ARM® Compiler

Version 6.00

armasm User Guide

ARM DUI 0801A (ID031214)

ARM Compiler
armasm User Guide

Release Information

The following changes have been made to this book.

Change History

Date	Issue	Confidentiality	Change
14 March 2014	A	Non-Confidential	ARM Compiler v6.00 Release

Product Status

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

Web Address

`http://www.arm.com`

Contents

ARM Compiler armasm User Guide

Chapter 1
Conventions and Feedback

The following describes the typographical conventions and how to give feedback:

Typographical conventions

The following typographical conventions are used:

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

Feedback on this product

If you have any comments and suggestions about this product, contact your supplier and give:

- your name and company

- the serial number of the product
- details of the release you are using
- details of the platform you are using, 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 content

If you have comments on content then send an e-mail to `errata@arm.com`. Give:

- the title
- the number, ARM DUI 0801A
- if viewing online, the topic names to which your comments apply
- if viewing a PDF version of a document, the page numbers to which your comments apply
- a concise explanation of your comments.

ARM also welcomes general suggestions for additions and improvements.

ARM periodically provides updates and corrections to its documentation on the ARM Information Center, together with knowledge articles and *Frequently Asked Questions* (FAQs).

Other information

- ARM Information Center `http://infocenter.arm.com/help/index.jsp`
- ARM Technical Support Knowledge Articles `http://infocenter.arm.com/help/topic/com.arm.doc.faqs/index.html`
- ARM Support and Maintenance `http://www.arm.com/support/services/support-maintenance.php`
- ARM Glossary `http://infocenter.arm.com/help/topic/com.arm.doc.aeg0014-/index.html`.

Chapter 2
Overview of armasm

The following topics introduce the assemblers provided with ARM® Compiler toolchain:

2.1 About the ARM Compiler toolchain assemblers

ARM Compiler toolchain provides:

- The freestanding legacy assembler, `armasm`. Use `armasm` to assemble existing A32 and T32 assembly language code.
- The `armclang` integrated assembler. Use this to assemble assembly language code written in GNU syntax.
- An optimizing inline assembler built into `armclang`. Use this to assemble assembly language code written in GNU syntax that is used inline in C or C++ source code.

2.1.1 See also

Concepts

Software Development Guide:

- Using inline assembly code
  `http://infocenter.arm.com/help/topic/com.arm.doc.dui0773-/chr1383748162225.html`
- Assembling assembly code
  `http://infocenter.arm.com/help/topic/com.arm.doc.dui0773-/chr1382606255397.html`.

2.2 Key features of the assembler

`armasm` supports:

- *Unified Assembly Language* (UAL) for both A32 and T32 code.
- Assembly language for A64 code.
- Advanced SIMD instructions in A64, A32, and T32 code.
- Floating-point instructions in A64, A32, and T32 code.
- Directives in assembly source code.
- Processing of user defined macros.

2.2.1 See also

Concepts

- *How the assembler works* on page 2-4
- *Unified Assembler Language* on page 7-3
- *Advanced SIMD* on page 3-5
- *Use of macros* on page 7-31
- *Architecture support for Advanced SIMD and floating-point* on page 11-3.

Getting Started Guide:

- About ARMv8 terminology
  `http://infocenter.arm.com/help/topic/com.arm.doc.dui0741-/chr1375353547214.html`

Reference

armasm Reference Guide:

- Chapter 4 *Advanced SIMD and Floating-point Programming (32-bit)*
- Chapter 10 *Directives Reference*.

2.3 How the assembler works

`armasm` is a 2 pass assembler that outputs object code from the assembly language source code. This means that it reads the source code twice. Each read of the source code is called a pass.

This is because assembly language source code often contains forward references. A forward reference occurs when a label is used as an operand, for example as a branch target, earlier in the code than the definition of the label. The assembler cannot know the address of the forward reference label until it reads the definition of the label. During each pass, the assembler performs different functions.

During the first pass, the assembler:

- Checks the syntax of the instruction or directive. It faults if there is an error in the syntax, for example if a label is specified on a directive that does not accept one.
- Determines the size of the instruction and data being assembled and reserves space.
- Determines offset of labels within sections.
- Creates a symbol table containing label definitions and their memory addresses.

During the second pass, the assembler:

- Faults if an undefined reference is specified in an instruction operand or directive.
- Encodes the instructions using the label offsets from pass 1, where applicable.
- Generates relocations.
- Generates debug information if requested.
- Outputs the object file.

Memory addresses of labels are determined and finalized in the first pass. Therefore, the assembly code must not change during the second pass. All instructions must be seen in both passes. Therefore you must not define a symbol after a `:DEF:` test for the symbol. The assembler faults if it sees code in pass 2 that was not seen in pass 1. Example 2-1 shows that `num EQU 42` is not seen in pass 1 but is seen in pass 2.

Example 2-1 Line not seen in pass 1

```
    AREA x,CODE
    [ :DEF: foo
num EQU 42
    ]
foo DCD num
    END
```

Assembling the code in Example 2-1 generates the error:

```
A1903E: Line not seen in first pass; cannot be assembled.
```

Example 2-2 on page 2-5 shows that `MOV r1,r2` is seen in pass 1 but not in pass 2.

Example 2-2 Line not seen in pass 2

```
    AREA x,CODE
    [ :LNOT: :DEF: foo
    MOV r1, r2
    ]
foo MOV r3, r4
    END
```

Assembling the code in Example 2-2 generates the error:

```
A1909E: Line not seen in second pass; cannot be assembled.
```

2.3.1 See also

Concepts

- *Directives that can be omitted in pass 2 of the assembler* on page 2-6
- *Two pass assembler diagnostics* on page 9-18
- *Instruction and directive relocations* on page 7-35.

Reference

armasm Reference Guide:

- *--diag_error* on page 2-20
- *--debug* on page 2-17.

2.4 Directives that can be omitted in pass 2 of the assembler

Most directives must appear in both passes of the assembly process. There are a number of directives that can be omitted from pass 2, but doing this is strongly discouraged. Directives that can be omitted from pass 2 are:

- `GBLA`, `GBLL`, `GBLS`
- `LCLA`, `LCLL`, `LCLS`
- `SETA`, `SETL`, `SETS`
- `RN`, `RLIST`
- `CN`, `CP`
- `SN`, `DN`, `QN`
- `EQU`
- `MAP`, `FIELD`
- `GET`, `INCLUDE`
- `IF`, `ELSE`, `ELIF`, `ENDIF`
- `WHILE`, `WEND`
- `ASSERT`
- `ATTR`
- `COMMON`
- `EXPORTAS`
- `IMPORT`
- `EXTERN`
- `KEEP`
- `MACRO`, `MEND`, `MEXIT`
- `REQUIRE8`
- `PRESERVE8`.

Note

Macros that appear only in pass 1 and not in pass 2 must contain only these directives.

For example, the code in Example 2-3 assembles without error although the `ASSERT` directive does not appear in pass 2.

Example 2-3 `ASSERT` directive appears in pass 1 only

```
    AREA ||.text||,CODE
x   EQU 42
    IF :LNOT: :DEF: sym
        ASSERT x == 42
    ENDIF
sym EQU 1
    END
```

Directives that appear in pass 2 but do not appear in pass 1 cause an assembly error. However, this does not cause an assembly error when using the `ELSE` and `ELIF` directives if their matching `IF` directive appears in pass 1. Example 2-4 on page 2-7 assembles without error because the `IF` directive appears in pass 1.

Example 2-4 Use of `ELSE` and `ELIF` directives

```
    AREA ||.text||,CODE
x   EQU 42
    IF :DEF: sym
    ELSE
        ASSERT x == 42
    ENDIF
sym EQU 1
    END
```

2.4.1 See also

Concepts

- *How the assembler works* on page 2-4
- *Two pass assembler diagnostics* on page 9-18
- *Instruction and directive relocations* on page 7-35.

Chapter 3
Overview of the ARM Architecture

The following topics give an overview of the ARMv8 architecture:

3.1 About the ARM architecture

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

ARMv8 is the next major architectural update after ARMv7. It introduces a 64-bit architecture, but maintains compatibility with existing 32-bit architectures. It uses two execution states:

AArch32 In AArch32 state, code has access to 32-bit general purpose registers.

Code executing in AArch32 state can only use the A32 and T32 instruction sets. This state is broadly compatible with the ARMv7-A architecture.

AArch64 In AArch64 state, code has access to 64-bit general purpose registers. The AArch64 state exists only in the ARMv8 architecture.

Code executing in AArch64 state can only use the A64 instruction set.

In the AArch32 execution state, there are the following instruction set states:

A32 state	The state that executes A32 instructions.
T32 state	The state that executes T32 instructions.

——— **Note** ———

Detailed information about the ARMv8 architecture is available under license. Contact your ARM Account Representative for details.

———————————

3.1.1 See also

Concepts

- *A64, A32, and T32 instruction sets* on page 3-3.

Other information

- *ARMv8-A Architecture Reference Manual*
  `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0487a.b/index.html`.

3.2 A64, A32, and T32 instruction sets

The A32 instruction set is a set of 32-bit instructions providing a comprehensive range of operations.

ARMv4T and later define a 16-bit instruction set called Thumb, or T32. Most of the functionality of the 32-bit A32 instruction set is available, but some operations require more instructions. The T32 instruction set provides better code density, at the expense of performance.

