tr>
<td>MUL</td>
<td>Multiply</td>
<td>page 5-21</td>
<td>4T</td>
</tr>
<tr>
<td>MVN, NEG</td>
<td>Move NOT, Negate</td>
<td>page 5-30</td>
<td>4T</td>
</tr>
<tr>
<td>NOP</td>
<td>No operation (pseudo-instruction)</td>
<td>page 5-51</td>
<td></td>
</tr>
<tr>
<td>Instruction mnemonic</td>
<td>Brief description</td>
<td>Page</td>
<td>Architecturea</td>
</tr>
<tr>
<td>----------------------</td>
<td>-------------------------------------------------------</td>
<td>--------</td>
<td>----------------</td>
</tr>
<tr>
<td>ORR</td>
<td>Logical OR</td>
<td>page 5-23</td>
<td>4T</td>
</tr>
<tr>
<td>POP, PUSH</td>
<td>Pop registers from stack, Push registers onto stack</td>
<td>page 5-11</td>
<td>4T</td>
</tr>
<tr>
<td>REV, REV16, REVSH</td>
<td>Reverse byte order</td>
<td>page 5-32</td>
<td>6T</td>
</tr>
<tr>
<td>ROR</td>
<td>Rotate right</td>
<td>page 5-24</td>
<td>4T</td>
</tr>
<tr>
<td>SBC</td>
<td>Subtract with carry</td>
<td>page 5-21</td>
<td>4T</td>
</tr>
<tr>
<td>SETEND</td>
<td>Set endianness for data accesses</td>
<td>page 5-45</td>
<td>6T</td>
</tr>
<tr>
<td>SXT</td>
<td>Sign extend</td>
<td>page 5-33</td>
<td>6T</td>
</tr>
<tr>
<td>STMIA</td>
<td>Store multiple registers, increment after</td>
<td>page 5-13</td>
<td>4T</td>
</tr>
<tr>
<td>STR</td>
<td>Store register, immediate offset</td>
<td>page 5-5</td>
<td>4T</td>
</tr>
<tr>
<td>STR</td>
<td>Store register, register offset</td>
<td>page 5-7</td>
<td>4T</td>
</tr>
<tr>
<td>STR</td>
<td>Store register, pc or sp relative</td>
<td>page 5-9</td>
<td>4T</td>
</tr>
<tr>
<td>SUB</td>
<td>Subtract</td>
<td>page 5-15</td>
<td>4T</td>
</tr>
<tr>
<td>SWI</td>
<td>Software interrupt</td>
<td>page 5-43</td>
<td>4T</td>
</tr>
<tr>
<td>TST</td>
<td>Test bits</td>
<td>page 5-31</td>
<td>4T</td>
</tr>
<tr>
<td>UXT</td>
<td>Zero extend</td>
<td>page 5-33</td>
<td>6T</td>
</tr>
</tbody>
</table>

a. $\*$T : available in T variants of ARM architecture version $n$ and above
5.1 Thumb memory access instructions

This section contains the following subsections:

- **LDR and STR, immediate offset** on page 5-5
  Load Register and Store Register. Address in memory specified as an immediate offset from a value in a register.

- **LDR and STR, register offset** on page 5-7
  Load Register and Store Register. Address in memory specified as a register-based offset from a value in a register.

- **LDR and STR, pc or sp relative** on page 5-9
  Load Register and Store Register. Address in memory specified as an immediate offset from a value in the pc or the sp.

- **PUSH and POP** on page 5-11
  Push low registers, and optionally the LR, onto the stack.
  Pop low registers, and optionally the pc, off the stack.

- **LDMIA and STMIA** on page 5-13
  Load and store multiple registers.
5.1.1 LDR and STR, immediate offset

Load Register and Store Register. Address in memory specified as an immediate offset from a value in a register.

**Syntax**

\[ \text{op } Rd, [Rn, \#immed_{5x4}] \]

\[ \text{opH } Rd, [Rn, \#immed_{5x2}] \]

\[ \text{opB } Rd, [Rn, \#immed_{5x1}] \]

where:

- \( \text{op} \) is either:
  - \text{LDR} Load register
  - \text{STR} Store register.
- \( H \) is a parameter specifying an unsigned halfword transfer.
- \( B \) is a parameter specifying an unsigned byte transfer.
- \( Rd \) is the register to be loaded or stored. \( Rd \) must be in the range \( r0\text{-}r7 \).
- \( Rn \) is the register containing the base address. \( Rn \) must be in the range \( r0\text{-}r7 \).
- \( \text{immed}_{5xN} \) is the offset. It is an expression evaluating (at assembly time) to a multiple of \( N \) in the range 0-31\( N \).

**Usage**

STR instructions store a word, halfword, or byte to memory.

LDR instructions load a word, halfword, or byte from memory.

The address is found by adding the offset to the base address from \( Rn \).

Immediate offset halfword and byte loads are unsigned. The data is loaded into the least significant word or byte of \( Rd \), and the rest of \( Rd \) is filled with zeroes.
Address alignment for word and halfword transfers

The address must be divisible by 4 for word transfers, and by 2 for halfword transfers.

If your system has a system coprocessor (cp15), you can enable alignment checking. Non-aligned 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:

- A non-aligned load corrupts $Rd$
- A non-aligned save corrupts two or four bytes in memory. The corrupted location in memory is $\text{[address AND NOT 0x1]}$ for halfword saves, and $\text{[address AND NOT 0x3]}$ for word saves.

Architectures

These instructions are available in all T variants of the ARM architecture.

Examples

LDR $r3,[r5,#0]$
STRB $r0,[r3,#31]$
STRH $r7,[r3,#16]$
LDRH $r2,[r4,#label-{PC}]$

Incorrect examples

LDR $r13,[r5,#40]$ ; high registers not allowed
STRB $r0,[r3,#32]$ ; 32 is out of range for byte transfers
STRH $r7,[r3,#15]$ ; offsets for halfword transfers must be even
LDRH $r6,[r0,#-6]$ ; negative offsets not supported
5.1.2 LDR and STR, register offset

Load Register and Store Register. Address in memory specified as a register-based offset from a value in a register.

Syntax

\( \text{op} \ R_d, [R_n, \ R_m] \)

where:

- \( \text{op} \) is one of the following:
  - \( \text{LDR} \) Load register, four-byte word
  - \( \text{STR} \) Store register, four-byte word
  - \( \text{LDRH} \) Load register, two-byte unsigned halfword
  - \( \text{LDRSH} \) Load register, two-byte signed halfword
  - \( \text{STRH} \) Store register, two-byte halfword
  - \( \text{LDRB} \) Load register, unsigned byte
  - \( \text{LDRSB} \) Load register, signed byte
  - \( \text{STRB} \) Store register, byte.

Note

There is no distinction between signed and unsigned store instructions.

- \( R_d \) is the register to be loaded or stored. \( R_d \) must be in the range \( r0-r7 \).
- \( R_n \) is the register containing the base address. \( R_n \) must be in the range \( r0-r7 \).
- \( R_m \) is the register containing the offset. \( R_m \) must be in the range \( r0-r7 \).

Usage

- STR instructions store a word, halfword, or byte from \( R_d \) to memory.
- LDR instructions load a word, halfword, or byte from memory to \( R_d \).

The address is found by adding the offset to the base address from \( R_n \).

Register offset halfword and byte loads can be signed or unsigned. The data is loaded into the least significant word or byte of \( R_d \), and the rest of \( R_d \) is filled with zeroes for an unsigned load, or with copies of the sign bit for a signed load.
**Address alignment for word and halfword transfers**

The address must be divisible by 4 for word transfers, and by 2 for halfword transfers.

If your system has a system coprocessor (cp15), you can enable alignment checking. Non-aligned 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:

- A non-aligned load corrupts \( Rd \).
- A non-aligned save corrupts memory. The corrupted location in memory is the halfword at \([\text{address AND NOT 0x1}]\) for halfword saves, and the word at \([\text{address AND NOT b11}]\) for word saves.

**Architectures**

These instructions are available in all T variants of the ARM architecture.

**Examples**

- `LDR r2,[r1,r5]`
- `LDRSH r0,[r0,r6]`
- `STRB r1,[r7,r0]`

**Incorrect examples**

- `LDR r13,[r5,r3]` ; high registers not allowed
- `STRSH r7,[r3,r1]` ; no signed store instruction
5.1.3     LDR and STR, pc or sp relative

Load Register and Store Register. Address in memory specified as an immediate offset from a value in the pc or the sp.

--- Note ---
There is no pc-relative STR instruction.

Syntax

LDR     Rd, [pc, #immed_8x4]
LDR     Rd, label
LDR     Rd, [sp, #immed_8x4]
STR     Rd, [sp, #immed_8x4]

where:

Rd       is the register to be loaded or stored. Rd must be in the range r0 to r7.
immed_8x4 is the offset. It is an expression evaluating (at assembly time) to a multiple of 4 in the range 0 to 1020.
label    is a program-relative expression. See Register-relative and program-relative expressions on page 3-29 for more information. label must be after the current instruction, and within 1KB of it.

Usage

STR instructions store a word to memory.
LDR instructions load a word from memory.

The address is found by adding the offset to the base address from pc or sp. Bit[1] of the pc is ignored. This ensures that the address is word-aligned.

Address alignment for word and halfword transfers

The address must be a multiple of 4.

If your system has a system coprocessor (cp15), you can enable alignment checking. Non-aligned 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:

- A non-aligned load corrupts Rd.
- A non-aligned save corrupts four bytes in memory. The corrupted location in memory is [address AND NOT b11].

**Architectures**

These instructions are available in all T variants of the ARM architecture.

**Examples**

```
LDR r2,[pc,#1016]
LDR r5,localdata
LDR r0,[sp,#920]
STR r1,[sp,#20]
```

**Incorrect examples**

```
LDR r13,[pc,#8] ; Rd must be in range r0-r7
STR r7,[pc,#64] ; there is no pc-relative STR instruction
STRH r0,[sp,#16] ; there are no pc- or sp-relative halfword or byte transfers
LDR r2,[pc,#81] ; immediate must be a multiple of four
LDR r1,[pc,#-24] ; immediate must not be negative
STR r1,[sp,#1024] ; maximum immediate value is 1020
```
5.1.4 PUSH and POP

Push low registers, and optionally the lr, onto the stack.
Pop low registers, and optionally the pc, off the stack.

Syntax

PUSH \{reglist\}
PUSH \{reglist, lr\}
POP \{reglist\}
POP \{reglist, pc\}

where:

\text{reglist} \quad \text{is a comma-separated list of low registers or low-register ranges.}

\begin{itemize}
  \item \textbf{Note}
  \begin{itemize}
    \item The braces in the syntax description are part of the instruction format.
    \item They do not indicate that the register list is optional.
    \item There must be at least one register in the list.
  \end{itemize}
\end{itemize}

Usage

Thumb stacks are full, descending stacks. The stack grows downwards, and the sp points to the last entry on the stack.

Registers are stored on the stack in numerical order, with the lowest numbered register at the lowest address.
**POP \{reglist, pc\}**

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

In ARM architecture v5T and above:

- if bits[1:0] of the value loaded to the 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 the pc are ignored, so POP cannot be used to change state.

**Condition flags**

These instructions do not affect the flags.

**Architectures**

These instructions are available in all T variants of the ARM architecture.

**Examples**

```assembly
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
```

**Incorrect examples**

```assembly
PUSH    \{r3,r5-r8\}  ; high registers not allowed
PUSH    {}              ; must be at least one register in list
PUSH    \{r1-r4,pc\}   ; cannot push the pc
POP     \{r1-r4,LR\}   ; cannot pop the LR
```
5.1.5  **LDMIA and STMIA**

Load and store multiple registers.

**Syntax**

\[ op \, Rn!, \{reglist\} \]

where:

- **op** is either:
  - LDMIA  Load multiple, increment after
  - STMIA  Store multiple, increment after.
- **Rn** is the register containing the base address. \( Rn \) must be in the range \( r0-r7 \).
- **reglist** is a comma-separated list of low registers or low-register ranges.

\[ \text{Note} \]

The braces in the syntax description are part of the instruction format. They do not indicate that the register list is optional. There must be at least one register in the list.

**Usage**

Registers are loaded stored and in numerical order, with the lowest numbered register at the address initially in \( Rn \).

The value in \( Rn \) is incremented by 4 times the number of registers in \( reglist \).

If \( Rn \) is in \( reglist \):

- for an LDMIA instruction, the final value of \( Rn \) is the value loaded, not the incremented address
- for an STMIA instruction, the value stored for \( Rn \) is:
  - the initial value of \( Rn \) if \( Rn \) is the lowest-numbered register in \( reglist \)
  - unpredictable otherwise.

**Architectures**

These instructions are available in all T variants of the ARM architecture.
Examples

LDMIA r3!, {r0,r4}
LDMIA r5!, {r0-r7}
STMIA r0!, {r6,r7}
STMIA r3!, {r3,r5,r7}

Incorrect examples

LDMIA r3!,{r0,r9} ; high registers not allowed
STMIA r5!, {}     ; must be at least one register
                 ; in list
STMIA r5!,{r1-r6} ; value stored from r5 is unpredictable
5.2 Thumb arithmetic instructions

This section contains the following subsections:

- **ADD and SUB, low registers** on page 5-16
  Add and subtract.

- **ADD, high or low registers** on page 5-18
  Add values in registers, one or both of them in the range r8 to r15.

- **ADD and SUB, sp** on page 5-19
  Increment or decrement sp by an immediate constant.

- **ADD, pc or sp relative** on page 5-20
  Add an immediate constant to the value from sp or pc, and place the result into a low register.

- **ADC, SBC, and MUL** on page 5-21
  Add with carry, Subtract with carry, and Multiply.
5.2.1 ADD and SUB, low registers

Add and subtract. There are three forms of these instructions that operate on low registers. You can:

- add or subtract the contents of two registers, and place the result in a third register
- add a small integer to, or subtract it from, the value in a register, and place the result in a different register
- add a larger integer to, or subtract it from, the value in a register, and return the result to the same register.

Syntax

\[
\text{op } Rd, Rn, Rm \\
\text{op } Rd, Rn, \#expr3 \\
\text{op } Rd, \#expr8
\]

where:

- \textit{op} is either \texttt{ADD} or \texttt{SUB}.
- \textit{Rd} is the destination register. It is also used for the first operand in \textit{op} \textit{Rd}, \#\textit{expr8} instructions.
- \textit{Rn} is a register containing the first operand.
- \textit{Rm} is a register containing the second operand.
- \textit{expr3} is an expression evaluating (at assembly time) to an integer in the range –7 to +7.
- \textit{expr8} is an expression evaluating (at assembly time) to an integer in the range –255 to +255.

Usage

\textit{op } \textit{Rd}, \textit{Rn}, \textit{Rm} performs an \textit{Rn} + \textit{Rm} or an \textit{Rn} – \textit{Rm} operation, and places the result in \textit{Rd}.

\textit{op } \textit{Rd}, \textit{Rn}, \#\textit{expr3} performs an \textit{Rn} + \textit{expr3} or an \textit{Rn} – \textit{expr3} operation, and places the result in \textit{Rd}.

\textit{op } \textit{Rd}, \#\textit{expr8} performs an \textit{Rd} + \textit{expr8} or an \textit{Rd} – \textit{expr8} operation, and places the result in \textit{Rd}. 
Note

An ADD instruction with a negative value for expr3 or expr8 assembles to the corresponding SUB instruction with a positive constant. A SUB instruction with a negative value for expr3 or expr8 assembles to the corresponding ADD instruction with a positive constant.

Be aware of this when looking at disassembly listings.

Restrictions

Rd, Rn, and Rm must all be low registers (that is, in the range r0 to r7).

Condition flags

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

Architectures

These instructions are available in all T variants of the ARM architecture.

Examples

```
ADD r3,r1,r5
SUB r0,r4,#5
ADD r7,#201
ADD r1,vc+4 ; vc + 4 must evaluate at assembly time to
            ; an integer in the range -255 to +255
```

Incorrect examples

```
ADD r9,r2,r6 ; high registers not allowed
SUB r4,r5,#201 ; immediate value out of range
SUB r3,#-99  ; negative immediate values not allowed
```
5.2.2 ADD, high or low registers

Add values in registers, returning the result to the first operand register.

Syntax

ADD Rd, Rm

where:

Rd is the destination register. It is also used for the first operand.
Rm is a register containing the second operand.

Usage

This instruction adds the values in Rd and Rm, and places the result in Rd.

Note

An ADD Rd, Rm instruction where both Rd and Rm are low registers assembles to an ADD Rd, Rd, Rm instruction (see ADD and SUB, low registers on page 5-16). Be aware of this when looking at disassembly listings.

Condition flags

The N, Z, C, and V condition flags are:

• updated if both Rd and Rm are low registers
• unaffected otherwise.

Architectures

This instruction is available in all T variants of the ARM architecture.

Examples

ADD r12, r4
ADD r10, r11
ADD r8, r8
ADD r2, r4 ; equivalent to ADD r2, r2, r4. Does affect flags.
5.2.3 ADD and SUB, sp

Increment or decrement sp by an immediate constant.

Syntax

ADD sp, #expr
SUB sp, #expr

where:

expr is an expression that evaluates (at assembly time) to a multiple of 4 in the range –508 to +508.

Usage

This instruction adds the value of expr to the value from Rp, and places the result in Rd.

——— Note ————

An ADD instruction with a negative value for expr assembles to the corresponding SUB instruction with a positive constant. A SUB instruction with a negative value for expr assembles to the corresponding ADD instruction with a positive constant.

Be aware of this when looking at disassembly listings.

Condition flags

These instructions do not affect the flags.

Architectures

These instructions are available in all T variants of the ARM architecture.

Examples

ADD sp,#312
SUB sp,#96
SUB sp,#abc+8 ; abc + 8 must evaluate at assembly time to a multiple of 4 in the range –508 to +508
5.2.4 ADD, pc or sp relative

Add an immediate constant to the value from sp or pc, and place the result into a low register.

Syntax

ADD Rd, Rp, #expr

where:

Rd is the destination register. Rd must be in the range r0-r7.
Rp is either sp or pc.
expr is an expression that evaluates (at assembly time) to a multiple of 4 in the range 0-1020.

Usage

This instruction adds the value of expr to the value from Rp, and places the result in Rd.

Note

If Rp is the pc, the value used is:

(the address of the current instruction + 4) AND FFFFFFFC.

Condition flags

This instruction does not affect the flags.

Architectures

This instruction is available in all T variants of the ARM architecture.

Examples

ADD r6,sp,#64
ADD r2,pc,#980
ADD r0,pc,#lit-{PC} ; lit - {PC} must evaluate, at assembly
 ; time, to a multiple of 4 in the range
 ; 0 to 1020
5.2.5 ADC, SBC, and MUL

Add with carry, Subtract with carry, and Multiply.

Syntax

```
op Rd, Rm
```

where:

- `op` is one of ADC, SBC, or MUL.
- `Rd` is the destination register. It also contains the first operand.
- `Rm` is a register containing the second operand.

Usage

- **ADC** adds the values in `Rd` and `Rm`, together with the carry flag, and places the result in `Rd`. Use this to synthesize multiword addition.
- **SBC** subtracts the value in `Rm` from the value in `Rd`, taking account of the carry flag, and places the result in `Rd`. Use this to synthesize multiword subtraction.
- **MUL** multiplies the values in `Rd` and `Rm`, and places the result in `Rd`.

Restrictions

`Rd` and `Rm` must be low registers (that is, in the range r0 to r7).

Condition flags

- **ADC** and **SBC** update the N, Z, C, and V flags.
- **MUL** updates the N and Z flags.

In ARMv4 and earlier, **MUL** corrupts the C and V flags. In ARMv5 and later, **MUL** has no effect on the C and V flags.

Architectures

These instructions are available in all T variants of the ARM architecture.

Example

```
ADC r2, r4
```
5.3 Thumb general data processing instructions

This section contains the following subsections:

- **AND, ORR, EOR, and BIC** on page 5-23
  Bitwise logical operations.

- **ASR, LSL, LSR, and ROR** on page 5-24
  Shift and rotate operations.

- **CMP and CMN** on page 5-26
  Compare and Compare Negative.

- **MOV and CPY** on page 5-28
  Move and Copy.

- **MVN and NEG** on page 5-30
  Move NOT, and Negate.

- **TST** on page 5-31
  Test bits.

- **REV, REV16, and REVSH** on page 5-32
  Reverse bytes or halfwords.

- **SXT and UXT** on page 5-33
  Sign or zero extend.
5.3.1 AND, ORR, EOR, and BIC

Bitwise logical operations.

**Syntax**

\[ \text{op } Rd, \text{ Rm} \]

where:

- \( \text{op} \) is one of AND, ORR, EOR, or BIC.
- \( \text{Rd} \) is the destination register. It also contains the first operand. \( \text{Rd} \) must be in the range \( r0-r7 \).
- \( \text{Rm} \) is the register containing the second operand. \( \text{Rm} \) must be in the range \( r0-r7 \).

**Usage**

These instructions perform a bitwise logical operation on the contents of \( \text{Rd} \) and \( \text{Rm} \), and place the result in \( \text{Rd} \). The operations are as follows:

- the AND instruction performs a logical AND operation
- the ORR instruction performs a logical OR operation
- the EOR instruction performs a logical Exclusive OR operation
- the BIC instruction performs an \( \text{Rd} \) AND NOT \( \text{Rm} \) operation.

**Condition flags**

These instructions update the N and Z flags according to the result. The C and V flags are not affected.

**Architectures**

These instructions are available in all T variants of the ARM architecture.

**Example**

\[ \text{AND } r2, r4 \]
5.3.2 ASR, LSL, LSR, and ROR

Shift and rotate operations. These instructions can use a value contained in a register, or an immediate shift value.

**Syntax**

\[ \text{op} \ Rd, \ Rs \]

\[ \text{op} \ Rd, \ Rm, \ #\text{expr} \]

where:

- **op** is one of:
  - **ASR** - Arithmetic Shift Right. Register contents are treated as two’s complement signed integers. The sign bit is copied into vacated bits.
  - **LSL** - Logical Shift Left. Vacated bits are cleared.
  - **LSR** - Logical Shift Right. Vacated bits are cleared.
  - **ROR** - Rotate Right. Bits moved out of the right-hand end of the register are rotated back into the left-hand end.

**Note**

ROR can only be used with a register-controlled shift.

**Rd**

is the destination register. It is also the source register for register-controlled shifts. Rd must be in the range r0-r7.

**Rs**

is the register containing the shift value for register-controlled shifts. Rm must be in the range r0-r7.

**Rm**

is the source register for immediate shifts. Rm must be in the range r0-r7.

**expr**

is the immediate shift value. It is an expression evaluating (at assembly time) to an integer in the range:

- 0-31 if op is LSL
- 1-32 otherwise.
**Register-controlled shift**

These instructions take the value from \( Rd \), apply the shift to it, and place the result back into \( Rd \).

Only the least significant byte of \( Rs \) is used for the shift value.

For all these instructions except \( ROR \):
- if the shift is 32, \( Rd \) is cleared, and the last bit shifted out remains in the C flag
- if the shift is greater than 32, \( Rd \) and the C flag are cleared.

**Immediate shift**

These instructions take the value from \( Rm \), apply the shift to it, and place the result into \( Rd \).

**Condition flags**

These instructions update the N and Z flags according to the result. The V flag is not affected.

The C flag:
- is unaffected if the shift value is zero
- otherwise, contains the last bit shifted out of the source register.

**Architectures**

These instructions are available in all T variants of the ARM architecture.

**Examples**

```assembly
ASR r3, r5
LSR r0, r2, #6
LSR r5, r5, av ; av must evaluate, at assembly time, to an integer in the range 1-32.
LSL r0, r4, #0 ; same as MOV r0, r4 except that C and V flags are not affected
```

**Incorrect examples**

```assembly
ROR r2, r7, #3 ; ROR cannot use immediate shift value
LSL r9, r1 ; high registers not allowed
LSL r0, r7, #32 ; immediate shift out of range
ASR r0, r7, #0 ; immediate shift out of range
```
5.3.3 CMP and CMN

Compare and Compare Negative.

Syntax

CMP Rn, #expr  
CMP Rn, Rm  
CMN Rn, Rm

where:

Rn is the register containing the first operand.
expr is an expression that evaluates (at assembly time) to an integer in the range 0-255.
Rm is a register containing the second operand.

Usage

These instructions update the condition flags, but do not place a result in a register.

The CMP instruction subtracts the value of expr, or the value in Rm, from the value in Rn.
The CMN instruction adds the values in Rm and Rn.

Restrictions

In CMP Rn, #expr, and CMN instructions, Rn and Rm must be in the range r0 to r7.
In CMP Rn, Rm instructions, Rn and Rm can be any register r0 to r15.

Condition flags

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

Architectures

These instructions are available in all T variants of the architecture.
Examples

CMP r2,#255
CMP r7,r12 ; high register IS allowed with CMP Rn,Rm
CMN r1,r5

Incorrect examples

CMP r2,#508 ; immediate value out of range
CMP r9,#24 ; high register not allowed with #expr
CMN r0,r10 ; high register not allowed with CMN
5.3.4 MOV and CPY

Move and Copy.

Syntax

MOV Rd, #expr
MOV Rd, Rm
CPY Rd, Rm

where:

\textit{Rd} \quad \text{is the destination register.}

\textit{expr} \quad \text{is an expression that evaluates (at assembly time) to an integer in the range 0-255.}

\textit{Rm} \quad \text{is the source register.}

Usage

The \textit{MOV} instruction places \#expr, or the value from \textit{Rm}, in \textit{Rd}.

The \textit{CPY} instruction copies the value from \textit{Rm} to \textit{Rd}.

Restrictions

In \textit{MOV Rd, #expr}, MVN, and NEG instructions, \textit{Rd} and \textit{Rm} must be in the range r0 to r7.

In \textit{CPY} instructions, \textit{Rd} and \textit{Rm} can be any register r0 to r15.

In \textit{MOV Rd, Rm} instructions, \textit{Rd} and \textit{Rm} can be any register r0 to r15, but see \textit{Condition flags} on page 5-29.
Condition flags

The CPY instruction does not update any flags.

The MOV Rd,#expr instruction updates the N and Z flags. It has no effect on the C or V flags.

MOV Rd, Rm behaves as follows:

- If either Rd or Rm is a high register (r8-r15), the flags are unaffected. In ARMv6 and above, it is better to use CPY in these cases.
- If both Rd and Rm are low registers (r0-r7), the N and Z flags are updated, and C and V flags are cleared.

Note

You can use LSL, with a shift of zero, to move between low registers without clearing the C and V flags (see ASR, LSL, LSR, and ROR on page 5-24). This is not necessary in ARMv6 and above, you can use CPY instead.

Architectures

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

The CPY instruction is available in T variants of ARMv6 and above.

Examples

MOV r3,#0
MOV r0,r12 ; does not update flags

Incorrect examples

MOV r2,#256 ; immediate value out of range
MOV r8,#3 ; cannot move immediate to high register
5.3.5 MVN and NEG

Move NOT and Negate.

**Syntax**

MVN Rd, Rm  
NEG Rd, Rm

where:

- **Rd** is the destination register.
- **Rm** is the source register.

**Usage**

The **MVN** instruction takes the value in **Rm**, performs a bitwise logical NOT operation on the value, and places the result in **Rd**.

The **NEG** instruction takes the value in **Rm**, multiplies it by –1, and places the result in **Rd**.

**Condition flags**

The **MVN** instruction updates the **N** and **Z** flags. It has no effect on the **C** or **V** flags.

The **NEG** instruction updates the **N**, **Z**, **C**, and **V** flags.

**Architectures**

These instructions are available in all T variants of the ARM architecture.

**Examples**

- MVN r7,r1  
- NEG r2,r2

**Incorrect examples**

- MVN r8,r2 ; high registers not allowed with MVN or NEG  
- NEG r0,#3 ; immediate value not allowed with MVN or NEG
5.3.6 TST

Test bits.

Syntax
TST \texttt{Rn}, \texttt{Rm}
where:
\begin{itemize}
  \item \texttt{Rn} is the register containing the first operand.
  \item \texttt{Rm} is the register containing the second operand.
\end{itemize}

Usage
This instruction performs a bitwise logical AND operation on the values in \texttt{Rm} and \texttt{Rn}. It updates the condition flags, but does not place a result in a register.

Restrictions
\texttt{Rn} and \texttt{Rm} must be in the range \texttt{r0}-\texttt{r7}.

Condition flags
TST updates the N and Z flags according to the result. The C and V flags are unaffected.

Architectures
TST is available in all T variants of the ARM architecture.

Example
\begin{itemize}
  \item TST \texttt{r2}, \texttt{r4}
\end{itemize}
5.3.7 REV, REV16, and REVSH

Reverse byte order in a word or halfword. Reverse bytes in a halfword and sign extend.

**Syntax**

\[ op \ Rd, \ Rm \]

where:

- \( op \) is any one of the following:
  - REV: Reverses byte order in a word.
  - REV16: Reverses byte order in each halfword of \( Rm \).
  - REVSH: Reverses byte order in the bottom halfword of \( Rm \), and sign extends to 32 bits.

- \( Rd \) is the destination register. \( Rd \) must be in the range \( r0-r7 \).

- \( Rm \) is the register containing the second operand. \( Rm \) must be in the range \( r0-r7 \).

Do not use \( r15 \) for \( Rd \) or \( Rm \).

**Condition flags**

These instructions do not affect the flags.

**Architectures**

These instructions are available in T variants of ARMv6 and above.

**Examples**

- REV  \( r3, r7 \)
- REV16  \( r0, r0 \)
- REVSH  \( r0, r5 \)
5.3.8 SXT and UXT

Signed and unsigned data unpacking instructions.

These instructions do any one of the following:
- sign or zero extend an 8-bit value to 32 bits
- sign or zero extend a 16-bit value to 32 bits.

Syntax

\[ \text{op } \text{Rd}, \text{Rm} \]

where:

- \( \text{op} \) is one of:
  - \( \text{SXTB} \) sign extend an 8-bit value to a 32-bit value.
  - \( \text{SXTH} \) sign extend a 16-bit value to a 32-bit value.
  - \( \text{UXTB} \) zero extend an 8-bit value to a 32-bit value.
  - \( \text{UXTH} \) zero extend a 16-bit value to a 32-bit value.

- \( \text{Rd} \) is the destination register. Must be in the range r0 to r7.
- \( \text{Rm} \) is the register holding the operand. Must be in the range r0 to r7.

Condition flags

These instructions do not affect any flags.

Architectures

These instructions are available in T variants of ARMv6 and above.

Examples

- \( \text{SXTB } r3, r1 \)
- \( \text{UXTH } r5, r0 \)
Synonyms

There are synonyms for these instructions. The use of these pseudonyms is deprecated. The synonyms are shown in Table 5-2.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Synonym</th>
</tr>
</thead>
<tbody>
<tr>
<td>SXTH</td>
<td>SEXT16</td>
</tr>
<tr>
<td>SXTB</td>
<td>SEXT8</td>
</tr>
<tr>
<td>UXTH</td>
<td>UEXT16</td>
</tr>
<tr>
<td>UXTB</td>
<td>UEXT8</td>
</tr>
</tbody>
</table>
5.4 Thumb branch instructions

This section contains the following subsections:

- **B** on page 5-36
  Branch.

- **BL** on page 5-38
  Branch with Link.

- **BX** on page 5-39
  Branch and exchange instruction set.

- **BLX** on page 5-40
  Branch with Link and exchange instruction set.
5.4.1 B

Branch. This is the only instruction in the Thumb instruction set that can be conditional.

Syntax

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

where:

- \text{cond}\) is an optional condition code (see Table 5-3 on page 5-37).
- \text{label}\) is a program-relative expression. This is usually a label within the same piece of code. See Register-relative and program-relative expressions on page 3-29 for more information.

\text{label}\) must be within:

- \(-252\) to \(+258\) bytes of the current instruction, if \text{cond}\) is used
- \(\pm 2\text{KB}\) if the instruction is unconditional.

Usage

The B instruction causes a branch to \text{label}\), if \text{cond}\) is satisfied, or if \text{cond}\) is not used.

\begin{itemize}
  \item \textbf{Note}\: \text{label}\) must be within the specified limits. The ARM linker cannot add code to generate longer branches.
\end{itemize}

Architectures

This instruction is available in all T variants of the ARM architecture.

Examples

\begin{verbatim}
  B dloop
  BEQ sectB
\end{verbatim}
### Table 5-3  Condition codes for Thumb B instruction

<table>
<thead>
<tr>
<th>Suffix</th>
<th>Flags</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>EQ</td>
<td>Z set</td>
<td>Equal</td>
</tr>
<tr>
<td>NE</td>
<td>Z clear</td>
<td>Not equal</td>
</tr>
<tr>
<td>CS/HS</td>
<td>C set</td>
<td>Higher or same (unsigned &gt;= )</td>
</tr>
<tr>
<td>CC/L0</td>
<td>C clear</td>
<td>Lower (unsigned &lt; )</td>
</tr>
<tr>
<td>MI</td>
<td>N set</td>
<td>Negative</td>
</tr>
<tr>
<td>PL</td>
<td>N clear</td>
<td>Positive or zero</td>
</tr>
<tr>
<td>VS</td>
<td>V set</td>
<td>Overflow</td>
</tr>
<tr>
<td>VC</td>
<td>V clear</td>
<td>No overflow</td>
</tr>
<tr>
<td>HI</td>
<td>C set and Z clear</td>
<td>Higher (unsigned &lt;= )</td>
</tr>
<tr>
<td>LS</td>
<td>C clear or Z set</td>
<td>Lower or same (unsigned &lt;= )</td>
</tr>
<tr>
<td>GE</td>
<td>N and V the same</td>
<td>Signed &gt;=</td>
</tr>
<tr>
<td>LT</td>
<td>N and V different</td>
<td>Signed &lt;</td>
</tr>
<tr>
<td>GT</td>
<td>Z clear, and N and V the same</td>
<td>Signed &gt;</td>
</tr>
<tr>
<td>LE</td>
<td>Z set, or N and V different</td>
<td>Signed &lt;=</td>
</tr>
</tbody>
</table>
5.4.2  BL

Long branch with Link.