ARMv6T2 introduces Thumb-2 technology. This is a major enhancement to the T32 instruction set by providing 32-bit T32 instructions. The 32-bit and 16-bit T32 instructions together provide almost exactly the same functionality as the A32 instruction set. This version of the T32 instruction set achieves the high performance of A32 code along with the benefits of better code density.

ARMv8 introduces a new set of 32-bit instructions called A64, with new encodings and assembly language. A64 is only available when the processor is in AArch64 state. It provides similar functionality to the A32 and T32 instruction sets, but gives access to a larger virtual address space, and has some other changes, including less conditionality.

In ARMv8, the A32 and T32 instruction sets are largely unchanged from ARMv7. They are only available when the processor is in AArch32 state. The main changes in ARMv8 are the addition of a few new instructions and the deprecation of some behavior, including many uses of the `IT` instruction.

ARMv8 also defines an optional Crypto Extension, which provides cryptographic and hash instructions in both the A32 and A64 instruction sets.

Note

- The term A32 is an alias for the ARM instruction set.
- The term T32 is an alias for the Thumb instruction set.

3.2.1 See also

Concepts

- *A32 instruction set overview* on page 4-15
- *A64 instruction set overview* on page 5-13
- *About the ARM architecture* on page 3-2.

Other information

- *ARMv8-A Architecture Reference Manual*
  `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0487a.b/index.html`.

3.3 Changing between AArch64 and AArch32 states

A processor that is executing A64 instructions is operating in AArch64 state. In this state, the instructions can access both the 64-bit and 32-bit registers.

A processor that is executing A32 or T32 instructions is operating in AArch32 state. In this state, the instructions can only access the 32-bit registers, and not the 64-bit registers.

A processor based on ARMv8 can run applications built for AArch32 and AArch64 states but a change between AArch32 and AArch64 states can only happen at exception boundaries.

ARM Compiler toolchain builds images for either the AArch32 state or AArch64 state. Therefore, an image built with ARM Compiler toolchain can either contain only A32 and T32 instructions or only A64 instructions.

A processor can only execute instructions from the instruction set that matches its current execution state. A processor in AArch32 state cannot execute A64 instructions, and a processor in AArch64 state cannot execute A32 or T32 instructions. You must ensure that the processor never receives instructions from the wrong instruction set for the current execution state.

3.3.1 See also

Concepts

- *About the ARM architecture* on page 3-2
- *Exception levels* on page 5-3.

3.4 Advanced SIMD

Advanced SIMD is a 64-bit and 128-bit hybrid *Single Instruction Multiple Data* (SIMD) technology targeted at advanced media and signal processing applications and embedded processors. It is implemented as part of the ARM core, but has its own execution pipelines and a register bank that is distinct from the ARM core register bank.

Advanced SIMD instructions are available in both A32 and A64. The A64 Advanced SIMD instructions are based on those in A32. The main differences are the following:

- Different instruction mnemonics and syntax.
- Thirty-two 128-bit vector registers, increased from sixteen in AArch32 state.
- A different register packing scheme. In AArch64 state, smaller registers occupy the low order bits of larger registers. For example, S31 maps to bits[31:0] of D31. In AArch32 state, smaller registers are packed into larger registers. For example, S31 maps to bits[63:32] of D15.
- A64 Advanced SIMD instructions support both single-precision and double-precision floating-point data types and arithmetic. A32 Advanced SIMD instructions support only single-precision floating-point data types.

3.4.1 See also

Concepts

- *Architecture support for Advanced SIMD and floating-point* on page 11-3
- *Views of the Advanced SIMD register bank in AArch32 state* on page 11-8
- Chapter 11 *Advanced SIMD and Floating-point Programming*.

3.5 Floating-point hardware

The floating-point hardware, together with associated support code, provides single-precision and double-precision floating-point arithmetic, as defined by *IEEE Std. 754-2008 IEEE Standard for Floating-Point Arithmetic*. This document is referred to as the IEEE 754 standard.

The floating-point hardware uses a register bank that is distinct from the ARM core register bank.

Note

The floating-point register bank is shared with the SIMD register bank.

In AArch32 state, floating-point support is largely unchanged from VFPv4, apart from the addition of a few instructions for compliance with the IEEE 754 standard.

The floating-point architecture in AArch64 state is also based on VFPv4. The main differences are the following:

- In AArch64 state, the number of 128-bit SIMD and floating-point registers increases from sixteen to thirty-two.
- Single-precision registers are no longer packed into double-precision registers, so register S*x* is D*x*[31:0].
- The presence of floating-point hardware is mandated, so software floating-point linkage is not supported.
- Earlier versions of the floating-point architecture, for instance VFPv2, VFPv3, and VFPv4, are not supported in AArch64 state.
- VFP vector mode is not supported in either AArch32 or AArch64 state. Use Advanced SIMD instructions for vector floating-point.
- Some new instructions have been added, including:
  - Direct conversion between half-precision and double-precision.
  - Load and store pair, replacing load and store multiple.
  - Fused multiply-add and multiply-subtract.
  - Instructions for IEEE 754-2008 compatibility.

3.5.1 See also

Concepts

- *Architecture support for Advanced SIMD and floating-point* on page 11-3
- *Views of the floating-point extension register bank in AArch32 state* on page 11-10
- *Views of the floating-point extension register bank in AArch64 state* on page 11-11
- Chapter 11 *Advanced SIMD and Floating-point Programming*.

Chapter 4
Overview of AArch32 state

The following topics give an overview of the AArch32 state of ARMv8:

4.1 Changing between A32 and T32 state

A processor that is executing A32 instructions is operating in *A32 state*. A processor that is executing T32 instructions is operating in *T32 state*. These are called *instruction set states*.

A processor in A32 state cannot execute T32 instructions, and a processor in T32 state cannot execute A32 instructions. You must ensure that the processor never receives instructions of the wrong instruction set for the current state.

The initial state after reset depends on the processor being used and its configuration.

To direct `armasm` to generate A32 or T32 instruction encodings, you must set the assembler mode using an `ARM` or `THUMB` directive. Assembly code using `CODE32` and `CODE16` directives can still be assembled, but ARM recommends you use `ARM` and `THUMB` for new code.

These directives do not change the instruction set state of the processor. To do this, you must use an appropriate instruction, for example `BX` or `BLX` to change between A32 and T32 states when performing a branch.

4.1.1 See also

Reference

armasm Reference Guide:

- *B, BL, BX, BLX, and BXJ* on page 3-44
- *Instruction set and syntax selection directives* on page 10-9.

4.2 Processor modes, and privileged and unprivileged software execution

ARM processors support the processor modes shown in Table 4-1.

Table 4-1 ARM processor modes

Processor mode	Mode number
User	0b10000
FIQ	0b10001
IRQ	0b10010
Supervisor	0b10011
Abort	0b10111
Undefined	0b11011
System	0b11111
Monitor	0b10110
Hypervisor	0b11010

User mode is an unprivileged mode, and has restricted access to system resources. All other modes have full access to system resources in the current security state, can change mode freely, and execute software as privileged.

Applications that require task protection usually execute in User mode. Some embedded applications might run entirely in any mode other than User mode. An application that requires full access to system resources usually executes in System mode.

Modes other than User mode are entered to service exceptions, or to access privileged resources.

Code can run in either a secure state or in a non-secure state. Hypervisor (Hyp) mode has privileged execution in non-secure state.

4.2.1 See also

Other information

- *ARM Architecture Reference Manual* `http://infocenter.arm.com/help/topic/com.arm.doc.subset.architecture.reference/`.

4.3 Registers in AArch32 state

In all ARM processors in AArch32 state, the following registers are available and accessible in any processor mode:

- 15 general-purpose registers R0-R12, the *Stack Pointer* (SP), and *Link Register* (LR).
- 1 *Program Counter* (PC).
- 1 *Application Program Status Register* (APSR).

Note

- SP and LR can be used as general-purpose registers, although ARM deprecates using SP other than as a stack pointer.

Additional registers are available in privileged software execution. ARM processors have a total of 43 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.