Syntax

BL  label

where:

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

Usage

BL copies the address of the next instruction into r14 (lr, the link register), and causes a branch to label.

The machine-level instruction cannot branch to an address outside ±4Mb of the current instruction. When necessary, the ARM linker inserts code (a veneer) to allow longer branches (see The ARM linker chapter in RealView Compilation Tools v2.1 Linker and Utilities Guide).

Architectures

BL is available in all T variants of the ARM architecture.

Example

    BL  extract
5.4.3 BX

Branch, and optionally exchange instruction set.

Syntax

BX Rm

where:

Rm is an ARM register containing the address to branch to.
Bit 0 of Rm is not used as part of the address.
If bit 0 of Rm is clear:
• bit 1 must also be clear
• the instruction clears the T flag in the CPSR, and the code at the
  destination is interpreted as ARM code.

Usage

BX causes a branch to the address held in Rm, and changes instruction set to Thumb if bit 0 of Rm is set.

Architectures

BX is available in all T variants of the ARM architecture.

Examples

BX r5
5.4.4 BLX

Branch with Link, and optionally exchange instruction set.

**Syntax**

\[ BLX \ Rm \]

\[ BLX \ label \]

where:

\( Rm \) is an ARM register containing the address to branch to.

Bit 0 of \( Rm \) is not used as part of the address. If bit 0 of \( Rm \) is clear:

- Bit 1 must also be clear.
- The instruction clears the T flag in the CPSR. Code at the destination is interpreted as ARM code.

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

BLX \( label \) always causes a change to ARM state.

**Usage**

The BLX instruction:

- copies the address of the next instruction into r14 (lr, the link register)
- causes a branch to \( label \), or to the address held in \( Rm \)
- changes instruction set to ARM if either:
  - bit 0 of \( Rm \) is clear
  - the BLX \( label \) form is used.

The machine-level instruction cannot branch to an address outside ±4Mb of the current instruction. When necessary, the ARM linker inserts code (a veneer) to allow longer branches (see *The ARM linker* chapter in *RealView Compilation Tools v2.1 Linker and Utilities Guide*).

**Architectures**

BLX is available in all T variants of ARMv5 and above.
Examples

BLX r6
BLX arnsub
5.5 Thumb miscellaneous instructions

This section contains the following subsections:

- **SWI** on page 5-43
  Software interrupt.

- **CPS** on page 5-44
  Change processor state.

- **SETEND** on page 5-45
  Set the endianness bit in the CPSR.

- **BKPT** on page 5-46
  Breakpoint.
5.5.1 SWI

Software interrupt.

Syntax

```c
SWI immed_8
```

where:

```c
immed_8
```

is a numeric expression evaluating to an integer in the range 0-255.

Usage

The SWI instruction causes a SWI exception. This means that the processor state changes to ARM, the processor mode changes to Supervisor, the CPSR is saved to the Supervisor Mode SPSR, and execution branches to the SWI vector (see the Handling Processor Exceptions chapter in RealView Compilation Tools v2.1 Developer Guide).

`immed_8` is ignored by the processor. However, it is present in bits[7:0] of the instruction opcode. It can be retrieved by the exception handler to determine what service is being requested.

Condition flags

SWI does not affect the flags.

Architectures

SWI is available in all T variants of the ARM architecture.

Example

```
SWI 12
```
5.5.2 CPS

Change processor state.

Syntax

CPS\textit{effect} \textit{iflags}

where:

\textit{effect} is one of the following:

- \textit{IE} Interrupt enable.
- \textit{ID} Interrupt disable.

\textit{iflags} is a sequence of one or more of the following:

- \textit{a} Enables or disables imprecise aborts.
- \textit{i} Enables or disables IRQ interrupts.
- \textit{f} Enables or disables FIQ interrupts.

Operation

CPS makes the changes specified, without affecting any other bits in the CPSR.

Condition flags

CPS does not affect any condition flags.

Architectures

CPS is available in T variants of ARMv6 and above.

Examples

- CPSIE if \text{; enable interrupts and fast interrupts}
- CPSID a \text{; disable imprecise aborts}

incorrect example

- CPSID ai, \#17 \text{; cannot change mode with Thumb CPS}
5.5.3 SETEND

Set the endianness bit in the CPSR.

Syntax

SETEND specifier

where:

specifier is one of the following:

   BE    Big endian.
   LE    Little endian.

Usage

Use SETEND to access data of different endianness, for example to access several big-endian DMA-formatted data fields by an otherwise little-endian application.

Architectures

SETEND is available in T variants of ARMv6 and above.

Example

    SETEND BE          ; Set the CPSR E bit for big-endian accesses
    LDR     r0, [r2, #header]
    LDR     r1, [r2, #CRC32]
    SETEND LE          ; Set the CPSR E bit for little-endian accesses for the
                       ; rest of the application
5.5.4  BKPT

Breakpoint.

Syntax

BKPT \texttt{immed\_8}

where:

\texttt{immed\_8} is an expression evaluating to an integer in the range 0-255.

Usage

BKPT causes the processor to enter Debug mode. Debug tools can use this to investigate system state when the instruction at a particular address is reached.

\texttt{immed\_8} is ignored by the processor. However, it is present in bits[7:0] of the instruction opcode. It can be used by a debugger to store additional information about the breakpoint.

Architectures

BKPT is available in T variants of ARMv5 and above.

Examples

\begin{verbatim}
BKPT 67
BKPT 2_10110
\end{verbatim}
5.6 Thumb pseudo-instructions

The ARM assembler supports a number of Thumb pseudo-instructions that are translated into the appropriate Thumb instructions at assembly time.

The pseudo-instructions that are available in Thumb state are in the following sections:

- *ADR Thumb pseudo-instruction* on page 5-48
- *LDR Thumb pseudo-instruction* on page 5-49
- *NOP Thumb pseudo-instruction* on page 5-51.
5.6.1 ADR Thumb pseudo-instruction

The ADR pseudo-instruction loads a program-relative address into a register.

**Syntax**

ADR register, expr

where:

- `register` is the register to load.
- `expr` is a program-relative expression. The offset must be positive and less than 1KB. `expr` must be defined locally, it cannot be imported.

**Usage**

In Thumb state, ADR can generate word-aligned addresses only. Use the ALIGN directive to ensure that `expr` is aligned (see ALIGN on page 7-54).

`expr` must evaluate to an address in the same code section as the ADR pseudo-instruction. There is no guarantee that the address will be within range after linking if it resides in another ELF section.

**Example**

ADR r4,txampl ; => ADD r4,pc,#nn
 ; code
ALIGN
txampl DCW 0,0,0,0
5.6.2 LDR Thumb pseudo-instruction

The LDR pseudo-instruction loads a low register with either:

- a 32-bit constant value
- an address.

--- Note ---

This section describes the LDR pseudo-instruction only. See Thumb memory access instructions on page 5-4 for information on the LDR instruction.

Syntax

LDR register, =[expr | label-exp]

where:

- register is the register to be loaded. LDR can access the low registers (r0-r7) only.
- expr evaluates to a numeric constant:
  - if the value of expr is within range of a MOV instruction, the assembler generates the instruction
  - if the value of expr is not within range of a MOV 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-exp is a program-relative or external expression. The assembler places the value of label-exp in a literal pool and generates a program-relative LDR instruction that loads the value from the literal pool.

If label-exp 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 ensures that the correct address is generated at link time.

The offset from the pc to the value in the literal pool must be positive and less than 1KB. You are responsible for ensuring that there is a literal pool within range. See LTORG on page 7-14 for more information.

Usage

The LDR pseudo-instruction is used for two main purposes:

- To generate literal constants when an immediate value cannot be moved into a register because it is out of range of the MOV instruction.
• To load a program-relative or external address into a register. The address remains valid regardless of where the linker places the ELF section containing the LDR.

Example

```
LDR r1, =0xfff ; loads 0xfff into r1
LDR r2, =labelname ; loads the address of labelname into r2
```
5.6.3 NOP Thumb pseudo-instruction

NOP generates the preferred Thumb no-operation instruction.

The following instruction might be used, but this is not guaranteed:

    MOV r8, r8

Syntax

The syntax for NOP is:

    NOP

Condition flags

ALU status flags are unaltered by NOP.
Chapter 6
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 6-4
- *Floating-point registers* on page 6-5
- *Vector and scalar operations* on page 6-7
- *VFP and condition codes* on page 6-8
- *VFP system registers* on page 6-10
- *Flush-to-zero mode* on page 6-13
- *VFP instructions* on page 6-15
- *VFP pseudo-instruction* on page 6-36
- *VFP directives and vector notation* on page 6-38.

See Table 6-1 on page 6-2 for locations of descriptions of individual instructions.
## Table 6-1 Location of descriptions of VFP instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Brief description</th>
<th>Page</th>
<th>Operation</th>
<th>Architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>FABS</td>
<td>Absolute value</td>
<td>page 6-16</td>
<td>Vector</td>
<td>All</td>
</tr>
<tr>
<td>FADD</td>
<td>Add</td>
<td>page 6-17</td>
<td>Vector</td>
<td>All</td>
</tr>
<tr>
<td>FCMP</td>
<td>Compare</td>
<td>page 6-18</td>
<td>Scalar</td>
<td>All</td>
</tr>
<tr>
<td>FCPY</td>
<td>Copy</td>
<td>page 6-16</td>
<td>Vector</td>
<td>All</td>
</tr>
<tr>
<td>FCVTDS</td>
<td>Convert single-precision to double-precision</td>
<td>page 6-19</td>
<td>Scalar</td>
<td>All</td>
</tr>
<tr>
<td>FCVTSD</td>
<td>Convert double-precision to single-precision</td>
<td>page 6-20</td>
<td>Scalar</td>
<td>All</td>
</tr>
<tr>
<td>FDIV</td>
<td>Divide</td>
<td>page 6-21</td>
<td>Vector</td>
<td>All</td>
</tr>
<tr>
<td>FLD</td>
<td>Load (see also <em>FLD pseudo-instruction</em> on page 6-36)</td>
<td>page 6-22</td>
<td>Scalar</td>
<td>All</td>
</tr>
<tr>
<td>FLDM</td>
<td>Load multiple</td>
<td>page 6-24</td>
<td>-</td>
<td>All</td>
</tr>
<tr>
<td>FMAC</td>
<td>Multiply-accumulate</td>
<td>page 6-26</td>
<td>Scalar</td>
<td>All</td>
</tr>
<tr>
<td>FMDRR, FMDLR</td>
<td>Transfer from one ARM® register to half of double-precision</td>
<td>page 6-28</td>
<td>Scalar</td>
<td>All</td>
</tr>
<tr>
<td>FMDRR</td>
<td>Transfer from two ARM registers to double-precision</td>
<td>page 6-27</td>
<td>Scalar</td>
<td>VFPv2</td>
</tr>
<tr>
<td>FMRDH, FMRDL</td>
<td>Transfer from half of double-precision to ARM register</td>
<td>page 6-28</td>
<td>Scalar</td>
<td>All</td>
</tr>
<tr>
<td>FMRD</td>
<td>Transfer from double-precision to two ARM registers</td>
<td>page 6-27</td>
<td>Scalar</td>
<td>VFPv2</td>
</tr>
<tr>
<td>FMRSS</td>
<td>Transfer between two ARM registers and two single-precision</td>
<td>page 6-30</td>
<td>Scalar</td>
<td>VFPv2</td>
</tr>
<tr>
<td>FMRX</td>
<td>Transfer from single-precision to ARM register</td>
<td>page 6-29</td>
<td>Scalar</td>
<td>All</td>
</tr>
<tr>
<td>FMSC</td>
<td>Multiply-subtract</td>
<td>page 6-26</td>
<td>Vector</td>
<td>All</td>
</tr>
<tr>
<td>FMXR</td>
<td>Transfer from VFP system register to ARM register</td>
<td>page 6-31</td>
<td>-</td>
<td>All</td>
</tr>
<tr>
<td>FMSTAT</td>
<td>Transfer VFP status flags to ARM CPSR status flags</td>
<td>page 6-31</td>
<td>-</td>
<td>All</td>
</tr>
<tr>
<td>FMUL</td>
<td>Multiply</td>
<td>page 6-32</td>
<td>Vector</td>
<td>All</td>
</tr>
<tr>
<td>FMOR</td>
<td>Transfer from ARM register to VFP system register</td>
<td>page 6-31</td>
<td>-</td>
<td>All</td>
</tr>
</tbody>
</table>
Table 6-1 Location of descriptions of VFP instructions (continued)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Brief description</th>
<th>Page</th>
<th>Operation</th>
<th>Architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>FNEG</td>
<td>Negate</td>
<td>page 6-16</td>
<td>Vector</td>
<td>All</td>
</tr>
<tr>
<td>FNMAC</td>
<td>Negate-multiply-accumulate</td>
<td>page 6-26</td>
<td>Vector</td>
<td>All</td>
</tr>
<tr>
<td>FNMSC</td>
<td>Negate-multiply-subtract</td>
<td>page 6-26</td>
<td>Vector</td>
<td>All</td>
</tr>
<tr>
<td>FNMUL</td>
<td>Negate-multiply</td>
<td>page 6-32</td>
<td>Vector</td>
<td>All</td>
</tr>
<tr>
<td>FSIT0</td>
<td>Convert signed integer to floating-point</td>
<td>page 6-33</td>
<td>Scalar</td>
<td>All</td>
</tr>
<tr>
<td>FSQRT</td>
<td>Square Root</td>
<td>page 6-34</td>
<td>Vector</td>
<td>All</td>
</tr>
<tr>
<td>FST</td>
<td>Store</td>
<td>page 6-22</td>
<td>Scalar</td>
<td>All</td>
</tr>
<tr>
<td>FSTM</td>
<td>Store multiple</td>
<td>page 6-24</td>
<td>-</td>
<td>All</td>
</tr>
<tr>
<td>FSUB</td>
<td>Subtract</td>
<td>page 6-17</td>
<td>Vector</td>
<td>All</td>
</tr>
<tr>
<td>FTOSI, FT0UI</td>
<td>Convert floating-point to signed or unsigned integer</td>
<td>page 6-35</td>
<td>Scalar</td>
<td>All</td>
</tr>
<tr>
<td>FUIT0</td>
<td>Convert unsigned integer to floating-point</td>
<td>page 6-33</td>
<td>Scalar</td>
<td>All</td>
</tr>
</tbody>
</table>
6.1 The vector floating-point coprocessor

The Vector Floating-Point (VFP) coprocessor, together with associated support code, provides single-precision and double-precision floating-point arithmetic, as defined by ANSI/IEEE Std. 754-1985 IEEE Standard for Binary Floating-Point Arithmetic. This document is referred to as the IEEE 754 standard in this chapter. There is a summary of the standard in the floating-point chapter in RealView Compilation Tools v2.1 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, allowing single-instruction, multiple-data (SIMD) parallelism. In addition, the floating-point load and store instructions have multiple register forms, allowing vectors to be transferred to and from memory efficiently.

For more details of the vector floating-point coprocessor, see ARM Architecture Reference Manual.

6.1.1 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 allow you to transfer two 32-bit words between ARM registers and VFP registers with one instruction.
6.2 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. d0 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.

6.2.1 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 6-1 for further clarification.

![Figure 6-1 VFP register banks](image-url)
6.2.2 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 \textit{LEN} bits in the \textit{FPSCR} (see \textit{FPSCR, the floating-point status and control register} on page 6-10).

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 \textit{STRIDE} bits in the \textit{FPSCR} (see \textit{FPSCR, the floating-point status and control register} on page 6-10). 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. Allowing 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.
6.3 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 6-10).

When LEN is 1 all operations are scalar.

6.3.1 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 6-5).

The behavior of instructions of the following general forms:

```plaintext
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`. 
6.4 VFP and condition codes

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.

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 6-10).

**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 6-31).

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 needed to allow for unordered results.

The meanings of the condition code mnemonics are shown in Table 6-2.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Meaning after ARM data processing instruction</th>
<th>Meaning after VFP FCMP instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>EQ</td>
<td>Equal</td>
<td>Equal</td>
</tr>
<tr>
<td>NE</td>
<td>Not equal</td>
<td>Not equal, or unordered</td>
</tr>
<tr>
<td>CS / HS</td>
<td>Carry set / Unsigned higher or same</td>
<td>Greater than or equal, or unordered</td>
</tr>
<tr>
<td>CC / LO</td>
<td>Carry clear / Unsigned lower</td>
<td>Less than</td>
</tr>
<tr>
<td>MI</td>
<td>Negative</td>
<td>Less than</td>
</tr>
<tr>
<td>PL</td>
<td>Positive or zero</td>
<td>Greater than or equal, or unordered</td>
</tr>
<tr>
<td>VS</td>
<td>Overflow</td>
<td>Unordered (at least one NaN operand)</td>
</tr>
<tr>
<td>VC</td>
<td>No overflow</td>
<td>Not unordered</td>
</tr>
</tbody>
</table>
Table 6-2 Condition codes (continued)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Meaning after ARM data processing instruction</th>
<th>Meaning after VFP FCMP instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>HI</td>
<td>Unsigned higher</td>
<td>Greater than, or unordered</td>
</tr>
<tr>
<td>LS</td>
<td>Unsigned lower or same</td>
<td>Less than or equal</td>
</tr>
<tr>
<td>GE</td>
<td>Signed greater than or equal</td>
<td>Greater than or equal</td>
</tr>
<tr>
<td>LT</td>
<td>Signed less than</td>
<td>Less than, or unordered</td>
</tr>
<tr>
<td>GT</td>
<td>Signed greater than</td>
<td>Greater than</td>
</tr>
<tr>
<td>LE</td>
<td>Signed less than or equal</td>
<td>Less than or equal, or unordered</td>
</tr>
<tr>
<td>AL</td>
<td>Always (normally omitted)</td>
<td>Always (normally omitted)</td>
</tr>
</tbody>
</table>

--- Note ---

The type of the instruction that last updated the flags in the CPSR determines the meaning of condition codes.
6.5 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 6-12
- **FPSID, the floating-point system ID register** on page 6-12.

A particular implementation of VFP can have additional registers (see the technical reference manual for the VFP coprocessor you are using).

6.5.1 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 6-8).

- **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 allow greater performance, depending on your hardware and software, at the expense of loss of range (see **Flush-to-zero mode** on page 6-13).

  **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 6-6). Stride is controlled as follows:
  - 0b00 stride = 1
  - 0b11 stride = 2.
bits[18:16] LEN is the number of registers used by each vector (see Vectors on page 6-6). 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 6-12).
6.5.2 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*).

6.5.3 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.

6.5.4 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 6-31.*
6.6 **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.

### 6.6.1 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.

### 6.6.2 The effects of using flush-to-zero mode

With certain exceptions (see *Operations not affected by flush-to-zero mode* on page 6-14), 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.
6.6.3 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 \textit{\texttt{FABS, FCPY, and FNEG}} on page 6-16)
- Load and store (see \textit{\texttt{FLD and FST}} on page 6-22)
- Load multiple and store multiple (see \textit{\texttt{FLDM and FSTM}} on page 6-24)
- Transfer between floating-point registers and ARM general-purpose registers
  (see \textit{\texttt{FMDRR and FMRRD}} on page 6-27 and \textit{\texttt{FMRRS and FMSRR}} on page 6-30).
6.7 VFP instructions

This section contains the following subsections:

- **FABS, FCPY, and FNEG** on page 6-16
  Floating-point absolute value, copy, and negate.
- **FADD and FSUB** on page 6-17
  Floating-point add and subtract.
- **FCMP** on page 6-18
  Floating-point compare.
- **FCVTDS** on page 6-19
  Convert single-precision floating-point to double-precision.
- **FCVTSD** on page 6-20
  Convert double-precision floating-point to single-precision.
- **FDIV** on page 6-21
  Floating-point divide.
- **FLD and FST** on page 6-22
  Floating-point load and store.
- **FLDM and FSTM** on page 6-24
  Floating-point load multiple and store multiple.
- **FMAC, FNMAC, FMSC, and FNMSC** on page 6-26
  Floating-point multiply accumulate instructions.
- **FMDRR and FMRRD** on page 6-27
  Transfer contents between ARM registers and a double-precision floating-point register.
- **FMRRS and FMSRR** on page 6-30
  Transfer contents between a single-precision floating-point register and an ARM register.
- **FMRX, FMXR, and FMSTAT** on page 6-31
  Transfer contents between an ARM register and a VFP system register.
- **FMUL and FNMUL** on page 6-32
  Floating-point multiply and negate-multiply.
- **FSITO and FUITO** on page 6-33
  Convert signed integer to floating-point and unsigned integer to floating-point.
- **FSQRT** on page 6-34
  Floating-point square root.
- **FTOSI and FTOUI** on page 6-35
  Convert floating-point to signed integer and floating-point to unsigned integer.
6.7.1  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 6-7).

Syntax

\(<\text{\textit{op}}><\text{\textit{precision}}>{\text{\textit{cond}}} Fd, Fm\)

where:

\(\text{\textit{op}}\) must be one of FCPY, FABS, or FNEG.

\(\text{\textit{precision}}\) must be either \texttt{S} for single-precision, or \texttt{D} for double-precision.

\textit{cond} is an optional condition code (see VFP and condition codes on page 6-8).

\(Fd\) 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 \(\text{\textit{precision}}\).

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
6.7.2 FADD and FSUB

Floating-point add and subtract.

FADD and FSUB can be scalar, vector, or mixed (see Vector and scalar operations on page 6-7).

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 6-8).
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
6.7.3 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 signalling 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 6-8).

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 6-8).

Exceptions

FCMP instructions can produce Invalid Operation exceptions.

Examples

FCMPS s3, s0
FCMPEDNE d5, d13
FCMPZSEQ s2
6.7.4 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 6-8).
- `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
```
6.7.5 FCVTSD

Convert double-precision floating-point to single-precision.

FCVTSD is always scalar.

**Syntax**

\[ \text{FCVTSD}(\text{cond}) \ Sd, \ Dm \]

where:

- \( \text{cond} \) is an optional condition code (see VFP and condition codes on page 6-8).
- \( Sd \) 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**

\[
\begin{align*}
\text{FCVTSD} & \quad S3, \ D14 \\
\text{FCVTSDM} & \quad S0, \ D1
\end{align*}
\]
6.7.6  FDIV

Floating-point divide. FDIV can be scalar, vector, or mixed (see Vector and scalar operations on page 6-7).

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 6-8).

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
6.7.7 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 6-8).
- \(Fd\) 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 \(D\).

There is also an FLD pseudo-instruction (see FLD pseudo-instruction on page 6-36).

**Examples**

- FLD d5, [r7, #-12]
- FLDSNE s3, [r2, #72+count]
FSTS  s2, [r5]
FLDD  d2, [r15, #addr-{PC}]
FLDS  s9, fpconst
6.7.8 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 6-8).

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.
**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, VFPregistors 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, VFPregistors 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, VFPregistors 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}
FSTMFD D 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:

```
FLDMIA0 r13!, {d0,d2,d3}
```
## 6.7.9 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 6-7).

### Syntax

\[ \langle \text{op} \rangle \langle \text{precision} \rangle \{ \text{cond} \} \ F_d, \ F_n, \ F_m \]

where:

- \( \langle \text{op} \rangle \) must be one of FMAC, FNMAC, FMSC, or FNMSC.
- \( \langle \text{precision} \rangle \) must be either S for single-precision, or D for double-precision.
- \( \text{cond} \) is an optional condition code (see VFP and condition codes on page 6-8).
- \( F_d \) is the VFP register for the result.
- \( F_n \) is the VFP register holding the first operand.
- \( F_m \) is the VFP register holding the second operand.

The precision of \( F_d, F_n \) and \( F_m \) must match the precision specified in \( \langle \text{precision} \rangle \).

### Usage

The FMAC instruction calculates \( F_d + F_n \times F_m \) and places the result in \( F_d \).

The FNMAC instruction calculates \( F_d - F_n \times F_m \) and places the result in \( F_d \).

The FMSC instruction calculates \(-F_d + F_n \times F_m \) and places the result in \( F_d \).

The FNMSC instruction calculates \(-F_d - F_n \times F_m \) and places the result in \( F_d \).

### Exceptions

These operations can produce Invalid Operation, Overflow, Underflow, or Inexact exceptions.

### Examples

- FMACD \( d8, d0, d8 \)
- FMACS \( s20, s24, s28 \)
- FNMSCSLE \( s6, s0, s26 \)
6.7.10 FMDRR and FMRRD

Transfer contents between two ARM registers and a double-precision floating-point register.

Syntax

\[
\text{FMDRR}\{\text{cond}\} \ Dn, \ Rd, \ Rn \\
\text{FMRRD}\{\text{cond}\} \ Rd, \ Rn, \ Dn
\]

where:

\(\text{cond}\) is an optional condition code (see VFP and condition codes on page 6-8).

\(Dn\) 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

\[
\text{FMDRR} \ d5, \ r3, \ r4 \\
\text{FMRRDPL} \ r12, \ r2, \ d2
\]
6.7.11 FMDHR, FMDLR, FMRDH, and FMRDL

Transfer contents between an ARM register and a half of a double-precision floating-point register.

Syntax

FMDHR\{cond\} \text{ \textit{Dn}, \textit{Rd}}
FMDLR\{cond\} \text{ \textit{Dn}, \textit{Rd}}
FMRDH\{cond\} \text{ \textit{Rd}, \textit{Dn}}
FMRDL\{cond\} \text{ \textit{Rd}, \textit{Dn}}

where:

\textit{cond} is an optional condition code (see \textit{VFP and condition codes} on page 6-8).
\textit{Dn} is the VFP double-precision register.
\textit{Rd} is the ARM register. \textit{Rd} must not be r15.

Usage

These instructions are used together as matched pairs:

- Use FMDHR with FMDLR
  FMDHR \text{ copy the contents of \textit{Rd} into the high half of \textit{Dn}}
  FMDLR \text{ copy the contents of \textit{Rd} into the low half of \textit{Dn}}

- Use FMRDH with FMRDL
  FMRDH \text{ copy the contents of the high half of \textit{Dn} into \textit{Rd}}
  FMRDL \text{ copy the contents of the low half of \textit{Dn} into \textit{Rd}}.

Exceptions

These instructions do not produce any exceptions.

Examples

FMDHR \text{ \textit{d5}, \textit{r3}}
FMDLR \text{ \textit{d5}, \textit{r12}}
FMRDH \text{ \textit{r5}, \textit{d3}}
FMRDL \text{ \textit{r9}, \textit{d3}}
FMDLRPL \text{ \textit{d2}, \textit{r1}}
6.7.12 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 6-8).
- **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**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Register 1</th>
<th>Register 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>FMRS</td>
<td>r2, s0</td>
<td></td>
</tr>
<tr>
<td>FMSRNE</td>
<td>s30, r5</td>
<td></td>
</tr>
</tbody>
</table>
6.7.13 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 6-8).
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
6.7.14 FMRX, FMXR, and FMSTAT

Transfer contents between an ARM register and a VFP system register.

**Syntax**

\[
\text{FMRX}\{\text{cond}\} \text{ } Rd, \text{ } VFPsysreg \\
\text{FMXR}\{\text{cond}\} \text{ } VFPsysreg, \text{ } Rd \\
\text{FMSTAT}\{\text{cond}\}
\]

where:

*cond* is an optional condition code (see *VFP and condition codes* on page 6-8).

*VFPsysreg* is the VFP system register, usually FPSCR, FPSID, or FPEXC (see *Floating-point registers* on page 6-5).

*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 6-8).

### 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
- **FMXR** r3, FPSID
6.7.15 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 6-7).

Syntax

\[
\text{FMUL<precision><cond> Fd, Fn, Fm}
\]

where:

\(<\text{precision}>\) must be either S for single-precision, or D for double-precision.

\(\text{cond}\) is an optional condition code (see VFP and condition codes on page 6-8).

\(Fd\) 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 \(<\text{precision}>\).

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

\[
\begin{align*}
\text{FNMUL} & \quad s10, s10, s14 \\
\text{FMULDLT} & \quad d0, d7, d8
\end{align*}
\]
6.7.16 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 6-8).
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
6.7.17  **FSQRT**

Floating-point square root instruction. This instruction can be scalar, vector, or mixed (see *Vector and scalar operations* on page 6-7).

**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 6-8).
- `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
6.7.18 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 6-8).
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
6.8 VFP pseudo-instruction

There is one VFP pseudo-instruction.

6.8.1 FLD pseudo-instruction

The FLD pseudo-instruction loads a VFP floating-point register with a single-precision or double-precision floating-point constant.

--- Note ---

You can use FLD only if the command line option -fpu is set to vfp, softvfp+vfp, vfpv1, vfpv2 or softvfp+vfpv2.

This section describes the FLD pseudo-instruction only. See FLD and FST on page 6-22 for information on the FLD instruction.

---

Syntax

FLD<precision>{cond} fp-register,=fp-literal

where:

<precision> can be S for single-precision, or D for double-precision.

cond is an optional condition code.

fp-register is the floating-point register to be loaded.

fp-literal is a single-precision or double-precision floating-point literal (see Floating-point literals on page 3-28).

Usage

The assembler places the constant in a literal pool and generates a program-relative FLD instruction to read the constant from the literal pool. One word in the literal pool is used to store a single-precision constant. Two words are used to store a double-precision constant.

The offset from pc to the constant must be less than 1KB. You are responsible for ensuring that there is a literal pool within range. See LTORG on page 7-14 for more information.
Examples

FLDD  d1,=3.12E106  ; loads 3.12E106 into d1
FLDS  s31,=3.12E-16   ; loads 3.12E-16 into s31
6.9  VFP directives and vector notation

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

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

- \textit{VFPASSERT SCALAR} on page 6-39
- \textit{VFPASSERT VECTOR} on page 6-40.

If you use \texttt{VFPASSERT} directives, you must specify vector details in all VFP data processing instructions. The vector notation is described below. If you do not use \texttt{VFPASSERT} directives you must not use this vector notation.

In VFP data processing instructions, specify vectors of VFP registers using angle brackets:

- \(sn\) is a single-precision scalar register \(n\)
- \(sn\leftarrow\) is a single-precision vector whose length and stride are given by the current vector length and stride, starting at register \(n\)
- \(sn\leftarrow L\) is a single-precision vector of length \(L\), stride 1, starting at register \(n\)
- \(sn\leftarrow L:S\) is a single-precision vector of length \(L\), stride \(S\), starting at register \(n\)

- \(dn\) is a double-precision scalar register \(n\)
- \(dn\leftarrow\) is a double-precision vector whose length and stride are given by the current vector length and stride, starting at register \(n\)
- \(dn\leftarrow L\) is a double-precision vector of length \(L\), stride 1, starting at register \(n\)
- \(dn\leftarrow L:S\) 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 \texttt{DN} and \texttt{SN} directives (see \textit{DN and SN} on page 7-11).

You must not use this vector notation in the \texttt{DN} and \texttt{SN} directives themselves.
6.9.1 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 the *Using the Procedure Call Standard* chapter in *RealView Compilation Tools v2.1 Developer Guide* for more information.

See also:

- *VFP directives and vector notation* on page 6-38
- *VFPASSERT VECTOR* on page 6-40.