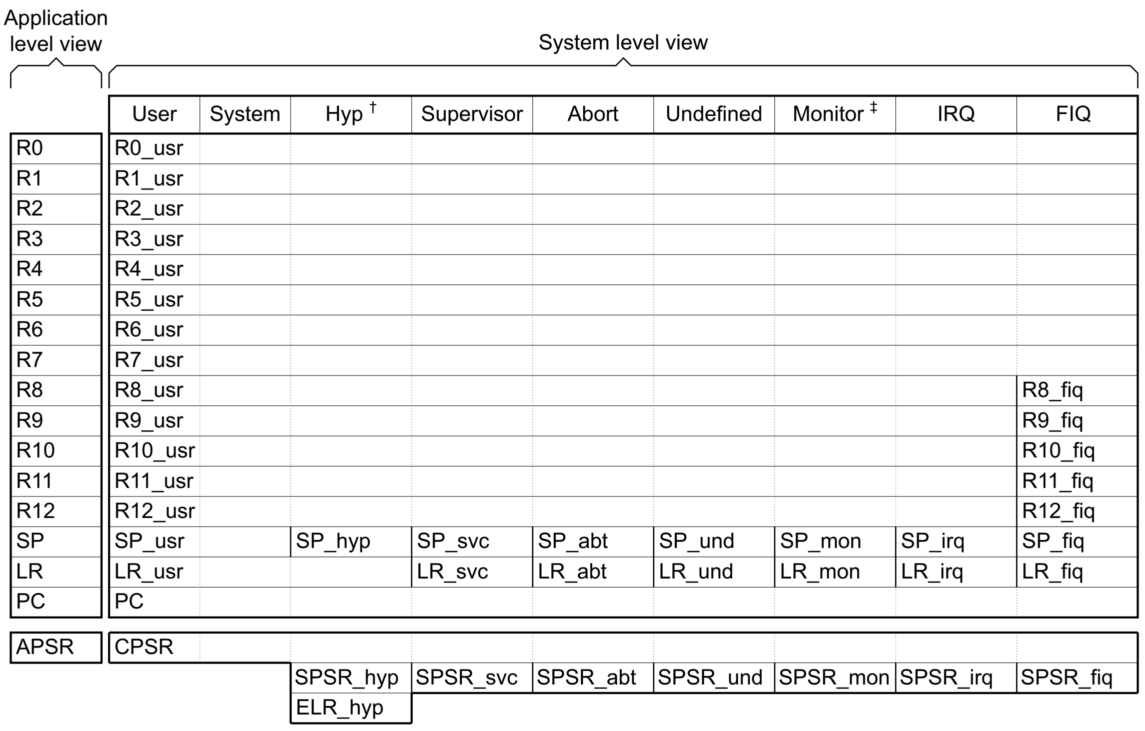
The additional registers in ARM processors are:

- 2 supervisor mode registers for banked SP and LR.
- 2 abort mode registers for banked SP and LR.
- 2 undefined mode registers for banked SP and LR.
- 2 interrupt mode registers for banked SP and LR.
- 7 FIQ mode registers for banked R8-R12, SP and LR.
- 2 monitor mode registers for banked SP and LR.
- 1 Hyp mode register for banked SP.
- 7 *Saved Program Status Register* (SPSRs), one for each exception mode.
- 1 Hyp mode register for ELR_Hyp to store the preferred return address from Hyp mode.

Note

In privileged software execution, CPSR is an alias for APSR and gives access to additional bits.

Figure 4-1 on page 4-5 shows how the registers are banked in the ARM architecture.



‡ Exists only in Secure state.

† Exists only in Non-secure state.

Figure 4-1 Organization of general-purpose registers and Program Status Registers

4.3.1 See also

Concepts

- *General-purpose registers in AArch32 state* on page 4-6
- *Program Counter in AArch32 state* on page 4-10
- *Application Program Status Register* on page 4-11
- *Saved Program Status Registers (SPSRs) in AArch32 state* on page 4-14
- *Current Program Status Register in AArch32* on page 4-13
- *Processor modes, and privileged and unprivileged software execution* on page 4-3
- *Registers in AArch64 state* on page 5-2.

Other information

- *ARM Architecture Reference Manual*
  `http://infocenter.arm.com/help/topic/com.arm.doc.subset.architecture.reference/`.

4.4 General-purpose registers in AArch32 state

There are 33 general-purpose 32-bit registers, including the banked SP and LR registers. Fifteen general-purpose registers are visible at any one time, depending on the current processor mode. These are R0-R12, SP, and LR. The PC (R15) is not considered a general-purpose register.

SP (or R13) is the *stack pointer*. The C and C++ compilers always use SP as the stack pointer. ARM deprecates most uses of SP as a general purpose register. In T32 state, SP is strictly defined as the stack pointer. The instruction pages in the *armasm Reference Guide* describe when SP and PC can be used.

In User mode, LR (or R14) is used as a *link register* to store the return address when a subroutine call is made. It can also be used as a general-purpose register if the return address is stored on the stack.

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

4.4.1 See also

Concepts

- *Program Counter in AArch32 state* on page 4-10
- *Register accesses in AArch32 state* on page 4-7
- *Predeclared core register names in AArch32 state* on page 4-8
- *Link registers* on page 5-4
- *Stack Pointer register* on page 5-5.

Reference

armasm Reference Guide:

- *MRS (PSR to general-purpose register)* on page 3-100
- *MSR (general-purpose register to PSR)* on page 3-103.

4.5 Register accesses in AArch32 state

Most 16-bit T32 instructions can only access R0 to R7. Only a small number of T32 instructions can access R8-R12, SP, LR, and PC. Registers R0 to R7 are called Lo registers. Registers R8-R12, SP, LR, and PC are called Hi registers.

All 32-bit T32 instructions can access R0 to R12, and LR. However, apart from a few designated stack manipulation instructions, most T32 instructions cannot use SP. Except for a few specific instructions where PC is useful, most T32 instructions cannot use PC.

In A32 state, all instructions can access R0 to R12, SP, and LR, and most instructions can also access PC (R15). However, the use of the SP in an A32 instruction, in any way that is not possible in the corresponding T32 instruction, is deprecated. Explicit use of the PC in an A32 instruction is not usually useful, and except for specific instances that are useful, such use is deprecated. Implicit use of the PC, for example in branch instructions or load (literal) instructions, is never deprecated.

The `MRS` instructions can move the contents of a status register to a general-purpose register, where they can be manipulated by normal data processing operations. You can use the `MSR` instruction to move the contents of a general-purpose register to a status register.

4.5.1 See also

Concepts

- *General-purpose registers in AArch32 state* on page 4-6
- *Program Counter in AArch32 state* on page 4-10
- *Application Program Status Register* on page 4-11
- *Current Program Status Register in AArch32* on page 4-13
- *Saved Program Status Registers (SPSRs) in AArch32 state* on page 4-14
- *Predeclared core register names in AArch32 state* on page 4-8
- *Read-Modify-Write procedure* on page 7-29.

Reference

armasm Reference Guide:

- *MRS (PSR to general-purpose register)* on page 3-100
- *MSR (general-purpose register to PSR)* on page 3-103.

4.6 Predeclared core register names in AArch32 state

Table 4-2 shows the predeclared core registers:

Table 4-2 Predeclared core registers in AArch32 state

Register names	Meaning
`R0-R15`	General purpose registers.
`a1-a4`	Argument, result or scratch registers. These are synonyms for R0 to R3.
`v1-v8`	Variable registers. These are synonyms for R4 to R11.
`SB`	Static base register. This is a synonym for R9.
`IP`	Intra-procedure call scratch register. This is a synonym for R12.
`SP`	Stack pointer. This is a synonym for R13.
`LR`	Link register. This is a synonym for R14.
`PC`	Program counter. This is a synonym for R15.

With the exception of `a1-a4` and `v1-v8`, you can write the register names either in all upper case or all lower case.

4.6.1 See also

Concepts

- *General-purpose registers in AArch32 state* on page 4-6
- *Predeclared core register names in AArch64 state* on page 5-6.

4.7 Predeclared extension register names in AArch32 state

Table 4-3 shows the predeclared extension register names:

Table 4-3 Predeclared extension registers in AArch32 state

Register names	Meaning
`Q0-Q15`	Advanced SIMD quadword registers
`D0-D31`	Advanced SIMD doubleword registers, floating-point double-precision registers
`S0-S31`	Floating-point single-precision registers

You can write the register names either in upper case or lower case.

4.7.1 See also

Concepts

- *Extension register bank mapping in AArch32 state* on page 11-4
- *Predeclared extension register names in AArch64 state* on page 5-7.

4.8 Program Counter in AArch32 state

The *Program Counter* (PC) is accessed as PC (or R15). It is incremented by the size of the instruction executed, which is always four bytes in A32 state. Branch instructions load the destination address into the PC. You can also load the PC directly using data operation instructions. For example, to branch to the address in a general purpose register, use:

```
MOV PC,R0
```

During execution, the PC does not contain the address of the currently executing instruction. The address of the currently executing instruction is typically PC–8 for A32, or PC–4 for T32.

——— **Note** ———

ARM recommends you use the `BX` instruction to jump to an address or to return from a function, rather than writing to the PC directly.

———————————

4.8.1 See also

Concepts

- *Register-relative and PC-relative expressions* on page 10-7
- *Program Counter in AArch64 state* on page 5-8.

Reference

- *B, BL, BX, BLX, and BXJ* on page 3-44.

4.9 Application Program Status Register

The *Application Program Status Register* (APSR) holds copies of the N, Z, C, and V *condition flags*. The processor uses them to determine whether or not to execute conditional instructions.

The APSR also holds the Q (saturation) flag and the GE (Greater than or Equal) flags. The GE flags can be set by the parallel add and subtract instructions. They are used by the `SEL` instruction to perform byte-based selection from two registers.

Note

The APSR exists only in AArch32 state. In AArch64 state, the Q and GE flags cannot be read or written to, and the condition flags are held in a special-purpose register called the NZCV register.

These flags are accessible in all modes, using the `MSR` and `MRS` instructions.

4.9.1 See also

Concepts

- *Updates to the condition flags in A32/T32 code* on page 8-7
- *Conditional execution in AArch64 state* on page 5-9
- *Conditional instructions* on page 8-2.

Reference

armasm Reference Guide:

- *MRS (PSR to general-purpose register)* on page 3-100
- *MSR (general-purpose register to PSR)* on page 3-103
- *SEL* on page 3-126
- *Parallel add and subtract* on page 3-109.

4.10 The Q flag in AArch32 state

The Q flag is set to 1 when saturation occurs in saturating arithmetic instructions, or when overflow occurs in certain multiply instructions.