---

**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
fadd   d4, d4, d0  ; scalar mode
fadds  s4<3>, s0, s8<3>  ; ok
fabss  s24<1>, s28<1>  ; ERROR, vector in scalar mode
                ; even though length==1
```
6.9.2 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 the Using the Procedure Call Standard chapter in RealView Compilation Tools v2.1 Developer Guide for more information.

See:
- VFP directives and vector notation on page 6-38
- VFPASSERT SCALAR on page 6-39.

Note

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

```assembly
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
fadd   d4, d4, d0          ; ERROR, scalar in vector mode
fadds  s16<3>, s8, 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>, s8, 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>, s2, s16<4>   ; ERROR, wrong stride
fabss  s16<4:2>, s28<4:2>  ; okay
fadds  s8<>, s2, s16<>     ; okay (s8 and s16 both have
; length 4 and stride 2.
; s2 is scalar.)
```
Chapter 7
Directives Reference

This chapter describes the directives that are provided by the ARM® assembler, armasm. It contains the following sections:

- *Alphabetical list of directives* on page 7-2
- *Symbol definition directives* on page 7-3
- *Data definition directives* on page 7-13
  Allocate memory, define data structures, set initial contents of memory.
- *Assembly control directives* on page 7-26
  Conditional assembly, looping, inclusions, and macros.
- *Frame directives* on page 7-34
- *Reporting directives* on page 7-48
- *Miscellaneous directives* on page 7-53.

---------- Note ----------
None of these directives are available in the inline assemblers in the ARM C and C++ compilers.
# 7.1 Alphabetical list of directives

Table 7-1 shows where you can find a description of each directive.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ALIGN</td>
<td>7-54</td>
<td>EXPORT or GLOBAL</td>
<td>7-63</td>
<td>KEEP</td>
<td>7-69</td>
</tr>
<tr>
<td>AREA</td>
<td>7-56</td>
<td>EXPORTAS</td>
<td>7-64</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ASSEMBLE</td>
<td>7-68</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ASSEMBLY</td>
<td>7-68</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ASSEMBLY-1</td>
<td>7-68</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ASMmn</td>
<td>7-68</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ASYMn</td>
<td>7-68</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ASSERT</td>
<td>7-48</td>
<td>EXTERN</td>
<td>7-65</td>
<td>LTORG</td>
<td>7-14</td>
</tr>
<tr>
<td>CP</td>
<td>7-10</td>
<td>FRAME POP</td>
<td>7-36</td>
<td>MEND</td>
<td></td>
</tr>
<tr>
<td>DATA</td>
<td>7-25</td>
<td>FRAME PUSH</td>
<td>7-37</td>
<td>MEXIT</td>
<td>7-29</td>
</tr>
<tr>
<td>DCB</td>
<td>7-18</td>
<td>FRAME REGISTER</td>
<td>7-38</td>
<td>NOFP</td>
<td>7-70</td>
</tr>
<tr>
<td>DCD</td>
<td>7-19</td>
<td>FRAME RESTORE</td>
<td>7-39</td>
<td>OPT</td>
<td>7-50</td>
</tr>
<tr>
<td>DCDU</td>
<td>7-19</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DCO</td>
<td>7-20</td>
<td>FRAME SAVE</td>
<td>7-41</td>
<td>PRESERVE8</td>
<td></td>
</tr>
<tr>
<td>DCFD</td>
<td>7-21</td>
<td>FRAME STATE REMEMBER</td>
<td>7-42</td>
<td>PROC</td>
<td></td>
</tr>
<tr>
<td>DCFDU</td>
<td>7-21</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DCFS</td>
<td>7-22</td>
<td>FRAME STATE RESTORE</td>
<td>7-43</td>
<td>REQUIRE</td>
<td>7-70</td>
</tr>
<tr>
<td>DCFSU</td>
<td>7-22</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DCI</td>
<td>7-23</td>
<td>FUNCTION or PROC</td>
<td>7-45</td>
<td>REQUIRE8 and PRESERVE8</td>
<td></td>
</tr>
<tr>
<td>DCQ</td>
<td>7-24</td>
<td>GBLA, GBLL, and GBLS</td>
<td>7-4</td>
<td>RLIST</td>
<td>7-8</td>
</tr>
<tr>
<td>DCW</td>
<td>7-25</td>
<td>GET or INCLUDE</td>
<td>7-66</td>
<td>RN</td>
<td>7-73</td>
</tr>
<tr>
<td>DCWU</td>
<td>7-25</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DN</td>
<td>7-11</td>
<td>GLOBAL</td>
<td>7-11</td>
<td>ROUT</td>
<td>7-74</td>
</tr>
<tr>
<td>SN</td>
<td>7-11</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ELIF, ELSE see IF</td>
<td></td>
<td></td>
<td></td>
<td>SETA, SETL, and SETS</td>
<td>7-7</td>
</tr>
<tr>
<td>END</td>
<td>7-60</td>
<td>IMPORT</td>
<td>7-67</td>
<td>SN</td>
<td></td>
</tr>
<tr>
<td>ENDFUNC or ENDP</td>
<td>7-47</td>
<td>INCBIN</td>
<td>7-68</td>
<td>SPACE</td>
<td>7-17</td>
</tr>
<tr>
<td>ENDFIF see IF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ENTRY</td>
<td>7-61</td>
<td>INFO</td>
<td>7-49</td>
<td>WHILE and WEND</td>
<td>7-33</td>
</tr>
<tr>
<td>EQU</td>
<td>7-62</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
7.2 Symbol definition directives

This section describes the following directives:

- **GBLA, GBLL, and GBLS** on page 7-4
  Declare a global arithmetic, logical, or string variable.

- **LCLA, LCLL, and LCLS** on page 7-6
  Declare a local arithmetic, logical, or string variable.

- **SETA, SETL, and SETS** on page 7-7
  Set the value of an arithmetic, logical, or string variable.

- **RLIST** on page 7-8
  Define a name for a set of general-purpose registers.

- **CN** on page 7-9
  Define a coprocessor register name.

- **CP** on page 7-10
  Define a coprocessor name.

- **DN and SN** on page 7-11
  Define a double-precision or single-precision VFP register name.

- **FN** on page 7-12
  Define an FPA register name (deprecated).
7.2.1  GBLA, GBLL, and GBLS

The GBLA directive declares a global arithmetic variable, and initializes its value to 0.
The GBLL directive declares a global logical variable, and initializes its value to \{FALSE\}.
The GBLS directive declares a global string variable and initializes its value to a null string, "".

Syntax

\(<gblx>\) variable

where:

\(<gblx>\) is one of GBLA, GBLL, or GBLS.

variable is the name of the variable. variable must be unique among symbols within a source file.

Usage

Using one of these directives for a variable that is already defined re-initializes the variable to the same values given above.

The scope of the variable is limited to the source file that contains it.

Set the value of the variable with a SETA, SETL, or SETS directive (see SETA, SETL, and SETS on page 7-7).

See LCLA, LCLL, and LCLS on page 7-6 for information on declaring local variables.

Global variables can also be set with the -predefine assembler command-line option. See Command syntax on page 3-2 for more information.
Examples

Example 7-1 declares a variable `objectsize`, sets the value of `objectsize` to 0xFF, and then uses it later in a SPACE directive.

Example 7-1

```
GBLA objectsize ; declare the variable name
objectsize SETA 0xFF ; set its value
. . ; other code
. SPACE objectsize ; quote the variable
```

Example 7-2 shows how to declare and set a variable when you invoke armasm. Use this when you need to set the value of a variable at assembly time. --pd is a synonym for --predefine.

Example 7-2

```
armasm --pd "objectsize SETA 0xFF" --o objectfile sourcefile
```
7.2.2  LCLA, LCLL, and LCLS

The LCLA directive declares a local arithmetic variable, and initializes its value to 0.
The LCLL directive declares a local logical variable, and initializes its value to \{FALSE\}.
The LCLS directive declares a local string variable, and initializes its value to a null string, "".

Syntax

\(<lclx>\) variable

where:

\(<lclx>\) is one of LCLA, LCLL, or LCLS.
variable is the name of the variable. variable must be unique within the macro that contains it.

Usage

Using one of these directives for a variable that is already defined re-initializes the variable to the same values given above.

The scope of the variable is limited to a particular instantiation of the macro that contains it (see MACRO and MEND on page 7-27).

Set the value of the variable with a SETA, SETL, or SETS directive (see SETA, SETL, and SETS on page 7-7).

See GBLA, GBLL, and GBLS on page 7-4 for information on declaring global variables.

Example

MACRO                    ; Declare a macro
$message \$a
$LCLS \$err
$err \$ets "error no: "    ; Set value of err
\$label ; code
_INFO 0, "err":CC::STR:\$a  ; Use string
%MEND
7.2.3 SETA, SETL, and SETS

The SETA directive sets the value of a local or global arithmetic variable.
The SETL directive sets the value of a local or global logical variable.
The SETS directive sets the value of a local or global string variable.

Syntax

```
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 7-4 and LCLA, LCLL, and LCLS on page 7-6 for more information.

You can also predefine variable names on the command line. See Command syntax on page 3-2 for more information.

Examples

```
GBLA    VersionNumber
VersionNumber   SETA    21
GBLL    Debug
Debug           SETL    {TRUE}
GBLS    VersionString
VersionString   SETS    "Version 1.0"
```
7.2.4 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}
7.2.5 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
c               ; coprocessor register 6
```
7.2.6 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
7.2.7 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 6-38).

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
```
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 deprecated. This directive will not be provided in future releases of RVDS.

### 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
```
7.3 Data definition directives

This section describes the following directives:

- **LTORG** on page 7-14
  Set an origin for a literal pool.
- **MAP** on page 7-15
  Set the origin of a storage map.
- **FIELD** on page 7-16
  Define a field within a storage map.
- **SPACE** on page 7-17
  Allocate a zeroed block of memory.
- **DCB** on page 7-18
  Allocate bytes of memory, and specify the initial contents.
- **DCD and DCDU** on page 7-19
  Allocate words of memory, and specify the initial contents.
- **DCDO** on page 7-20
  Allocate words of memory, and specify the initial contents as offsets from the static base register.
- **DCFD and DCFDU** on page 7-21
  Allocate doublewords of memory, and specify the initial contents as double-precision floating-point numbers.
- **DCFS and DCFSU** on page 7-22
  Allocate words of memory, and specify the initial contents as single-precision floating-point numbers.
- **DCI** on page 7-23
  Allocate words of memory, and specify the initial contents. Mark the location as code not data.
- **DCQ and DCQU** on page 7-24
  Allocate doublewords of memory, and specify the initial contents as 64-bit integers.
- **DCW and DCWU** on page 7-25
  Allocate halfwords of memory, and specify the initial contents.
- **DATA** on page 7-25
  Mark data within a code section. Obsolete, for backwards compatibility only.
7.3.1 LTORG

The LTORG directive instructs the assembler to assemble the current literal pool immediately.

Syntax

LTORG

Usage

The assembler assembles the current literal pool at the end of every code section. The end of a code section is determined by the AREA directive at the beginning of the following section, or the end of the assembly.

These default literal pools can sometimes be out of range of some LDR, LDFD, and LDFS pseudo-instructions. See LDR ARM pseudo-instruction on page 4-124 and LDR Thumb pseudo-instruction on page 5-49 for more information. 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

; function body
func1                           ; code
LDR     r1,=0x55555555  ; => LDR R1, [pc, #offset to Literal Pool 1]
; code
MOV     pc,lr           ; end function
LTORG                   ; Literal Pool 1 contains literal &55555555.

data    SPACE   4200            ; Clears 4200 bytes of memory,
; starting at current location.
END                     ; Default literal pool is empty.
```
7.3.2 MAP

The MAP directive sets the origin of a storage map to a specified address. The storage-map location counter, \{VAR\}, is set to the same address. \^\ is a synonym for MAP.

Syntax

\texttt{MAP expr\{,base-register\}}

where:

\texttt{expr} is a numeric or program-relative expression:

\begin{itemize}
  \item If \texttt{base-register} is not specified, \texttt{expr} evaluates to the address where the storage map starts. The storage map location counter is set to this address.
  \item If \texttt{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.
\end{itemize}

\texttt{base-register} specifies a register. If \texttt{base-register} is specified, the address where the storage map starts is the sum of \texttt{expr}, and the value in \texttt{base-register} at runtime.

Usage

Use the MAP directive in combination with the FIELD directive to describe a storage map.

Specify \texttt{base-register} to define register-relative labels. The base register becomes implicit in all labels defined by following FIELD directives, until the next MAP directive. The register-relative labels can be used in load and store instructions. See FIELD on page 7-16 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

\begin{verbatim}
MAP 0,r9
MAP 0xff,r9
\end{verbatim}
7.3.3 FIELD

The FIELD directive describes space within a storage map that has been defined using the MAP directive. # is a synonym for FIELD.

Syntax

\{label\} FIELD expr

where:

label is an optional label. If specified, label is assigned the value of the storage location counter, \{VAR\}. The storage location counter is then incremented by the value of expr.

expr is an expression that evaluates to the number of bytes to increment the storage counter.

Usage

If a storage map is set by a MAP directive that specifies a base-register, the base register is implicit in all labels defined by following FIELD directives, until the next MAP directive. These register-relative labels can be quoted in load and store instructions (see MAP on page 7-15).

Note

You must be careful when using MAP, FIELD, and register-relative labels. See Describing data structures with MAP and FIELD directives on page 2-53 for more information.

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]
```
7.3.4 SPACE

The SPACE directive reserves a zeroed block of memory. \% is a synonym for SPACE.

Syntax

\{label\} SPACE expr

where:

\textit{expr} evaluates to the number of zeroed bytes to reserve (see \textit{Numeric expressions} on page 3-26).

Usage

You \textit{must} use a DATA directive if you use SPACE to define labeled data within Thumb® code. See \textit{DATA} on page 7-25 for more information.

Use the ALIGN directive to align any code following a SPACE directive. See \textit{ALIGN} on page 7-54 for more information.

See also:

- \textit{DCB} on page 7-18
- \textit{DCD and DCDU} on page 7-19
- \textit{DCDO} on page 7-20
- \textit{DCW and DCWU} on page 7-25.

Example

\begin{verbatim}
AREA    MyData, DATA, READWRITE
data1   SPACE   255     ; defines 255 bytes of zeroed store
\end{verbatim}
7.3.5 DCB

The DCB directive allocates one or more bytes of memory, and defines the initial runtime contents of the memory. = is a synonym for DCB.

Syntax

\{label\} DCB expr{,expr}...

where:

expr is either:

- A numeric expression that evaluates to an integer in the range –128 to 255 (see Numeric expressions on page 3-26).
- 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 7-54 for more information.

See also:
- DCD and DCDU on page 7-19
- DCQ and DCQU on page 7-24
- DCW and DCWU on page 7-25
- SPACE on page 7-17.

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
7.3.6  DCD and DCDU

The DCD directive allocates one or more words of memory, aligned on four-byte boundaries, and defines the initial runtime contents of the memory.

& is a synonym for DCD.

DCDU is the same, except that the memory alignment is arbitrary.

Syntax

\{label\} DCD{U} expr{,expr}

where:

expr is either:

- a numeric expression (see Numeric expressions on page 3-26).
- 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 7-18
- DCW and DCWU on page 7-25
- DCQ and DCQU on page 7-24
- SPACE on page 7-17.

Examples

```
data1  DCD     1,5,20      ; Defines 3 words containing decimal values 1, 5, and 20

data2  DCD     mem06 + 4   ; Defines 1 word containing 4 + the address of the label mem06

AREA    MyData, DATA, READWRITE
DCB     255     ; Now misaligned ...

data3  DCDU    1,5,20      ; Defines 3 words containing 1, 5 and 20, not word aligned
```
7.3.7 DCDO

The DCDO directive allocates one or more words of memory, aligned on four-byte boundaries, and defines the initial runtime contents of the memory as an offset from the static base register, sb (r9).

Syntax

```
{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.
```
7.3.8 DCFD and DCFDU

The DCFD directive allocates memory for word-aligned double-precision floating-point numbers, and defines the initial runtime contents of the memory. Double-precision numbers occupy two words and must be word aligned to be used in arithmetic operations.

DCFDU is the same, except that the memory alignment is arbitrary.

Syntax

\{label\} DCFD\{U\} \ fpliteral\{,fpliteral\}...

where:

\ fpliteral\ is a double-precision floating-point literal (see Floating-point literals on page 3-28).

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 7-22.

Examples

<table>
<thead>
<tr>
<th>DCFD</th>
<th>1E308, -4E-100</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCFDU</td>
<td>10000, -.1, 3.1E26</td>
</tr>
</tbody>
</table>
7.3.9 DCFS and DCFSU

The DCFS directive allocates memory for word-aligned single-precision floating-point numbers, and defines the initial runtime contents of the memory. Single-precision numbers occupy one word and must be word aligned to be used in arithmetic operations.

DCFSU is the same, except that the memory alignment is arbitrary.

Syntax

{label} 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 7-21.

Example

DCFS  1E3, -4E-9
DCFSU 1.0, -.1, 3.1E6
7.3.10 DCI

In ARM code, the DCI directive allocates one or more words of memory, aligned on four-byte boundaries, and defines the initial runtime contents of the memory.

In Thumb code, the DCI directive allocates one or more halfwords of memory, aligned on two-byte boundaries, and defines the initial runtime contents of the memory.

Syntax

{label} DCI 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.

See also DCD and DCDU on page 7-19 and DCW and DCWU on page 7-25.

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
7.3.11 DCQ and DCQU

The DCQ directive allocates one or more eight-byte blocks of memory, aligned on four-byte boundaries, and defines the initial runtime contents of the memory.

DCQU is the same, except that the memory alignment is arbitrary.

Syntax

{label} DCQ{U} {-}literal{,{-}literal}...

where:

literal is a 64-bit numeric literal (see Numeric literals on page 3-27).

The range of numbers allowed is 0 to $2^{64} - 1$.

In addition to the characters normally allowed in a numeric literal, you can prefix literal with a minus sign. In this case, the range of numbers allowed is $-2^{63}$ to $-1$.

The result of specifying $-n$ is the same as the result of specifying $2^{64} - n$.

Usage

DCQ inserts up to three bytes of padding before the first defined eight-byte block, if necessary, to achieve four-byte alignment.

Use DCQU if you do not require alignment.

See also:

- DCB on page 7-18
- DCD and DCDU on page 7-19
- DCW and DCWU on page 7-25
- SPACE on page 7-17.

Example

```
AREA MiscData, DATA, READWRITE
data DCQ -225,2_101 ; 2_101 means binary 101.
DCQU number+4 ; number must already be defined.
```
7.3.12 DCW and DCWU

The DCW directive allocates one or more halfwords of memory, aligned on two-byte boundaries, and defines the initial runtime contents of the memory.

DCWU is the same, except that the memory alignment is arbitrary.

Syntax

\{label\} DCW 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 7-18
- DCD and DCDU on page 7-19
- DCQ and DCQU on page 7-24
- SPACE on page 7-17.

Example

data DCW -225,2\*number ; number must already be defined

DCWU number+4

7.3.13 DATA

The DATA directive is no longer needed. It is ignored by the assembler.
7.4 Assembly control directives

This section describes the following directives:

- *MACRO and MEND* on page 7-27
- *MEXIT* on page 7-29
- *IF, ELSE, ENDIF, and ELIF* on page 7-30
- *WHILE and WEND* on page 7-33.

7.4.1 Nesting directives

The following structures can be nested to a total depth of 256:

- *MACRO* definitions
- *WHILE...WEND* loops
- *IF...ELSE...ENDIF* conditional structures
- *INCLUDE* file inclusions.

The limit applies to all structures taken together, however they are nested. The limit is not 256 of each type of structure.
### 7.4.2 MACRO and MEND

The MACRO directive marks the start of the definition of a macro. Macro expansion terminates at the MEND directive. See *Using macros* on page 2-50 for more information.

#### Syntax

Two directives are used to define a macro. The syntax is:

```assembly
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:
  ```assembly
  $parameter="default value"
  ```
  Double quotes must be used if there are any spaces within, or at either end of, the default value.

#### Usage

If you start any WHILE...WEND loops or IF...ENDIF conditions within a macro, they must be closed before the MEND directive is reached. See *MEXIT* on page 7-29 if you need to allow 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.
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 7-6). Macros can be nested (see Nesting directives on page 7-26).

Examples

; macro definition
MACRO $label          xmac    $p1,$p2
; code
$label.loop1    ; code
; code
BGE     $label.loop1
$label.loop2    ; code
BL      $p1
BGT     $label.loop2
; code
ADR     $p2
; code
MEND        ; end macro definition

; macro invocation
abc             xmac    subr1,de ; invoke macro
; code          ; this is what is
abcloop1        ; code          ; is produced when
; code          ; the xmac macro is
BGE     abcloop1 ; expanded
abcloop2        ; code
BL      subr1
BGT     abcloop2
; code
ADR     de
; code
Using a macro to produce assembly-time diagnostics:

```
MACRO                        ; Macro definition
diagnose $param1="default"  ; This macro produces
INFO    0,"$param1"        ; assembly-time diagnostics
MEND    ; (on second assembly pass)

; macro expansion

diagnose            ; Prints blank line at assembly-time
diagnose "hello"    ; Prints "hello" at assembly-time

diagnose |          ; Prints "default" at assembly-time
```

### 7.4.3 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 7-27.

**Example**

```
MACRO $abc    macro   abc $param1,$param2
    ; code
    WHILE condition1
    ; code
    IF condition2
    ; code
    MEXIT
    ELSE
    ; code
    ENDIF
    WEND
    ; code
    MEND
```
7.4.4 IF, ELSE, ENDF, 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 ENDF directive marks the end of a sequence of instructions and/or directives that you want to be conditionally assembled. ] is a synonym for ENDF.

The ELIF directive creates a structure equivalent to ELSE IF, without the need for nesting or repeating the condition. See Using ELIF on page 7-31 for details.

**Syntax**

```plaintext
IF logical-expression
...
{ELSE
...
}
ENDIF
```

where:

- `logical-expression` is an expression that evaluates to either {TRUE} or {FALSE}.

See Relational operators on page 3-37.

**Usage**

Use IF with ENDF, and optionally with ELSE, for sequences of instructions and/or directives that are only to be assembled or acted on under a specified condition.

IF...ENDIF conditions can be nested (see Nesting directives on page 7-26).
Using ELIF

Without using ELIF, you can construct a nested set of conditional instructions like this:

```
IF logical-expression
  instructions
ELSE
  IF logical-expression2
    instructions
  ELSE
    IF logical-expression3
      instructions
    ENDIF
  ENDIF
ENDIF
ENDIF
ENDIF
```

A nested structure like this can be nested up to 256 levels deep.

You can write the same structure more simply using ELIF:

```
IF logical-expression
  instructions
ELIF logical-expression2
  instructions
ELIF logical-expression3
  instructions
ENDIF
```

This structure only adds one to the current nesting depth, for the IF ENDIF pair.
Examples

Example 7-3 assembles the first set of instructions if NEWVERSION is defined, or the alternative set otherwise.

```
Example 7-3 Assembly conditional on a variable being defined

IF :DEF:NEWVERSION
    ; first set of instructions/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 7-4 assembles the first set of instructions if NEWVERSION has the value {TRUE}, or the alternative set otherwise.

```
Example 7-4 Assembly conditional on a variable 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
```
7.4.5 WHILE and WEND

The WHILE directive starts a sequence of instructions or directives that are to be assembled repeatedly. The sequence is terminated with a WEND directive.

**Syntax**

WHILE logical-expression

code

WEND

where:

logical-expression

is an expression that can evaluate to either {TRUE} or {FALSE} (see Logical expressions on page 3-29).

**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 7-26).

**Example**

```plaintext
count SETA 1                   ; you are not restricted to
WHILE count <= 4          ; such simple conditions
  count SETA count+1             ; In this case,
    ; code
    ; code
  WEND ; this code will be
          ; repeated four times
```
7.5 Frame directives

This section describes the following directives:

- FRAME ADDRESS on page 7-35
- FRAME POP on page 7-36
- FRAME PUSH on page 7-37
- FRAME REGISTER on page 7-38
- FRAME RESTORE on page 7-39
- FRAME RETURN ADDRESS on page 7-40
- FRAME SAVE on page 7-41
- FRAME STATE REMEMBER on page 7-42
- FRAME STATE RESTORE on page 7-43
- FRAME UNWIND ON on page 7-44
- FRAME UNWIND OFF on page 7-44
- FUNCTION or PROC on page 7-45
- ENDFUNC or ENDP on page 7-47.

Correct use of these directives:

- allows the arm\texttt{link} -\texttt{callgraph} option to calculate stack usage of assembler functions
- helps you to avoid errors in function construction, particularly when you are modifying existing code
- allows the assembler to alert you to errors in function construction
- enables backtracing of function calls during debugging
- allows 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
- 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 2, the canonical frame address is an address on the stack specifying where the call frame of an interrupted function is located.
7.5.1 FRAME ADDRESS

The FRAME ADDRESS directive describes how to calculate the canonical frame address for following instructions. You can only use it in functions with FUNCTION and ENDFUNC or PROC and ENDP directives.

Syntax

FRAME ADDRESS reg[,offset]

where:

reg is the register on which the canonical frame address is to be based. This is sp unless the function uses a separate frame pointer.

offset is the offset of the canonical frame address from reg. If offset is zero, you can omit it.

Usage

Use FRAME ADDRESS if your code alters which register the canonical frame address is based on, or if it 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 7-37).

If your code uses a single instruction to load registers and alter the stack pointer, you can use FRAME POP instead of using both FRAME ADDRESS and FRAME RESTORE (see FRAME POP on page 7-36).

Example

_fn     FUNCTION        ; CFA (Canonical Frame Address) is value of sp on entry to function
          ; of sp on entry to function
STMFD   sp!, {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
7.5.2 FRAME POP

Use the FRAME POP directive to inform the assembler when the callee reloads registers. You can only use it within functions with FUNCTION and ENDFUNC or PROC and ENDP directives.

You 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 7-35 and FRAME RESTORE on page 7-39.
7.5.3 FRAME PUSH

Use the FRAME PUSH directive to inform the assembler when the callee saves registers, normally at function entry. You can only use it within functions with FUNCTION and ENDFUNC or PROC and ENDP directives.

Syntax

There are two alternative syntaxes for FRAME PUSH:

FRAME PUSH {reglist}

FRAME PUSH 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 7-35 and FRAME SAVE on page 7-41.
Example

```
p PROC ; Canonical frame address is sp + 0
EXPORT p
STMFD sp!,{r4-r6,lr}
; sp has moved relative to the canonical frame address,
; and registers r4, r5, r6 and lr are now on the stack
FRAME PUSH {r4-r6,lr}
; Equivalent to:
; FRAME ADDRESS    sp,16       ; 16 bytes in {r4-r6,lr}
; FRAME SAVE    {r4-r6,lr},-16
```

7.5.4 FRAME REGISTER

Use the `FRAME REGISTER` directive to maintain a record of the locations of function arguments held in registers. You can only use it within functions with `FUNCTION` and `ENDFUNC` or `PROC` and `ENDP` directives.

Syntax

```
FRAME REGISTER reg1,reg2
```

where:

- `reg1` is the register that held the argument on entry to the function.
- `reg2` is the register in which the value is preserved.

Usage

Use the `FRAME REGISTER` directive when you use a register to preserve an argument that was held in a different register on entry to a function.
7.5.5 FRAME RESTORE

Use the FRAME RESTORE directive to inform the assembler that the contents of specified registers have been restored to the values they had on entry to the function. You can only use it within functions with FUNCTION and ENDFUNC or PROC and ENDP directives.

Syntax

FRAME RESTORE {reglist}

where:

reglist is a list of registers whose contents have been restored. There must be at least one register in the list.

Usage

Use FRAME RESTORE immediately after the callee reloads registers from the stack. You 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 7-36).
7.5.6 FRAME RETURN ADDRESS

The FRAME RETURN ADDRESS directive provides for functions that use a register other than r14 for their return address. You can only use it within functions with FUNCTION and ENDFUNC or PROC and ENDP directives.

Note

Any function that uses a register other than r14 for its return address is not AAPCS compliant. Such a function must not be exported.

Syntax

```
FRAME RETURN ADDRESS reg
```

where:

```
reg
```

is the register used for the return address.

Usage

Use the FRAME RETURN ADDRESS directive in any function that does not use r14 for its return address. Otherwise, a debugger cannot backtrace through the function.

Use FRAME RETURN ADDRESS immediately after the FUNCTION or PROC directive that introduces the function.
7.5.7 FRAME SAVE

The FRAME SAVE directive describes the location of saved register contents relative to the canonical frame address. You can only use it within functions with FUNCTION and ENDFUNC or PROC and ENDP directives.

Syntax

FRAME SAVE {reglist}, offset

where:

reglist is a list of registers stored consecutively starting at offset from the canonical frame address. There must be at least one register in the list.

Usage

Use FRAME SAVE immediately after the callee stores registers onto the stack.

reglist can include registers which are not required for backtracing. The assembler determines which registers it 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 7-37).
7.5.8 FRAME STATE REMEMBER

The FRAME STATE REMEMBER directive saves the current information on how to calculate the canonical frame address and locations of saved register values. You can only use it within functions with FUNCTION and ENDFUNC or PROC and ENDP directives.

Syntax

FRAME STATE REMEMBER

Usage

During an inline exit sequence the information about calculation of canonical frame address and locations of saved register values can change. After the exit sequence another branch can continue using the same information as before. Use FRAME STATE REMEMBER to preserve this information, and FRAME STATE RESTORE to restore it.

These directives can be nested. Each FRAME STATE RESTORE directive must have a corresponding FRAME STATE REMEMBER directive. See:

- FRAME STATE RESTORE on page 7-43
- FUNCTION or PROC on page 7-45.

Example

; function code
FRAME STATE REMEMBER
; save frame state before in-line exit sequence
LDMFD sp!,{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
LDMFD sp!,{r4-r6,pc}
ENDP
7.5.9 FRAME STATE RESTORE

The FRAME STATE RESTORE directive restores information about how to calculate the canonical frame address and locations of saved register values. You can only use it within functions with FUNCTION and ENDFUNC or PROC and ENDP directives.

Syntax

FRAME STATE RESTORE

Usage

See:

- FRAME STATE REMEMBER on page 7-42
- FUNCTION or PROC on page 7-45.
7.5.10 FRAME UNWIND ON

The FRAME UNWIND ON directive instructs the assembler to produce unwind tables for this and subsequent functions.

Syntax
FRAME UNWIND ON

Usage
You can use this directive outside functions. In this case, the assembler produces unwind tables for all following functions until it reaches a FRAME UNWIND OFF directive.

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

7.5.11 FRAME UNWIND OFF

The FRAME UNWIND OFF directive instructs the assembler to produce nounwind tables for this and subsequent functions.

Syntax
FRAME UNWIND OFF

Usage
You can use this directive outside functions. In this case, the assembler produces nounwind tables for all following functions until it reaches a FRAME UNWIND ON directive.

See also Controlling exception table generation on page 3-13.
7.5.12 FUNCTION or PROC

The FUNCTION directive marks the start of an AAPCS-conforming function. PROC is a synonym for FUNCTION.

Syntax

`label FUNCTION [{Reglist1} [, {Reglist2}]]`

where:

- `reglist1` is an optional list of callee saved ARM registers. If `reglist1` is not present, and your debugger checks register usage, it will assume that the AAPCS is in use.
- `reglist2` is an optional list of callee saved VFP registers.

Usage

Use FUNCTION to mark the start of functions. The assembler uses FUNCTION to identify the start of a function when producing DWARF call frame information for ELF.

FUNCTION sets the canonical frame address to be sp, and the frame state stack to be empty.

Each FUNCTION directive must have a matching ENDFUNC directive. You must not nest FUNCTION/ENDFUNC pairs, and they must not contain PROC or ENDP directives.

You can use the optional `reglist` parameters to inform the debugger about an alternative procedure call standard, if you are using your own. Not all debuggers support this feature. See your debugger documentation for details.

See also FRAME ADDRESS on page 7-35 to FRAME STATE RESTORE on page 7-43.

--- Note ---

FUNCTION does not automatically cause alignment to a word boundary (or halfword boundary for Thumb). Use ALIGN if necessary to ensure alignment, otherwise the call frame might not point to the start of the function. See ALIGN on page 7-54 for further information.
Examples

ALIGN ; ensures alignment
dadd FUNCTION ; without the ALIGN directive, this might not be word-aligned
EXPORT dadd
STMFD sp!,{r4-r6,lr} ; this line automatically word-aligned
FRAME PUSH {r4-r6,lr}
; subroutine body
LDMFD sp!,{r4-r6,pc}
ENDFUNC

func6 PROC {r4-r8,r12},{D1-D3} ; non-AAPCS-conforming function
... ENDP
7.5.13 ENDFUNC or ENDP

The ENDFUNC directive marks the end of an AAPCS-conforming function (see FUNCTION or PROC on page 7-45). ENDP is a synonym for ENDFUNC.
7.6 Reporting directives

This section describes the following directives:

- **ASSERT**
  generates an error message if an assertion is false during assembly.

- **INFO** on page 7-49
  generates diagnostic information during assembly.

- **OPT** on page 7-50
  sets listing options.

- **TTL and SUBT** on page 7-52
  insert titles and subtitles in listings.

7.6.1 ASSERT

The ASSERT directive generates an error message during the second pass of the assembly if a given assertion is false.

**Syntax**

```
ASSERT logical-expression
```

where:

- `logical-expression` is an assertion that can evaluate to either \{TRUE\} or \{FALSE\}.

**Usage**

Use ASSERT to ensure that any necessary condition is met during assembly.

If the assertion is false an error message is generated and assembly fails.

See also INFO on page 7-49.

**Example**

```
ASSERT label1 <= label2 ; Tests if the address
; represented by label1 ; is <= the address
; represented by label2.
```
7.6.2 INFO

The INFO directive supports diagnostic generation on either pass of the assembly.

! is very similar to INFO, but has less detailed reporting.

Syntax

INFO numeric-expression, string-expression

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 7-48.

Examples

INFO 0, "Version 1.0"

IF endofdata <= label1
    INFO 4, "Data overrun at label1"
ENDIF
7.6.3 OPT

The OPT directive sets listing options from within the source code.

Syntax

OPT \( n \)

where:

\( n \) is the OPT directive setting. Table 7-2 lists valid settings.

<table>
<thead>
<tr>
<th>OPT ( n )</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Turns on normal listing.</td>
</tr>
<tr>
<td>2</td>
<td>Turns off normal listing.</td>
</tr>
<tr>
<td>4</td>
<td>Page throw. Issues an immediate form feed and starts a new page.</td>
</tr>
<tr>
<td>8</td>
<td>Resets the line number counter to zero.</td>
</tr>
<tr>
<td>16</td>
<td>Turns on listing for SET, GBL and LCL directives.</td>
</tr>
<tr>
<td>32</td>
<td>Turns off listing for SET, GBL and LCL directives.</td>
</tr>
<tr>
<td>64</td>
<td>Turns on listing of macro expansions.</td>
</tr>
<tr>
<td>128</td>
<td>Turns off listing of macro expansions.</td>
</tr>
<tr>
<td>256</td>
<td>Turns on listing of macro invocations.</td>
</tr>
<tr>
<td>512</td>
<td>Turns off listing of macro invocations.</td>
</tr>
<tr>
<td>1024</td>
<td>Turns on the first pass listing.</td>
</tr>
<tr>
<td>2048</td>
<td>Turns off the first pass listing.</td>
</tr>
<tr>
<td>4096</td>
<td>Turns on listing of conditional directives.</td>
</tr>
<tr>
<td>8192</td>
<td>Turns off listing of conditional directives.</td>
</tr>
<tr>
<td>16384</td>
<td>Turns on listing of MEND directives.</td>
</tr>
<tr>
<td>32768</td>
<td>Turns off listing of MEND directives.</td>
</tr>
</tbody>
</table>

Usage

Specify the -list assembler option to turn on listing.
By default the -list option produces a normal listing that includes variable declarations, macro expansions, call-conditioned directives, and MEND directives. The listing is produced on the second pass only. Use the OPT directive to modify the default listing options from within your code. See 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