The Q flag is a *sticky* flag. Although the saturating and certain multiply instructions can set the flag, they cannot clear it. You can execute a series of such instructions, and then test the flag to find out whether saturation or overflow occurred at any point in the series, without having to check the flag after each instruction.

To clear the Q flag, use an `MSR` instruction to read-modify-write the APSR:

```
MRS r5, APSR
BIC r5, r5, #(1<<27)
MSR APSR_nzcvq, r5
```

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.

```
MRS r6, APSR
TST r6, #(1<<27); Z is clear if Q flag was set
```

4.10.1 See also

Concepts

- *Read-Modify-Write procedure* on page 7-29
- *The Q flag in AArch64 state* on page 5-10.

Reference

armasm Reference Guide:

- *MRS (PSR to general-purpose register)* on page 3-100
- *MSR (general-purpose register to PSR)* on page 3-103
- *QADD, QSUB, QDADD, and QDSUB* on page 3-118
- *SMULxy and SMLAxy* on page 3-141
- *SMULWy and SMLAWy* on page 3-140.

4.11 Current Program Status Register in AArch32

The *Current Program Status Register* (CPSR) holds:

- The APSR flags.
- The processor mode.
- The interrupt disable flags.
- The instruction set state (A32 or T32).
- The endianness state.
- The execution state bits for the IT block.

The execution state bits control conditional execution in the IT block.

Only the APSR flags are accessible in all modes. ARM deprecates using an `MSR` instruction to change the endianness bit (E) of the CPSR, in any mode. Each exception level can have its own endianness, but mixed endianness within an exception level is deprecated. The `SETEND` instruction is deprecated in A32 and T32 and has no equivalent in A64.

The execution state bits for the IT block (IT[1:0]) and the T32 bit (T) can be accessed by `MRS` only in Debug state.

4.11.1 See also

Concepts

- *Updates to the condition flags in A32/T32 code* on page 8-7
- *Saved Program Status Registers (SPSRs) in AArch32 state* on page 4-14
- *Process State* on page 5-11.

Reference

armasm Reference Guide:

- *IT* on page 3-63
- *SETEND* on page 3-128
- *MSR (general-purpose register to PSR)* on page 3-103
- *MRS (PSR to general-purpose register)* on page 3-100.

4.12 *Saved Program Status Registers* (SPSRs) in AArch32 state

The SPSR stores the current value of the CPSR when an exception is taken so that it can be restored after handling the exception. Each exception handling mode can access its own SPSR. User mode and System mode do not have an SPSR because they are not exception handling modes.

The execution state bits, including the endianness state and current instruction set state can be accessed from the SPSR in any exception mode, using the `MSR` and `MRS` instructions. You cannot access the SPSR using `MSR` or `MRS` in User or System mode.

4.12.1 See also

Concepts

- *Current Program Status Register in AArch32* on page 4-13
- *Saved Program Status Registers (SPSRs) in AArch64 state* on page 5-12.

4.13 A32 instruction set overview

All A32 instructions are 32 bits long. Instructions are stored word-aligned, so the least significant two bits of instruction addresses are always zero in A32 state.

T32 instructions are either 16 or 32 bits long. Instructions are stored half-word aligned. Some instructions use the least significant bit of the address to determine whether the code being branched to is T32 or A32.

Before the introduction of 32-bit T32 instructions, the T32 instruction set was limited to a restricted subset of the functionality of the A32 instruction set. Almost all T32 instructions were 16-bit. Together, the 32-bit and 16-bit T32 instructions provide functionality that is almost identical to that of the A32 instruction set.

Table 4-4 describes some of the functional groupings of the available instructions.

Table 4-4 A32 instruction groups

Instruction group	Description
Branch and control	These instructions do the following: • Branch to subroutines. • Branch backwards to form loops. • Branch forward in conditional structures. • Make the following instruction conditional without branching. • Change the processor between A32 state and T32 state.
Data processing	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 an immediate value supplied within the instruction. Long multiply instructions give a 64-bit result in two registers.
Register load and store	These instructions load or store the value of a single register from or to memory. They can load or store a 32-bit word, a 16-bit halfword, or an 8-bit unsigned byte. Byte and halfword loads can either be sign extended or zero extended to fill the 32-bit register. A few instructions are also defined that can load or store 64-bit doubleword values into two 32-bit registers.
Multiple register load and store	These instructions load or store any subset of the general-purpose registers from or to memory.
Status register access	These instructions move the contents of a status register to or from a general-purpose register.

4.13.1 See also

Concepts

- *Load and store multiple register instructions* on page 7-21
- *A64 instruction set overview* on page 5-13.

Reference

armasm Reference Guide:

- Chapter 3 *A32 and T32 Instructions*.

4.14 Media processing instructions

Media processing instructions were introduced in the media extension for ARMv6. The following media processing instructions are available:

Parallel add and subtract instructions:

- *Parallel add and subtract* on page 3-109.

Extend instructions:

- *SXT, SXTA, UXT, and UXTA* on page 3-152.

Multiply instructions:

- *SMLAD and SMLSD* on page 3-132
- *SMLALD and SMLSLD* on page 3-135
- *SMMUL, SMMLA, and SMMLS* on page 3-137
- *SMUAD{X} and SMUSD{X}* on page 3-139.

Packing, unpacking, saturation, and reversal instructions:

- *PKHBT and PKHTB* on page 3-112
- *REV, REV16, REVSH, and RBIT* on page 3-120
- *SEL* on page 3-126
- *SSAT and USAT* on page 3-145
- *SSAT16 and USAT16* on page 3-147.

Absolute sum and bit field instructions:

- *BFC and BFI* on page 3-47
- *SBFX and UBFX* on page 3-124
- *USAD8 and USADA8* on page 3-162.

4.15 Access to the inline barrel shifter in AArch32 state

The ARM arithmetic logic unit has a 32-bit barrel shifter that is capable of shift and rotate operations. The second operand to many A32 and T32 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 an immediate value.
- Constructing immediate values.

32-bit T32 instructions give almost the same access to the barrel shifter as A32 instructions.

16-bit T32 instructions only allow access to the barrel shifter using separate instructions.

4.15.1 See also

Concepts

- *Load immediates into registers* on page 7-6
- *Load immediate values using MOV and MVN* on page 7-7.

Chapter 5
Overview of AArch64 state

The following topics give an overview of the AArch64 state of ARMv8:

5.1 Registers in AArch64 state

In AArch64 state, the following registers are available:

- Thirty-one 64-bit general-purpose registers X0-X30, the bottom halves of which are accessible as W0-W30.
- Four stack pointer registers SP_EL0, SP_EL1, SP_EL2, SP_EL3.
- Three exception link registers ELR_EL1, ELR_EL2, ELR_EL3.
- Three saved program status registers SPSR_EL1, SPSR_EL2, SPSR_EL3.
- One program counter.

All these registers are 64 bits wide except SPSR_EL1, SPSR_EL2, and SPSR_EL3, which are 32 bits wide.

Most A64 integer instructions can operate on either 32-bit or 64-bit registers. The register width is determined by the register identifier, where W means 32-bit and X means 64-bit. The names W*n* and X*n*, where *n* is in the range 0-30, refer to the same register. When you use the 32-bit form of an instruction, the upper 32 bits of the source registers are ignored and the upper 32 bits of the destination register are set to zero.

There is no register named W31 or X31. Depending on the instruction, register 31 is either the stack pointer or the zero register. When used as the stack pointer, you refer to it as SP. When used as the zero register, you refer to it as WZR in a 32-bit context or XZR in a 64-bit context.

5.1.1 See also

Concepts

- *Registers in AArch32 state* on page 4-4
- *Link registers* on page 5-4
- *Program Counter in AArch64 state* on page 5-8
- *Conditional execution in AArch64 state* on page 5-9
- *Saved Program Status Registers (SPSRs) in AArch64 state* on page 5-12
- *Process State* on page 5-11
- *Exception levels* on page 5-3
- *Stack Pointer register* on page 5-5.

5.2 Exception levels

ARMv8 defines four exception levels, EL0 to EL3, where EL3 is the highest exception level with the most execution privilege. When taking an exception, the exception level can either increase or remain the same, and when returning from an exception, it can either decrease or remain the same.

The following is a common usage model for the exception levels:

EL0	Applications.
EL1	OS kernels and associated functions that are typically described as privileged.
EL2	Hypervisor.
EL3	Secure monitor.

When taking an exception to a higher exception level, the execution state can either remain the same, or change from AArch32 to AArch64.

When returning to a lower exception level, the execution state can either remain the same or change from AArch64 to AArch32.

The only way the execution state can change is by taking or returning from an exception. It is not possible to change between execution states, as between A32 and T32 code in AArch32 state.

On powerup and on reset, the processor enters the highest implemented exception level. The execution state for this exception level is a property of the implementation, and might be determined by a configuration input signal.

For exception levels other than EL0, the execution state is determined by one or more control register configuration bits. These bits can be set only in a higher exception level.

For EL0, the execution state is determined as part of the exception return to EL0, under the control of the exception level that the execution is returning from.

5.2.1 See also

Concepts

- *Link registers* on page 5-4
- *Saved Program Status Registers (SPSRs) in AArch64 state* on page 5-12
- *Changing between AArch64 and AArch32 states* on page 3-4
- *Process State* on page 5-11.

5.3 Link registers

In AArch64 state, the Link Register (LR) stores 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. The LR maps to register 30. Unlike in AArch32 state, the LR is distinct from the Exception Link Registers (ELRs) and is therefore unbanked.

There are three Exception Link Registers, ELR_EL1, ELR_EL2, and ELR_EL3, that correspond to each of the exception levels. When an exception is taken, the Exception Link Register for the target exception level stores the return address to jump to after the handling of that exception completes. If the exception was taken from AArch32 state, the top 32 bits in the ELR are all set to zero. Subroutine calls within the exception level use the LR to store the return address from the subroutine.

For example when the exception level changes from EL0 to EL1, the return address is stored in ELR_EL1.

When in an exception level, if you enable interrupts that use the same exception level, you must ensure you store the ELR on the stack because it will be overwritten with a new return address when the interrupt is taken.

5.3.1 See also

Concepts

- *Program Counter in AArch64 state* on page 5-8
- *Predeclared core register names in AArch64 state* on page 5-6
- *General-purpose registers in AArch32 state* on page 4-6.

5.4 Stack Pointer register

In AArch64 state, SP represents the 64-bit Stack Pointer. SP_EL0 is an alias for SP. Do not use SP as a general purpose register. You can only use SP as an operand in the following instructions:

- As the base register for loads and stores. In this case it must be quadword-aligned before adding any offset, or a stack alignment exception occurs.
- As a source or destination for arithmetic instructions, but it cannot be used as the destination in instructions that set the condition flags.
- In logical instructions, for example in order to align it.

There is a separate stack pointer for each of the three exception levels, SP_EL1, SP_EL2, and SP_EL3. Within an exception level you can either use the dedicated stack pointer for that exception level or you can use SP_EL0, the stack pointer associated with EL0. You can use the SPSel register to select which stack pointer to use in the exception level.

The choice of stack pointer is indicated by the letter t or h appended to the exception level name, for example EL0t or EL3h. The t suffix indicates that the exception level uses SP_EL0 and the h suffix indicates it uses SP_EL*x*, where *x* is the current exception level number. EL0 always uses SP_EL0 so cannot have an h suffix.

5.4.1 See also

Concepts

- *General-purpose registers in AArch32 state* on page 4-6
- *Registers in AArch64 state* on page 5-2
- *Process State* on page 5-11
- *Exception levels* on page 5-3.

5.5 Predeclared core register names in AArch64 state

In AArch64 state, the predeclared core registers are different from those in AArch32 state.

Table 5-1 shows the predeclared core registers in AArch64 state:

Table 5-1 Predeclared core registers in AArch64 state

Register names	Meaning
`W0-W30`	32-bit general purpose registers.
`X0-X30`	64-bit general purpose registers.
`WZR`	32-bit RAZ/WI register. This is the name for register 31 when it is used as the zero register in a 32-bit context.
`XZR`	64-bit RAZ/WI register. This is the name for register 31 when it is used as the zero register in a 64-bit context.
`WSP`	32-bit stack pointer. This is the name for register 31 when it is used as the stack pointer in a 32-bit context.
`SP`	64-bit stack pointer. This is the name for register 31 when it is used as the stack pointer in a 64-bit context.
`LR`	Link register. This is a synonym for X30.

You can write the register names either in all upper case or all lower case.

Note

In AArch64 state, the PC is not a general purpose register and you cannot access it by name.

5.5.1 See also

Concepts

- *Predeclared core register names in AArch32 state* on page 4-8
- *Registers in AArch64 state* on page 5-2
- *Link registers* on page 5-4
- *Stack Pointer register* on page 5-5
- *Program Counter in AArch64 state* on page 5-8.

5.6 Predeclared extension register names in AArch64 state

Table 5-2 shows the predeclared extension register names in AArch64 state:

Table 5-2 Predeclared extension registers in AArch64 state

Register names	Meaning
`V0-V31`	Advanced SIMD 128-bit vector registers.
`Q0-Q31`	Advanced SIMD registers holding a 128-bit scalar.
`D0-D31`	Advanced SIMD registers holding a 64-bit scalar, floating-point double-precision registers.
`S0-S31`	Advanced SIMD registers holding a 32-bit scalar, floating-point single-precision registers.
`H0-H31`	Advanced SIMD registers holding a 16-bit scalar, floating-point half-precision registers.
`B0-B31`	Advanced SIMD registers holding an 8-bit scalar.

You can write the register names either in upper case or lower case.

5.6.1 See also

Concepts

- *Predeclared extension register names in AArch32 state* on page 4-9
- *Extension register bank mapping in AArch64 state* on page 11-6
- *Registers in AArch64 state* on page 5-2.

5.7 Program Counter in AArch64 state

In AArch64 state, the *Program Counter* (PC) contains the address of the currently executing instruction. It is incremented by the size of the instruction executed, which is always four bytes.

In AArch64 state, the PC is not a general purpose register and you cannot access it explicitly. The following types of instructions read it implicitly:

- Instructions that compute a PC-relative address.
- PC-relative literal loads.
- Direct branches to a PC-relative label.
- Branch and link instructions, which store it in the procedure link register.

The only types of instructions that can write to the PC are:

- Conditional and unconditional branches.
- Exception generation and exception returns.

Branch instructions load the destination address into the PC.

5.7.1 See also

Concepts

- *Program Counter in AArch32 state* on page 4-10
- *Register-relative and PC-relative expressions* on page 10-7.

Reference

- *B, BL, BX, BLX, and BXJ* on page 3-44.

5.8 Conditional execution in AArch64 state

In AArch64 state, the NZCV register holds copies of the N, Z, C, and V *condition flags*. The processor uses them to determine whether or not to execute conditional instructions. The NZCV register contains the flags in bits[31:28].

The condition flags are accessible in all exception levels, using the `MSR` and `MRS` instructions.

A64 makes less use of conditionality than A32. For example, in A64:

- Only a few instructions can set or test the condition flags.
- There is no equivalent of the T32 `IT` instruction.
- The only conditionally executed instruction, which behaves as a NOP if the condition is false, is the conditional branch, `B.`*`cond`*.

5.8.1 See also

Concepts

- *Application Program Status Register* on page 4-11
- *Updates to the condition flags in A32/T32 code* on page 8-7
- *Updates to the condition flags in A64 code* on page 8-8
- *Conditional instructions* on page 8-2.

Reference

armasm Reference Guide:

- *MRS (PSR to general-purpose register)* on page 3-100
- *MSR (general-purpose register to PSR)* on page 3-103.

5.9 The Q flag in AArch64 state

In AArch64 state, you cannot read or write to the Q flag because in A64 there are no saturating arithmetic instructions that operate on the general purpose registers.

The Advanced SIMD saturating arithmetic instructions set the QC bit in the floating-point status register (FPSR) to indicate that saturation has occurred. You can identify such instructions by the Q mnemonic modifier, for example `SQADD`.

5.9.1 See also

Concepts

- *The Q flag in AArch32 state* on page 4-12.

Reference

armasm Reference Guide:

- *A64 Advanced SIMD scalar instructions in alphabetical order* on page 8-2
- *A64 Advanced SIMD vector instructions in alphabetical order* on page 9-2.

5.10 Process State

In AArch64 state, there is no *Current Program Status Register* (CPSR). You can access the different components of the traditional CPSR independently as the following *Process State* fields:

- N, Z, C, and V condition flags (NZCV).
- Current register width (nRW).
- Stack pointer selection bit (SPSel).
- Interrupt disable flags (DAIF).
- Current exception level (EL).
- Single step process state bit (SS).
- Illegal exception return state bit (IL).

You can use `MSR` to write to:

- The N, Z, C, and V flags in the NZCV register.
- The interrupt disable flags in the DAIF register.
- The SP selection bit in the SPSel register, in EL1 or higher.

You can use `MRS` to read:

- The N, Z, C, and V flags in the NZCV register.
- The interrupt disable flags in the DAIF register.
- The exception level bits in the CurrentEL register, in EL1 or higher.
- The SP selection bit in the SPSel register, in EL1 or higher.

When an exception occurs, all Process State fields associated with the current exception level are stored in a single register associated with the target exception level, the SPSR. You can access the SS, IL, and nRW bits only from the SPSR.

5.10.1 See also

Concepts

- *Current Program Status Register in AArch32* on page 4-13
- *Saved Program Status Registers (SPSRs) in AArch32 state* on page 4-14
- *Updates to the condition flags in A32/T32 code* on page 8-7
- *Updates to the condition flags in A64 code* on page 8-8
- *Saved Program Status Registers (SPSRs) in AArch64 state* on page 5-12.

Reference

armasm Reference Guide:

- *MSR (general-purpose register to PSR)* on page 3-103
- *MRS (PSR to general-purpose register)* on page 3-100.

5.11 *Saved Program Status Registers* (SPSRs) in AArch64 state

The SPSRs are 32-bit registers that store the process state of the current exception level when an exception is taken to an exception level that uses AArch64 state. This allows the process state to be restored after the exception has been handled.

In AArch64 state, each target exception level has its own SPSR:

- SPSR_EL1.
- SPSR_EL2.
- SPSR_EL3.

When taking an exception, the process state of the current exception level is stored in the SPSR of the target exception level. On returning from an exception, the exception handler uses the SPSR of the exception level that is being returned from to restore the process state of the exception level that is being returned to.

Note

On returning from an exception, the preferred return address is restored from the ELR associated with the exception level that is being returned from.