```assembly
AREA Example, CODE, READONLY
start ; code ; code
        BL func1
        ; code
        OPT 4 ; places a page break before func1
func1 ; code
```
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**

<table>
<thead>
<tr>
<th>TTL</th>
<th>First Title</th>
<th>; places a title on the first and subsequent pages of a listing file.</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUBT</td>
<td>First Subtitle</td>
<td>; places a subtitle on the second and subsequent pages of a listing file.</td>
</tr>
</tbody>
</table>
7.7 Miscellaneous directives

This section describes the following directives:

- ALIGN on page 7-54
- AREA on page 7-56
- CODE16 and CODE32 on page 7-59
- END on page 7-60
- ENTRY on page 7-61
- EQU on page 7-62
- EXPORT or GLOBAL on page 7-63
- EXTERN on page 7-65
- GET or INCLUDE on page 7-66
- IMPORT on page 7-67
- INCBIN on page 7-68
- KEEP on page 7-69
- NOFP on page 7-70
- REQUIRE on page 7-70
- REQUIRE8 and PRESERVE8 on page 7-71
- RN on page 7-73
- ROUT on page 7-74.
7.7.1 ALIGN

The ALIGN directive aligns the current location to a specified boundary by padding with zeroes.

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:

\[ \text{offset} + n \cdot \text{expr} \]

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 zeroes if pad is not specified.

Usage

Use ALIGN to ensure that your data and code is aligned to appropriate boundaries. This is typically required in the following circumstances:

- The ADR Thumb pseudo-instruction can only load addresses that are word aligned, but a label within Thumb code might not be word aligned. Use ALIGN 4 to ensure four-byte alignment of an address within Thumb code.

- Use ALIGN to take advantage of caches on some ARM processors. For example, the ARM940T has a cache with 16-byte lines. Use ALIGN 16 to align function entries on 16-byte boundaries and maximize the efficiency of the cache.

- LDRD and STRD doubleword data transfers must be eight-byte aligned. Use ALIGN 8 before memory allocation directives such as DCQ (see Data definition directives on page 7-13) if the data is to be accessed using LDRD or STRD.
A label on a line by itself can be arbitrarily aligned. Following ARM code is word-aligned (Thumb code is halfword aligned). The label therefore does not address the code correctly. Use ALIGN 4 (or ALIGN 2 for Thumb) before the label.

Alignment is relative to the start of the ELF section where the routine is located. The section must be aligned to the same, or coarser, boundaries. The ALIGN attribute on the AREA directive is specified differently (see AREA on page 7-56 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
```
7.7.2 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 2-16 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 2expression-byte boundary. For example, if expression is 10, the section is aligned on a 1KB boundary. This is not the same as the way that the ALIGN directive is specified. See ALIGN on page 7-54.

--- Note ---

Do not use ALIGN=0 or ALIGN=1 for code sections.

ASSOC=section

section specifies an associated ELF section. sectionnamemust 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.
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 **Linker** chapter in *RealView Compilation Tools v2.1 Linker and Utilities Guide*.

**COMMON**

Is a common data section. You must not define any code or data in it. It is initialized to zeroes 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, DCU, 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 **Linker** chapter in *RealView Compilation Tools v2.1 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 7-74).

There must be at least one **AREA** directive for an assembly.
Example

The following example defines a read-only code section named Example.

```
AREA Example, CODE, READONLY ; An example code section.
; code
```
7.7.3 CODE16 and CODE32

The CODE16 directive instructs the assembler to interpret subsequent instructions as 16-bit Thumb instructions. If necessary, it also inserts a byte of padding to align to the next halfword boundary.

The CODE32 directive instructs the assembler to interpret subsequent instructions as 32-bit ARM instructions. If necessary, it also inserts up to three bytes of padding to align to the next word boundary.

Syntax

CODE16

CODE32

Usage

In files that contain a mixture of ARM and Thumb code:

- Use CODE16 when changing from ARM state to Thumb state. CODE16 must precede any Thumb code.
- Use CODE32 when changing from Thumb state to ARM state. CODE32 must precede any ARM code.

CODE16 and CODE32 do not assemble to instructions that change the state. They only instruct the assembler to assemble Thumb or ARM 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

CODE32

; 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 Thumb
start MOV r1,#10    ; Thumb instructions
```
7.7.4 END

The END directive informs the assembler that it has reached the end of a source file.

Syntax

END

Usage

Every assembly language source file must end with END on a line by itself.

If the source file has been included in a parent file by a GET directive, the assembler returns to the parent file and continues assembly at the first line following the GET directive. See GET or INCLUDE on page 7-66 for more information.

If END is reached in the top-level source file during the first pass without any errors, the second pass begins.

If END is reached in the top-level source file during the second pass, the assembler finishes the assembly and writes the appropriate output.
7.7.5 ENTRY

The ENTRY directive declares an entry point to a program.

Syntax
ENTRY

Usage
You must specify at least one ENTRY point for a program. If no ENTRY exists, a warning is generated at link time.

You must not use more than one ENTRY directive in a single source file. Not every source file has to have an ENTRY directive. If more than one ENTRY exists in a single source file, an error message is generated at assembly time.

Example
AREA ARMex, CODE, READONLY
ENTRY ; Entry point for the application
7.7.6  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

\[ \text{name EQU expr[\{, type\}]} \]

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:
  - **CODE16**
  - **CODE32**
  - **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 CODE16, CODE32, or DATA, according to type. This can be used by the linker.

Usage

Use EQU to define constants. This is similar to the use of \#define to define a constant in C.

See **KEEP** on page 7-69 and **EXPORT or GLOBAL** on page 7-63 for information on exporting symbols.

Examples

- abc EQU 2  ; assigns the value 2 to the symbol abc.
- xyz EQU label+8  ; assigns the address (label+8) to the symbol xyz.
- fiq EQU 0x1C, CODE32  ; assigns the absolute address 0x1C to the symbol fiq, and marks it as code
7.7.7 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]}

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.

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.

See also IMPORT on page 7-67.

Example

```
AREA Example, CODE, READONLY
EXPORT DoAdd ; Export the function name
             ; to be used by external modules.

DoAdd ADD r0, r0, r1
```
7.7.8 EXPORTAS

The EXPORTAS directive allows you to export a symbol to the object file, corresponding to a different symbol in the source file.

Syntax

EXPORTAS symbol1, symbol2

where:

symbol1 is the symbol name in the source file. symbol1 must have been defined already. It can be any symbol, including an area name, a label, or a constant.

symbol2 is the symbol name you want to appear in the object file.

The symbol names are case-sensitive.

Usage

Use EXPORTAS to change a symbol in the object file without having to change every instance in the source file.

See also EXPORT or GLOBAL on page 7-63.

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.
7.7.9 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 7-67, and EXPORT or GLOBAL on page 7-63).

Syntax

EXTERN symbol{[WEAK]}

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.

Usage

The name is resolved at link time to a symbol defined in a separate object file. The symbol is treated as a program address. If [WEAK] is not specified, the linker generates an error if no corresponding symbol is found at link time.

If [WEAK] is specified and no corresponding symbol is found at link time:

- If the reference is the destination of a B or BL instruction, the value of the symbol is taken as the address of the following instruction. This makes the B or BL instruction effectively a NOP.
- Otherwise, the value of the symbol is taken as zero.

Example

This example tests to see if the C++ library has been linked, and branches conditionally on the result.

```
AREA Example, CODE, READONLY
EXTERN __CPP_INITIALIZE{[WEAK]} ; If C++ library linked, gets the address of __CPP_INITIALIZE function.
LDR r0,=__CPP_INITIALIZE ; If not linked, address is zeroed.
CMP r0,#0 ; Test if zero.
BEQ nocplusplus ; Branch on the result.
```
7.7.10  GET or INCLUDE

The GET directive includes a file within the file being assembled. The included file is assembled at the location of the GET directive. INCLUDE is a synonym for GET.

Syntax

GET filename

where:

filename is the name of the file to be included in the assembly. The assembler accepts pathnames in either UNIX or MS-DOS format.

Usage

GET is useful for including macro definitions, EQUs, and storage maps in an assembly. When assembly of the included file is complete, assembly continues at the line following the GET directive.

By default the assembler searches the current place for included files. The current place is the directory where the calling file is located. Use the -i assembler command-line option to add directories to the search path. File names and directory names containing spaces must not be enclosed in double quotes (" ").

The included file can contain additional GET directives to include other files (see Nesting directives on page 7-26).

If the included file is in a different directory from the current place, this becomes the current place until the end of the included file. The previous current place is then restored.

GET cannot be used to include object files (see INCBIN on page 7-68).

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 allowed
```
7.7.11 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 7-65, and EXPORT or GLOBAL on page 7-63).

Syntax

IMPORT symbol{[WEAK]}

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.

Usage

The name is resolved at link time to a symbol defined in a separate object file. The symbol is treated as a program address. If [WEAK] is not specified, the linker generates an error if no corresponding symbol is found at link time.

If [WEAK] is specified and no corresponding symbol is found at link time:

- If the reference is the destination of a B or BL instruction, the value of the symbol is taken as the address of the following instruction. This makes the B or BL instruction effectively a NOP.

- Otherwise, the value of the symbol is taken as zero.

To avoid trying to access symbols that are not found at link time, use code like the example in EXTERN on page 7-65.
7.7.12 INCBIN

The INCBIN directive includes a file within the file being assembled. The file is included as it is, without being assembled.

Syntax

INCBIN filename

where:

filename is the name of the file to be included in the assembly. The assembler accepts pathnames in either UNIX or MS-DOS format.

Usage

You can use INCBIN to include executable files, literals, or any arbitrary data. The contents of the file are added to the current ELF section, byte for byte, without being interpreted in any way. Assembly continues at the line following the INCBIN directive.

By default the assembler searches the current place for included files. The current place is the directory where the calling file is located. Use the -i assembler command-line option to add directories to the search path. File names and directory names containing spaces must not be enclosed in double quotes (" ").

Example

AREA Example, CODE, READONLY
INCBIN file1.dat ; includes file1 if it exists in the current place.
INCBIN c:\project\file2.txt ; includes file2
7.7.13 KEEP

The KEEP directive instructs the assembler to retain local symbols in the symbol table in the object file.

**Syntax**

KEEP {
  symbol
}

where:

*symbol* is the name of the local symbol to keep. If *symbol* is not specified, all local symbols are kept except register-relative symbols.

**Usage**

By default, the only symbols that the assembler describes in its output object file are:

- exported symbols
- symbols that are relocated against.

Use KEEP to preserve local symbols that can be used to help debugging. Kept symbols appear in the ARM debuggers and in linker map files.

KEEP cannot preserve register-relative symbols (see MAP on page 7-15).

**Example**

```assembly
label  ADC     r2,r3,r4
       KEEP label ; makes label available to debuggers
       ADD     r2,r2,r5
```
7.7.14 NOFP

The NOFP directive disallows floating-point instructions in an assembly language source file.

**Syntax**

NOFP

**Usage**

Use NOFP to ensure that no floating-point instructions are used in situations where there is no support for floating-point instructions either in software or in target hardware.

If a floating-point instruction occurs after the NOFP directive, an Unknown opcode error is generated and the assembly fails.

If a NOFP directive occurs after a floating-point instruction, the assembler generates the error:

Too late to ban floating point instructions

and the assembly fails.

7.7.15 REQUIRE

The REQUIRE directive specifies a dependency between sections.

**Syntax**

REQUIRE label

where:

`label` is the name of the required label.

**Usage**

Use REQUIRE to ensure that a related section is included, even if it is not directly called. If the section containing the REQUIRE directive is included in a link, the linker also includes the section containing the definition of the specified label.
7.7.16 REQUIRE8 and PRESERVE8

Some systems require eight-byte address alignment for STRD and LDRD instructions. If your code uses these instructions for reading from or writing to the stack, then you must ensure that the stack is eight-byte aligned whenever these instructions execute. This normally means that the stack must be eight-byte aligned at all software interfaces.

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 \{\textit{bool}\}
PRESERVE8 \{\textit{bool}\}

where:

\textit{bool} is an optional Boolean constant, either \{TRUE\} or \{FALSE\}. 
Usage

LDRD and STRD instructions (doubleword transfers) only work correctly if the address they access is eight-byte aligned.

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.

If your code preserves eight-byte alignment of the stack, use PRESERVE8 to set the PRES8 build attribute on your file. If your code does not preserve eight-byte alignment of the stack, use PRESERVE8 {FALSE} to ensure that the PRES8 build attribute is not set.

Note

If you omit both PRESERVE8 and PRESERVE8 {FALSE}, the assembler decides whether to set the PRES8 build attribute or not, by examining instructions that modify the sp. ARM recommends that you specify PRESERVE8 explicitly.

You can enable a warning with:

armasm --diag_warning 1546

See Command syntax on page 3-2 for details.

This gives you warnings like:

"test.s", line 37: Warning: A1546W: Stack pointer update potentially breaks 8 byte stack alignment

37 00000044 STMFD sp!,{r2,r3,lr}

Examples

REQUIRE8
REQUIRE8 {TRUE} ; equivalent to REQUIRE8
REQUIRE8 {FALSE} ; equivalent to absence of REQUIRE8
PRESERVE8 {TRUE} ; equivalent to PRESERVE8
PRESERVE8 {FALSE} ; NOT exactly equivalent to absence of PRESERVE8
The RN directive defines a register name for a specified register.

**Syntax**

\[ \text{name RN expr} \]

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
7.7.18 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 7-56).

Use the name option to ensure that each reference is to the correct local label. If the name of a label or a reference to a label does not match the preceding ROUT directive, the assembler generates an error message and the assembly fails.

Example

```
routineA    ROUT            ; ROUT is not necessarily a routine
; code
3routineA   ROUT            ; this label is checked
; code
   BEQ     %4routineA       ; this reference is checked
; code
   BGE     %3               ; refers to 3 above, but not checked
; code
4routineA   ROUT            ; this label is checked
; code
otherstuff  ROUT            ; start of next scope
```

# Glossary

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>AAPCS</strong></td>
<td><em>See Procedure Call Standard for The ARM Architecture.</em></td>
</tr>
<tr>
<td><strong>American National Standards Institute (ANSI)</strong></td>
<td>An organization that specifies standards for, among other things, computer software.</td>
</tr>
<tr>
<td><strong>Angel™</strong></td>
<td>Angel is a program that enables you to develop and debug applications running on ARM-based hardware. Angel can debug applications running in either ARM state or Thumb state.</td>
</tr>
<tr>
<td><strong>ANSI</strong></td>
<td><em>See American National Standards Institute.</em></td>
</tr>
<tr>
<td><strong>Architecture</strong></td>
<td>The term used to identify a group of processors that have similar characteristics.</td>
</tr>
<tr>
<td><strong>Big-endian</strong></td>
<td>Memory organization where the least significant byte of a word is at a higher address than the most significant byte.</td>
</tr>
<tr>
<td><strong>Byte</strong></td>
<td>A unit of memory storage consisting of eight bits.</td>
</tr>
<tr>
<td><strong>Canonical Frame Address (CFA)</strong></td>
<td>In DWARF 2, this is an address on the stack specifying where the call frame of an interrupted function is located.</td>
</tr>
<tr>
<td><strong>CFA</strong></td>
<td><em>See Canonical Frame Address.</em></td>
</tr>
<tr>
<td><strong>Coprocessor</strong></td>
<td>An additional processor that is used for certain operations. Usually used for floating-point math calculations, signal processing, or memory management.</td>
</tr>
<tr>
<td>Glossary</td>
<td></td>
</tr>
<tr>
<td>------------------------</td>
<td></td>
</tr>
<tr>
<td><strong>CPSR</strong></td>
<td>See Current Processor Status Register.</td>
</tr>
<tr>
<td><strong>Current place</strong></td>
<td>In compiler terminology, the directory that contains files to be included in the compilation process.</td>
</tr>
<tr>
<td><strong>Current Processor Status Register (CPSR)</strong></td>
<td>CPSR. A register containing the current state of control bits and flags. See also Saved Processor Status Register.</td>
</tr>
<tr>
<td><strong>Debugger</strong></td>
<td>An application that monitors and controls the execution of a second application. Usually used to find errors in the application program flow.</td>
</tr>
<tr>
<td><strong>Doubleword</strong></td>
<td>A 64-bit unit of information. Contents are taken as being an unsigned integer unless otherwise stated.</td>
</tr>
<tr>
<td><strong>DWARF</strong></td>
<td>Debug With Arbitrary Record Format.</td>
</tr>
<tr>
<td><strong>ELF</strong></td>
<td>Executable Linkable Format.</td>
</tr>
<tr>
<td><strong>Global variables</strong></td>
<td>Variables that are accessible to all code in the application. See also Local variables.</td>
</tr>
<tr>
<td><strong>Halfword</strong></td>
<td>A 16-bit unit of information. Contents are taken as being an unsigned integer unless otherwise stated.</td>
</tr>
<tr>
<td><strong>Image</strong></td>
<td>An executable file that has been loaded onto a processor for execution.</td>
</tr>
<tr>
<td></td>
<td>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.</td>
</tr>
<tr>
<td><strong>Interrupt</strong></td>
<td>A change in the normal processing sequence of an application caused by, for example, an external signal.</td>
</tr>
<tr>
<td><strong>Interworking</strong></td>
<td>Producing an application that uses both ARM and Thumb code.</td>
</tr>
<tr>
<td><strong>Library</strong></td>
<td>A collection of assembler or compiler output objects grouped together into a single repository.</td>
</tr>
<tr>
<td><strong>Linker</strong></td>
<td>Software that produces a single image from one or more source assembler or compiler output objects.</td>
</tr>
<tr>
<td><strong>Little-endian</strong></td>
<td>Memory organization where the least significant byte of a word is at a lower address than the most significant byte.</td>
</tr>
<tr>
<td><strong>Local variable</strong></td>
<td>A variable that is only accessible to the subroutine that created it. See also Global variables.</td>
</tr>
</tbody>
</table>
PIC
Position Independent Code.