The SPSRs store the following information:

- N, Z, C, and V flags.
- D, A, I, and F interrupt disable bits.
- The register width.
- The execution mode.
- The IL and SS bits.

5.11.1 See also

Concepts

- *Stack Pointer register* on page 5-5
- *Process State* on page 5-11
- *Saved Program Status Registers (SPSRs) in AArch32 state* on page 4-14.

5.12 A64 instruction set overview

Table 5-3 describes some of the functional groupings of the instructions in A64.

Table 5-3 A64 instruction groups

Instruction group	Description
Branch and control	These instructions do the following: • Branch to and return from subroutines. • Branch backwards to form loops. • Branch forward in conditional structures. • Generate and return from exceptions.
Data processing	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 an immediate value supplied within the instruction. The addition and subtraction instructions can optionally left shift the immediate operand, or can sign or zero-extend and shift the final source operand register. A64 includes signed and unsigned 32-bit and 64-bit multiply and divide instructions.
Register load and store	These instructions load or store the value of a single register or pair of registers from or to memory. You can load or store a single 64-bit doubleword, 32-bit word, 16-bit halfword, or 8-bit byte, or a pair of words or doublewords. Byte and halfword loads can either be sign-extended or zero-extended to fill the 32-bit register. You can also load and sign-extend a signed byte, halfword or word into a 64-bit register, or load a pair of signed words into two 64-bit registers.
System register access	These instructions move the contents of a system register to or from a general-purpose register.

5.12.1 See also

Concepts

- *A32 instruction set overview* on page 4-15.

Reference

armasm Reference Guide:

- *A64 general instructions in alphabetical order* on page 5-2
- *A64 data transfer instructions in alphabetical order* on page 6-2.

Chapter 6
Structure of Assembly Language Modules

Assembly language is the language that the assembler (`armasm`) parses and assembles to produce object code. The following topics describe the structure of the assembly source files:

6.1 Syntax of source lines in assembly language

The general form of source lines in assembly language is:

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

All three sections of the source line are optional.

`symbol` is usually a label. 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. It cannot contain any white space character such as a space or a tab unless it is enclosed by bars (|).

Labels are symbolic representations of addresses. You can use labels to mark specific addresses that you want to refer to from other parts of the code. Numeric local labels are a subclass of labels that begin with a number in the range 0-99. Unlike other labels, a numeric local label can be defined many times. This makes them useful when generating labels with a macro.

Directives provide important information to the assembler that either affects the assembly process or affects the final output image.

Instructions and pseudo-instructions make up the code a processor uses to perform tasks.

Note

Instructions, pseudo-instructions, and directives must be preceded by white space, such as a space or a tab, irrespective of whether there is a preceding label or not.

Some directives do not allow the use of a label.

A comment is the final part of a source line. The first semicolon on a line marks the beginning of a comment except where the semicolon appears inside a string literal. The end of the line is the end of the comment. A comment alone is a valid line. The assembler ignores all comments. You can use blank lines to make your code more readable.

Instruction mnemonics, pseudo-instructions, directives, and symbolic register names (except `a1-a4 and v1-v8` in A32 instructions) can be written in all uppercase or all lowercase, but not mixed. Labels and comments can be in uppercase or lowercase, or mixed.

Example 6-1

```
        AREA     A32ex, CODE, READONLY
                                ; Name this block of code A32ex
        ENTRY                   ; Mark first instruction to execute
start
        MOV      r0, #10        ; Set up parameters
        MOV      r1, #3
        ADD      r0, r0, r1     ; r0 = r0 + r1
stop
        MOV      r0, #0x18      ; angel_SWIreason_ReportException
        LDR      r1, =0x20026   ; ADP_Stopped_ApplicationExit
        SVC      #0x123456      ; A32 semihosting (formerly SWI)
        END                     ; Mark end of file
```

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

Note

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

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

6.1.1 See also

Concepts

- *Labels* on page 10-8
- *Numeric local labels* on page 10-12
- *Symbol naming rules* on page 10-3
- *Numeric literals* on page 10-17
- *String literals* on page 10-15
- *Literals* on page 6-4
- *Syntax differences between UAL and A64 assembly language* on page 7-4.

6.2 Literals

In assembly source files, literals can be expressed as:

- Decimal numbers, for example `123`.
- Hexadecimal numbers, for example `0x7B`.
- Numbers in any base from `2` to `9`, for example `5_204` is a number in base `5`.
- Floating-point numbers, for example `123.4`.
- Boolean values `{TRUE}` or `{FALSE}`.
- Single character values enclosed by single quotes, for example `‘w’`.
- Strings enclosed in double quotes, for example `"This is a string"`.

Note

In most cases, a string containing a single character is accepted as a single character value. For example `ADD r0,r1,#”a”` is accepted, but `ADD r0,r1,#”ab”` is faulted.

You can also use variables and names to represent the literals.

6.2.1 See also

Concepts

- *Syntax of source lines in assembly language* on page 6-2.

6.3 ELF sections and the AREA directive

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

The output of an assembly or compilation can include:

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

The linker places each section in a program image according to section placement rules. Sections that are adjacent in source files are not necessarily adjacent in the application image

In a source file, the `AREA` directive marks the start of a section. This directive names the section and sets its attributes. The attributes are placed after the name, separated by commas.

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

Example 6-2 defines a single read-only section called `A32ex` that contains code.

Example 6-2

```
        AREA     A32ex, CODE, READONLY
                                ; Name this block of code A32ex
```

6.3.1 See also

Concepts

- *An example assembly language module* on page 6-6.

armlink User Guide:

- Chapter 8 *Using scatter files*.

Reference

armasm Reference Guide:

- *AREA* on page 10-14.

6.4 An example assembly language module

Example 6-3 and Example 6-4 illustrate some of the core constituents of an assembly language module. They are written in A32 and A64 assembly language respectively.

The constituent parts of these examples are:

- ELF sections (defined by the AREA directive)
- Application entry (defined by the ENTRY directive).
- Application execution.
- Application termination.
- Program end (defined by the END directive).

Example 6-3 defines a single section called A32ex that contains code and is marked as being READONLY. This example uses the A32 instruction set.

Example 6-3 Constituents of an A32 assembly language module

```
        AREA     A32ex, CODE, READONLY
                               ; Name this block of code A32ex
        ENTRY                  ; Mark first instruction to execute
start
        MOV      r0, #10       ; Set up parameters
        MOV      r1, #3
        ADD      r0, r0, r1    ; r0 = r0 + r1
stop
        MOV      r0, #0x18     ; angel_SWIreason_ReportException
        LDR      r1, =0x20026  ; ADP_Stopped_ApplicationExit
        SVC      #0x123456     ; A32 semihosting (formerly SWI)
        END                    ; Mark end of file
```

Example 6-4 defines a single section called A64ex that contains code and is marked as being READONLY. This example uses the A64 instruction set.

Example 6-4 Constituents of an A64 assembly language module

```
        AREA     A64ex, CODE, READONLY
                               ; Name this block of code A64ex
        ENTRY                  ; Mark first instruction to execute
start
        MOV      w0, #10       ; Set up parameters
        MOV      w1, #3
        ADD      w0, w0, w1    ; w0 = w0 + w1
stop
        MOV      x1, #0x26
        MOVK     x1, #2, LSL #16
        STR      x1, [sp,#0]   ; ADP_Stopped_ApplicationExit
        MOV      x0, #0
        STR      x0, [sp,#8]   ; Exit status code
        MOV      x1, sp        ; x1 contains the address of parameter block
        MOV      w0, #0x18     ; angel_SWIreason_ReportException
        HLT      0xf000        ; AArch64 semihosting
        END                    ; Mark end of file
```

6.4.1 Application entry

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

6.4.2 Application execution

The application code begins executing at the label `start`, where it loads the decimal values 10 and 3 into registers `R0` and `R1` or `W0` and `W1`. These registers are added together and the result placed in `R0` or `W0`.

6.4.3 Application termination

After executing the main code, the application terminates by returning control to the debugger. You do this in A32 using the A32 semihosting SVC (`0x123456` by default), or in A64, using `HLT 0xF000` to invoke the semihosting interface.

A32 code uses the following parameters:

- R0 equal to `angel_SWIreason_ReportException` (`0x18`).
- R1 equal to `ADP_Stopped_ApplicationExit` (`0x20026`).

A64 code uses the following parameters:

- W0 equal to `angel_SWIreason_ReportException` (`0x18`).
- X1 is the address of a block of 2 parameters. The first is the exception type, `ADP_Stopped_ApplicationExit` (`0x20026`) and the second is the exit status code.

6.4.4 Program end

The `END` 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. Any lines following the `END` directive are ignored by the assembler.

6.4.5 See also

Concepts

- *ELF sections and the AREA directive* on page 6-5.

Reference

armasm Reference Guide:

- *ENTRY* on page 10-35
- *END* on page 10-33.

Chapter 7
Writing A32/T32 Assembly Language

The following topics describe the use of a few basic A32 and T32 instructions and the use of macros:

7.1 Unified Assembler Language

Unified Assembler Language (UAL) is a common syntax for A32 and T32 instructions. It supersedes earlier versions of both the ARM and Thumb assembler languages.

Code written using UAL can be assembled for A32 or T32 for any ARM processor. `armasm` faults the use of unavailable instructions.

`armasm` can assemble code written in pre-UAL and UAL syntax.

By default, `armasm` expects source code to be written in UAL. `armasm` accepts UAL syntax if any of the directives `CODE32`, `ARM`, or `THUMB` is used or if you assemble with any of the `--32`, `--arm`, or `--thumb` command-line options. `armasm` also accepts source code written in pre-UAL ARM assembly language when you assemble with `CODE32` or `ARM`.

`armasm` accepts source code written in pre-UAL Thumb assembly language when you assemble using the `--16` command line option, or the `CODE16` directive in the source code.

——— **Note** ———

The pre-UAL Thumb assembly language does not support 32-bit T32 instructions.

———————————

7.2 Syntax differences between UAL and A64 assembly language

UAL is the assembler syntax used by the A32 and T32 instruction sets. A64 assembly language is the assembler syntax used by the A64 instruction set.

UAL in ARMv8 is unchanged from ARMv7.

The general statement format and operand order of A64 assembly language is the same as that of UAL, but there are some differences between them. Table 7-1 describes the main differences:

Table 7-1 Syntax differences between UAL and A64 assembly language

UAL	A64
You make an instruction conditional by appending a condition code suffix directly to the mnemonic, with no delimiter. For example: `BEQ label`	For conditionally executed instructions, you separate the condition code suffix from the mnemonic using a `.` delimiter. For example: `B.EQ label`
Apart from the `IT` instruction, there are no unconditionally executed integer instructions that use a condition code as an operand.	A64 provides several unconditionally executed instructions that use a condition code as an operand. For these instructions, you specify the condition code to test for in the final operand position. For example: `CSEL w1,w2,w3,EQ`
The `.W` and `.N` instruction width specifiers control whether the assembler generates a 32-bit or 16-bit encoding for a T32 instruction.	A64 is a fixed width 32-bit instruction set so does not support `.W` and `.N` qualifiers.
The core register names are R0-R15.	You must qualify register names to indicate the operand's data size, either 32-bit (W0-W31) or 64-bit (X0-X31).
You can refer to registers R13, R14 and R15 as synonyms for SP, LR, and PC respectively.	In AArch64, there is no register named W31 or X31. Instead, you can refer to register 31 as SP, WZR or XZR, depending on the context. You cannot refer to PC either by name or number. LR is an alias for register 30.
A32 has no equivalent of the extend operators.	You can specify an extend operator in several instructions to control how a portion of the second source register value is sign or zero extended. For example, in the following instruction, `UXTB` is the extend type (zero extend, byte) and `#2` is an optional left shift amount: `ADD X1, X2, W3, UXTB #2`

7.2.1 See also

Concepts

- *Syntax of source lines in assembly language* on page 6-2
- *Instruction width selection in T32 code* on page 9-21
- *Differences between A32/T32 and A64 Advanced SIMD and floating-point instruction syntax* on page 11-12.

7.3 Subroutine calls

A subroutine is a block of code that performs a task based on some arguments and optionally returns a result. By convention, you use registers R0 to R3 to pass arguments to subroutines, and R0 to pass a result back to the callers. A subroutine that requires more than 4 inputs uses the stack for the additional inputs.

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 PC-relative expression.

The `BL` instruction:

- Places the return address in the link register.
- Sets the PC to the address of the subroutine.

After the subroutine code is executed you can use a `BX LR` instruction to return.

Note

Calls between separately assembled or compiled modules must comply with the restrictions and conventions defined by the Procedure Call Standard for the ARM Architecture.

Example 7-1 shows a subroutine, `doadd`, that adds the values of two arguments and returns a result in `R0`.

Example 7-1 Add two arguments

```
        AREA    subrout, CODE, READONLY     ; Name this block of code
        ENTRY                          ; Mark first instruction to execute
start   MOV     r0, #10                ; Set up parameters
        MOV     r1, #3
        BL      doadd                  ; Call subroutine
stop    MOV     r0, #0x18              ; angel_SWIreason_ReportException
        LDR     r1, =0x20026           ; ADP_Stopped_ApplicationExit
        SVC     #0x123456              ; ARM semihosting
doadd   ADD     r0, r0, r1             ; Subroutine code
        BX      lr                     ; Return from subroutine
        END                            ; Mark end of file
```

7.3.1 See also

Concepts

- *Stack operations for nested subroutines* on page 7-25.
- *Register-relative and PC-relative expressions* on page 10-7.

Reference

- *B, BL, BX, BLX, and BXJ* on page 3-44.

Other information

- *Procedure Call Standard for the ARM Architecture*
  `http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042-/index.html`.

7.4 Load immediates into registers

You can load an immediate from a range of values into a register using a `MOV` or `MVN` instruction. The range depends on the instruction set. Certain 32-bit values cannot be represented as an immediate operand to a single 32-bit instruction, although these values can be loaded from memory in a single instruction.

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

You can also use the `LDR` pseudo-instruction to load immediate values into a register.

You can include many commonly-used immediate values directly as operands within data processing instructions, without a separate load operation. The range of immediate values that you can include as operands in 16-bit T32 instructions is much smaller.

7.4.1 See also

Concepts

- *Load immediate values using MOV and MVN* on page 7-7
- *Load 32-bit values to a register using MOV32* on page 7-10
- *Load immediate 32-bit values to a register using LDR Rd, =const* on page 7-11
- *Load values to SIMD and floating-point registers* on page 11-14.

7.5 Load immediate values using MOV and MVN

In A32 state:

- `MOV` can load any 8-bit immediate value, giving a range of `0x0`-`0xFF` (0-255).

  It can also rotate these values by any even number.

  These values are also available as immediate operands in many data processing operations, without being loaded in a separate instruction.

- `MVN` can load the bitwise complements of these values. The numerical values are `-(`*n*`+1)`, where *n* is the value available in `MOV`.

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

Table 7-2 shows the range of 8-bit values that can be loaded in a single A32 `MOV` or `MVN` instruction (for data processing operations). The value to load must be a multiple of the value shown in the Step column.

Table 7-3 shows the range of 16-bit values that can be loaded in a single `MOV` A32 instruction.

Table 7-2 A32 state immediate values (8-bit)

Binary	Decimal	Step	Hexadecimal	MVN value[a]	Notes
`000000000000000000000000abcdefgh`	0-255	1	`0-0xFF`	–1 to –256	-
`0000000000000000000000abcdefgh00`	0-1020	4	`0-0x3FC`	–4 to –1024	-
`00000000000000000000abcdefgh0000`	0-4080	16	`0-0xFF0`	–16 to –4096	-
`000000000000000000abcdefgh000000`	0-16320	64	`0-0x3FC0`	–64 to –16384	-
...	...	...	...	...	-
`abcdefgh000000000000000000000000`	0-255 x 2^{24}	2^{24}	`0-0xFF000000`	1-256 x $–2^{24}$	-
`cdefgh000000000000000000000000ab`	(bit pattern)	-	-	(bit pattern)	See b in Note
`efgh000000000000000000000000abcd`	(bit pattern)	-	-	(bit pattern)	See b in Note
`gh000000000000000000000000abcdef`	(bit pattern)	-	-	(bit pattern)	See b in Note

Table 7-3 A32 state immediate values in MOV instructions

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

Note

These notes give extra information on Table 7-2 and Table 7-3.

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

b These values are available in A32 state only. All the other values in this table are also available in 32-bit T32 instructions.

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

In T32 state:

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

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

The 16-bit T32 `MOV` instruction can load any immediate value in the range 0-255.

Table 7-4 shows the range of values that can be loaded in a single 32-bit T32 `MOV` or `MVN` instruction (for data processing operations). The value to load must be a multiple of the value shown in the Step column.

Table 7-5 shows the range of 16-bit values that can be loaded by the `MOV` 32-bit T32 instruction.

Table 7-4 32-bit T32 immediate values

Binary	Decimal	Step	Hexadecimal	MVN value[a]	Notes
`000000000000000000000000abcdefgh`	0-255	1	`0-0xFF`	–1 to –256	-
`00000000000000000000000abcdefgh0`	0-510	2	`0-0x1FE`	–2 to –512	-
`0000000000000000000000abcdefgh00`	0-1020	4	`0-0x3FC`	–4 to –1024	-
...	...	...	...	...	-
`0abcdefgh00000000000000000000000`	0-255 x 2^{23}	2^{23}	`0-0x7F800000`	1-256 x -2^{23}	-
`abcdefgh000000000000000000000000`	0-255 x 2^{24}	2^{24}	`0-0xFF000000`	1-256 x -2^{24}	-
`abcdefghabcdefghabcdefghabcdefgh`	(bit pattern)	-	`0xXYXYXYXY`	`0xXYXYXYXY`	-
`00000000abcdefgh00000000abcdefgh`	(bit pattern)	-	`0x00XY00XY`	`0xFFXYFFXY`	-
`abcdefgh00000000abcdefgh00000000`	(bit pattern)	-	`0xXY00XY00`	`0xXYFFXYFF`	-
`00000000000000000000abcdefghijkl`	0-4095	1	`0-0xFFF`	-	See b in Note

Table 7-5 32-bit T32 immediate values in MOV instructions

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

Note

These notes give extra information on Table 7-4 and Table 7-5.

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

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

In both A32 and T32, you do not have to decide whether to use `MOV` or `MVN`. `armasm` uses whichever is appropriate. This is useful if the value is an assembly-time variable.

If you write an instruction with an immediate value that is not available, `armasm` reports the error: `Immediate` *`n`* `out of range for this operation.`

7.5.1 See also

Concepts

- *Load immediates into registers* on page 7-6.