See also ROPI.

PID
Position Independent Data or the ARM Platform-Independent Development card.

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 run-time.

Read Write Position Independent (RWPI)
Read/write data addresses can be changed at run-time.

ROPI
See Read Only Position Independent.

RVCT
See RealView Compilation Tools.

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.

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.
<table>
<thead>
<tr>
<th><strong>Glossary</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Stack</strong></td>
</tr>
<tr>
<td><strong>SWI</strong></td>
</tr>
<tr>
<td><strong>Target</strong></td>
</tr>
<tr>
<td><strong>Vector Floating Point (VFP)</strong></td>
</tr>
<tr>
<td><strong>Veneer</strong></td>
</tr>
<tr>
<td><strong>VFP</strong></td>
</tr>
<tr>
<td><strong>Word</strong></td>
</tr>
<tr>
<td><strong>Zero Initialized (ZI)</strong></td>
</tr>
<tr>
<td><strong>ZI</strong></td>
</tr>
</tbody>
</table>
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**
- Absolute addresses 3-21
- ADD instruction 2-60
- Addresses
  - loading into registers 2-32
- ADR
  - ARM pseudo-instruction 4-121, 4-122
  - Thumb pseudo-instruction 5-48
- ADR pseudo-instruction 2-32, 2-60
- ADR Thumb pseudo-instruction 2-32
- ADRL pseudo-instruction 2-32, 2-60
- ALIGN directive 2-58, 7-54
  - Alignment 2-58
- ALU status flags 2-22
- AND: operator 3-35
- AREA directive 2-14, 2-16, 7-56
- AREA directive (literal pools) 2-30
  - armasm
    - command syntax 3-2
- Assembly language
  - absolute addresses 3-21
  - alignment 2-58
  - base register 2-54
  - binary operators 3-35
  - block copy 2-46
  - Boolean constants 2-15
  - built-in variables 3-16
  - case rules 2-13
  - character constants 2-15
  - code size 2-63
  - comments 2-14
  - condition code suffixes 2-23
  - conditional execution 2-22
  - constants 2-15
  - coprocessor names 3-15
  - data structures 2-53
  - defining macros 7-27
  - ELF sections 2-16
  - entry point 2-17, 7-61
  - examples 2-2, 2-16, 2-18, 2-24, 2-30, 2-33, 2-37, 2-39, 2-46, 2-63, 2-65
  - examples, Thumb 2-19, 2-26, 2-40, 2-48
- execution speed 2-63
- floating-point literals 3-28
- format of source lines 3-14
- global variables 7-4, 7-7
- immediate constants, ARM 2-28
- jump tables 2-34
- labels 2-14, 3-21
- line format 2-13
- line length 2-13
- literal pools 2-30
- loading addresses 2-32
- loading constants 2-27
- local labels 2-14, 3-22
- logical expressions 3-29
- variables 3-19
- logical literals 3-29
- macros 2-50
- maintenance 2-58
- maps 2-53
- multiple register transfers 2-41
- multiplicative operators 3-35
- nesting subroutines 2-45
Index

numeric constants  2-15, 3-19
numeric expressions  3-26
numeric literals  3-27
numeric variables  3-19
operator precedence  3-30, 3-31
padding  2-58
pc  2-5, 2-42, 2-45, 2-48, 3-17, 3-21, 3-29
program counter  2-5, 3-17, 3-21, 3-29
program-relative  2-14
expressions  3-29
program-relative labels  3-21
program-relative maps  2-56
register names  3-15
register-based maps  2-55
register-relative expressions  3-29
labels  3-21
register-relative address  2-14
relational operators  3-37
relative maps  2-54
shift operators  3-36
speed  2-63
stacks  2-44
string expressions  3-25
manipulation  3-35
variables  3-19
string constants  2-15
string literals  3-25
subroutines  2-18
symbol naming rules  3-18
symbols  2-60, 3-18
Thumb block copy  2-48
unary operators  3-32
variable substitution  3-20
variables  3-19
built-in  3-16
global  7-4, 7-7
local  7-6, 7-7
VFP directives and notation  6-38
ASSERT directive  2-57, 2-67, 7-48

Barrel shifter  2-9, 2-22
Barrel shifter, Thumb  2-11
:BASE: operator  2-60, 3-32
Base register  2-54
Binary operators, assembly  3-35
BL instruction  2-18
BL instruction, Thumb  2-22
Block copy, assembly language  2-46
Block copy, Thumb  2-48
Boolean constants, assembly language  2-15
Branch instructions  2-7
Branch instructions, Thumb  2-11
BX instruction  2-20

C

Case rules, assembly language  2-13
Character constants, assembly language  2-15
:CHR: operator  3-32
CN directive  7-9
Code size  2-24, 2-63
CODE16 directive  2-19, 3-2, 7-59
CODE32 directive  2-19, 7-59
Command syntax
  armasm  3-2
Comments
  assembly language  2-14
Compiler options
  --diag_style  3-3, 3-11
  --fpmode  3-3
Condition code suffixes  2-23
Conditional execution, assembly  2-22, 2-24
Conditional execution, Thumb  2-10, 2-11
Constants, assembly  2-15
Coprocessor names, assembly  3-15
CP directive  7-10
CPSR  2-5, 2-22
Current Program Status Register  2-5

D

DATA directive  7-25
Data maps, assembly  2-53
Data processing instructions  2-7
Data processing instructions, Thumb  2-11
Data structure, assembly  2-53
DCB directive  7-18
DCDO directive  7-20
DCD, DCDU directives  7-19
DCFD, DCFDU directives  7-21
DCFS, DCFSU directives  7-22
DCI directive  7-23
DCQ, DCQU directives  7-24
DCW, DCWU directives  7-25
Diagnostic messages
  arm style  3-11
  ide style  3-11
  severity  3-11, 3-13
Directives, assembly language
  ALIGN  2-58, 7-54
  AREA  2-14, 2-16, 7-56
  AREA (literal pools)  2-30
  ASSERT  2-57, 2-67, 7-48
  CN  7-9
  CODE16  2-19, 3-2, 7-59
  CODE32  2-19, 7-59
  CP  7-10
  DATA  7-25
  DCB  7-18
  DCDO  7-20
  DCD, DCDU  7-19
  DCFD, DCFDU  7-21
  DCFS, DCFSU  7-22
  DCI  7-23
  DCQ, DCQU  7-24
  DCW, DCWU  7-25
  DN  7-11
  ELIF  7-30
  ELSE  7-30
  END  2-17, 7-60
  END (literal pools)  2-30
  ENDFUNC  7-47
  ENDIF  7-30
  ENDP  7-47
  ENTRY  2-17, 7-61
  EQU  3-19, 7-62
  EXPORT  7-63
  EXPORTAS  7-64
  EXTERN  7-65
  FIELD  7-16
  FN  7-12

Index-2
FRAME ADDRESS 7-35
FRAME POP 7-36
FRAME PUSH 7-37
FRAME REGISTER 7-38
FRAME RESTORE 7-39
FRAME RETURN ADDRESS 7-40
FRAME SAVE 7-41
FRAME STATE REMEMBER 7-42
FRAME STATE RESTORE 7-43, 7-44
FUNCTION 7-45
GBLA 3-10, 3-19, 7-4, 7-50
GBLL 3-10, 3-19, 7-4, 7-50
GBLS 3-10, 3-19, 7-4, 7-50
GET 3-3, 7-66
GLOBAL 7-63
IF 7-29, 7-30, 7-33
IMPORT 7-67
INCBIN 7-68
INCLUDE 3-3, 7-66
INFO 7-49
KEEP 7-69
LCLA 3-19, 7-6, 7-50
LCLL 3-19, 7-50
LCLS 3-19, 7-50
LTORG 7-14
MACRO 2-50, 7-27
MAP 2-53, 7-15
MEND 7-27, 7-50
MEXIT 7-29
nesting 7-26
NOFP 7-70
OPT 3-17, 7-50
PRESERVE8 7-71
PROC 7-45
REQUIRE 7-70
REQUIRE8 7-71
RLIST 3-2, 7-8
RN 7-73
ROUT 2-14, 3-22, 3-23, 7-74
SETA 3-4, 3-10, 3-17, 3-19, 7-7, 7-50
SETL 3-4, 3-10, 3-17, 3-19, 7-7, 7-50
SETS 3-4, 3-10, 3-17, 3-19, 7-7, 7-50
SN 7-11
SPACE 7-17
SUBT 7-52
TTL 7-52
VFPASSERT SCALAR 6-39
VFPASSERT VECTOR 6-40
WEND 7-33
WHILE 7-29, 7-33
! 7-49
# 7-16
% 7-17
& 7-19
* 7-62
= 7-18
[ 7-30
] 7-30
^ 7-15
| 7-30
DN directive 7-11
FRAME RESTORE directive 7-39
FRAME RETURN ADDRESS directive 7-40
FRAME SAVE directive 7-41
FRAME STATE REMEMBER directive 7-42
FRAME STATE RESTORE directive 7-43, 7-44
FUNCTION directive 7-45
GBLA directive 3-10, 3-19, 7-4, 7-50
GBLL directive 3-10, 3-19, 7-4, 7-50
GBLS directive 3-10, 3-19, 7-4, 7-50
GET directive 3-3, 7-66
GLOBAL directive 7-63
Halfwords
in load and store instructions 2-7
IF directive 7-29, 7-30, 7-33
Immediate constants, ARM 2-28
IMPORT directive 7-67
INCBIN directive 7-68
INCLUDE directive 3-3, 7-66
:INDEX: operator 2-60, 3-32
INFO directive 7-49
Instruction set
ARM 2-7
Thumb 2-10
Instructions, assembly language
ADD 2-60
BL 2-18
BX 2-20
LDM 2-41, 2-56, 3-2, 4-20, 7-8
LDM, Thumb 2-48
LDR 4-9, 4-14, 4-17
MOV 2-27, 2-28, 2-54
MRS 2-9
MSR 2-9
MVN 2-27, 2-28
Index

PLD  4-22
POP, Thumb  2-48
PUSH, Thumb  2-48
STM  2-41, 2-56, 3-2, 4-20, 7-8
STM, Thumb  2-48
STR  4-9, 4-14, 4-17
SWP  4-31
Invoke  3-2

J
Jump tables, assembly language   2-34

K
KEEP directive  7-69

L
Labels, assembly  3-21
Labels, local, assembly  3-22
LCLA directive  3-19, 7-6, 7-50
LCLL directive  3-19, 7-50
LCLS directive  3-19, 7-50
LDFD pseudo-instruction   6-36, 7-14
LDFS pseudo-instruction  7-14
LDM
   instruction  4-20
   instruction  2-41, 2-56, 3-2, 7-8
   Thumb   2-48
LDR
   instruction  4-9, 4-14, 4-17
   pseudo-instruction  2-27, 2-29, 2-37, 4-125
   relative maps  2-54
   Thumb pseudo-instruction   5-49
LDR pseudo-instruction   7-14
   literal pools  2-30
   loading constants  2-29, 2-37
   string copying  2-39
:LEFT: operator  3-35
:LEN: operator  3-32
Line format, assembly language   2-13
Line length, assembly language   2-13
Link register   2-5, 2-18

Linking
   assembly language labels   2-14
Literal pools, assembly language   2-30
Loading constants, assembly language   2-27
Local
   labels, assembly   3-22
   variables, assembly   7-6, 7-7
Local labels, assembly language   2-14
Logical
   expressions, assembly   3-29
   variable, assembly   3-19
Logical literals, assembly   3-29
LTORG directive  7-14

M
MACRO directive   2-50, 7-27
MAP directive   2-53, 7-15
Maps, assembly language
   program-relative   2-56
   register-based   2-55
   relative   2-54
MEND directive   7-27, 7-50
MEXIT directive   7-29
MOV instruction   2-27, 2-28, 2-54
MRS instruction   2-9
MSR instruction   2-9
Multiple register transfers   2-41
Multiplicative operators, assembly   3-35
MVN instruction   2-27, 2-28

N
Nesting directives   7-26
Nesting subroutines, assembly language   2-45
NOFP directive   7-70
NOP pseudo-instruction   4-121, 4-127
NOP Thumb pseudo-instruction   5-51
Numeric constants, assembly   3-19
Numeric constants, assembly language   2-15
Numeric expressions, assembly   3-26
numeric literals, assembly   3-27
Numeric variable, assembly   3-19

O
Operator precedence, assembly   3-30, 3-31
Operators, assembly language
   :BASE:   2-60
   :INDEX:   2-60
   :AND:   2-58
OPT directive   3-17, 7-50

P
Padding   2-58
Parameters, assembly macros   2-50
pc, assembly   3-17, 3-21, 3-29
pc, assembly language   2-5, 2-42, 2-45, 2-48
PLD
   instruction  4-22
   POP instruction, Thumb   2-48
   PRESERVE8 directive   7-71
   PROC directive   7-45
   Processor modes   2-4
   Program counter, assembly   3-17, 3-21, 3-29
   program counter, assembly language   2-5
   Program counter, Thumb   2-12
   Program-relative
      expressions   3-29
      labels   3-21
   Program-relative address   2-14
   Program-relative maps   2-56
   Prototype statement   2-50
   Pseudo-instructions, assembly language
      ADR   2-32, 2-60, 4-121, 4-122
      ADR (Thumb)   2-32, 5-48
      ADRL   2-32, 2-60
      LDFD   6-36, 7-14
      LDFS   7-14
      LDR
         2-27, 2-29, 2-37, 4-125
         (literal pools)   2-30
         (loading addresses)   2-37
         (loading constants)   2-29
         (Thumb)   5-49
      NOP   4-121, 4-127
      NOP (Thumb)   5-51
      PUSH instruction, Thumb   2-48

Index-4
Index

R

:RCONST: operator 3-32
Register
  names, assembly 3-15
Register access, Thumb 2-10
Register banks 2-4
Register-based
  symbols 2-60
Register-based maps 2-55
Register-relative
  expressions 3-29
Register-relative address 2-14
Register-relative labels 3-21
Registers 2-4
Relational operators, assembly 3-37
Relative maps 2-54
REQUIRE directive 7-70
REQUIRE8 directive 7-71
:RIGHT: operator 3-35
RLIST directive 3-2, 7-8
RN directive 7-73
ROUT directive 2-14, 3-22, 3-23, 7-74

S
Scope, assembly language 2-14
SETA directive 3-4, 3-10, 3-17, 3-19,
  7-7, 7-50
SETL directive 3-4, 3-10, 3-17, 3-19,
  7-7, 7-50
SETS directive 3-4, 3-10, 3-17, 3-19,
  7-7, 7-50
Shift operators, assembly 3-36
SN directive 7-11
SPACE directive 7-17
Stack pointer 2-4
Stacks, assembly language 2-44
Status flags 2-22
STM
  instruction 4-20
STM instruction 2-41, 2-56, 3-2, 7-8
  Thumb 2-48
STR
  instruction 4-9, 4-14, 4-17
  :STR: operator 3-32
String
  manipulation, assembly 3-35
  variable, assembly 3-19
String constants, assembly language 2-15
String literals, assembly 3-25
Subroutines, assembly language 2-18
SUBT directive 7-52
SWP
  instruction 4-31
Symbols
  assembly language 3-18
  assembly language, Naming rules 3-18
  Symbols, register-based 2-60

T
Thumb
  BX instruction 2-20
  conditional execution 2-22
  direct loading 2-29
  example assembly language 2-19
  instruction set 2-10
  LDM and STM instructions 2-48
  popping pc 2-45
  TTL directive 7-52

U
Unary operators, assembly 3-32

V
Variables, assembly 3-19
  built-in 3-16
  global 7-4, 7-7
  local 7-6, 7-7
  substitution 3-20
VFP directives and notation 6-38
VFPASSERT SCALAR directive 6-39
VFPASSERT VECTOR directive 6-40
Visual C++
  diagnostic message style 3-3, 3-11

W
WEAK symbol 7-65, 7-67
WEND directive 7-33
WHILE directive 7-29, 7-33

Symbols

! directive 7-49
# directive 7-16
% directive 7-17
& directive 7-19
* directive 7-62
= directive 7-18
[ directive 7-30
] directive 7-30
^ directive 7-15
| directive 7-30