7.6 Load 32-bit values to a register using MOV32

Both A32 and T32 instruction sets include:

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

You can use these two instructions to construct any 32-bit immediate value in a register. Alternatively, you can use the `MOV32` pseudo-instruction. `armasm` generates the `MOV`, `MOVT` instruction pair for you.

You can also use the `MOV32` pseudo-instruction to load addresses into registers by using a label or any PC-relative expression in place of an immediate value. `armasm` puts a relocation directive into the object file for the linker to resolve the address at link-time.

7.6.1 See also

Concepts

- *Register-relative and PC-relative expressions* on page 10-7.

Reference

armasm Reference Guide:

- *MOV32 pseudo-instruction* on page 3-97.

7.7 Load immediate 32-bit values to a register using LDR Rd, =const

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

The `LDR` pseudo-instruction generates the most efficient single instruction for the specified immediate value:

- If the immediate value can be constructed with a single `MOV` or `MVN` instruction, `armasm` generates the appropriate instruction.
- If the immediate value cannot be constructed with a single `MOV` or `MVN` instruction, `armasm`:
  - 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 PC-relative address that reads the constant from the literal pool.

  For example:

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

  You must ensure that there is a literal pool within range of the `LDR` instruction generated by `armasm`.

7.7.1 See also

Concepts

- *Literal pools* on page 7-12.

Reference

armasm Reference Guide:

- *LDR pseudo-instruction* on page 3-83.

7.8 Literal pools

armasm uses literal pools to hold certain constant values that are to be loaded into registers. armasm places a literal pool at the end of each section. The end of a section is defined either by the END directive at the end of the assembly or by the AREA directive at the start of the following section. 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 A32 or T32 code, when the 32-bit LDR instruction is available, but can be in either direction.
- Forward and less than 1KB when only the 16-bit T32 LDR instruction is available.

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

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

If the next literal pool is out of range, armasm 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 the valid range for an LDR instruction.

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 7-2 shows how this works.

The instructions listed as comments are the A32 instructions generated by armasm.

Example 7-2 Placing literal pools

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

7.8.1 See also

Concepts

- *Load immediate 32-bit values to a register using LDR Rd, =const* on page 7-11.

Reference

armasm Reference Guide:

- *LTORG* on page 10-66.

7.9 Load addresses into registers

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

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

You can load an address into a register either:

- Using the instruction `ADR`.
- Using the instruction `ADRL`.
- Using the instruction `MOV32`.
- From a literal pool using the pseudo-instruction `LDR Rd, =Label`.

7.9.1 See also

Concepts

- *Load addresses to a register using ADR* on page 7-15
- *Load addresses to a register using ADRL* on page 7-17
- *Load 32-bit values to a register using MOV32* on page 7-10
- *Load addresses to a register using LDR Rd, =label* on page 7-18.

7.10 Load addresses to a register using ADR

The ADR instruction enables you to generate an address, within a certain range, without performing a data load. ADR accepts a PC-relative expression, that is, a label with an optional offset where the address of the label is relative to the current PC.

Note

The label used with ADR must be within the same code section. armasm faults references to labels that are out of range in the same section.

The available range of addresses for the ADR instruction depends on the instruction set and encoding:

A32 Any value that can be produced by rotating an 8-bit value right by any even number of bits within a 32-bit word. The range is relative to the PC.

32-bit T32 encoding

±4095 bytes to a byte, halfword, or word-aligned address.

16-bit T32 encoding

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

7.10.1 Example of a jump table implementation with ADR

Example 7-3 shows A32 code that implements a jump table. Here, the ADR instruction loads the address of the jump table.

Example 7-3 Implementing a jump table (A32)

```
        AREA    Jump, CODE, READONLY     ; Name this block of code
        ARM                              ; Following code is A32 code
num     EQU     2                        ; Number of entries in jump table
        ENTRY                            ; Mark first instruction to execute
start                                    ; First instruction to call
        MOV     r0, #0                   ; Set up the three parameters
        MOV     r1, #3
        MOV     r2, #2
        BL      arithfunc                ; Call the function
stop
        MOV     r0, #0x18                ; angel_SWIreason_ReportException
        LDR     r1, =0x20026             ; ADP_Stopped_ApplicationExit
        SVC     #0x123456                ; ARM semihosting
arithfunc                                ; Label the function
        CMP     r0, #num                 ; Treat function code as unsigned integer
        BXHS    lr                       ; If code is >= num then return
        ADR     r3, JumpTable            ; Load address of jump table
        LDR     pc, [r3,r0,LSL#2]        ; Jump to the appropriate routine
JumpTable
        DCD     DoAdd
        DCD     DoSub
DoAdd
        ADD     r0, r1, r2               ; Operation 0
        BX      lr                       ; Return
DoSub
```

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

In Example 7-3 on page 7-15, the function `arithfunc` takes three arguments and returns a result in `R0`. The first argument determines the operation to be carried out on the second and third arguments:

argument1=0	Result = argument2 + argument3.
argument1=1	Result = argument2 – argument3.

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

`EQU` Is an assembler directive. You use it to give a value to a symbol. In Example 7-3 on page 7-15 it assigns the value 2 to *`num`*. When *`num`* is used elsewhere in the code, the value 2 is substituted. Using `EQU` in this way is similar to using `#define` to define a constant in C.

`DCD` Declares one or more words of store. In Example 7-3 on page 7-15 each `DCD` stores the address of a routine that handles a particular clause of the jump table.

`LDR` The `LDR PC,[R3,R0,LSL#2]` instruction loads the address of the required clause of the jump table into the PC. It:

- Multiplies the clause number in `R0` by 4 to give a word offset.
- Adds the result to the address of the jump table.
- Loads the contents of the combined address into the PC.

7.10.2 See also

Concepts

- *Load addresses to a register using LDR Rd, =label* on page 7-18
- *Load addresses to a register using ADR* on page 7-15.

Reference

armasm Reference Guide:

- *ADR (PC-relative)* on page 3-32.

7.11 Load addresses to a register using ADRL

The `ADRL` pseudo-instruction enables you to generate an address, within a certain range, without performing a data load. `ADRL` accepts a PC-relative expression, that is, a label with an optional offset where the address of the label is relative to the current PC.

Note

The label used with `ADRL` must be within the same code section. `armasm` faults references to labels that are out of range in the same section.

`armasm` converts an `ADRL` *`rn`*`,`*`label`* pseudo-instruction by generating:

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

The available range depends on the instruction set and encoding:

A32 Any value that can be generated by two `ADD` or two `SUB` instructions. That is, any value that can be produced by the addition of two values, each of which is 8 bits rotated right by any even number of bits within a 32-bit word. The range is relative to the PC.

32-bit T32 encoding

±1MB to a byte, halfword, or word-aligned address.

16-bit T32 encoding

`ADRL` is not available.

7.11.1 See also

Concepts

- *Load addresses to a register using LDR Rd, =label* on page 7-18
- *Load addresses to a register using ADR* on page 7-15.

7.12 Load addresses to a register using LDR Rd, =label

The `LDR Rd,=` pseudo-instruction can load any 32-bit numeric value into a register. It also accepts PC-relative expressions such as labels, and labels with offsets.

`armasm` 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 PC-relative `LDR` instruction that reads the address from the literal pool, for example:

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

  You must ensure that there is a literal pool within range (see *Literal pools* on page 7-12 for more information).

Unlike the `ADR` and `ADRL` pseudo-instructions, you can use `LDR` with labels that are outside the current section. `armasm` 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 7-4 shows how this works.

The instructions listed in the comments are the A32 instructions generated by `armasm`.

Example 7-4 Loading using LDR Rd, =label

```
        AREA    LDRlabel, CODE,READONLY
        ENTRY                              ; Mark first instruction to execute
start
        BL      func1                      ; Branch to first subroutine
        BL      func2                      ; Branch to second subroutine
stop
        MOV     r0, #0x18                  ; angel_SWIreason_ReportException
        LDR     r1, =0x20026               ; ADP_Stopped_ApplicationExit
        SVC     #0x123456                  ; ARM semihosting
func1
        LDR     r0, =start                 ; => LDR R0,[PC, #offset into
                                           ; Literal Pool 1]
        LDR     r1, =Darea + 12            ; => LDR R1,[PC, #offset into
                                           ; Literal Pool 1]
        LDR     r2, =Darea + 6000          ; => LDR R2, [PC, #offset into
                                           ; Literal Pool 1]
        BX      lr                         ; Return
        LTORG                              ; Literal Pool 1
func2
        LDR     r3, =Darea + 6000          ; => LDR r3, [PC, #offset into
                                           ; Literal Pool 1]
                                           ; (sharing with previous literal)
        ; LDR   r4, =Darea + 6004          ; If uncommented, produces an error
                                           ; because Literal Pool 2 is out of range
        BX      lr                         ; Return
Darea   SPACE   8000                       ; Starting at the current location,
                                           ; clears a 8000 byte area of memory
                                           ; to zero
        END                                ; Literal Pool 2 is automatically inserted
```

```
                                         ; after the END directive.
                                         ; It is out of range of all the LDR
                                         ; instructions in this example.
```

7.12.1 An LDR Rd, =label example: string copying

Example 7-5 shows an A32 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.
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 7-5 String copy

```
        AREA    StrCopy, CODE, READONLY
        ENTRY                           ; Mark first instruction to execute
start
        LDR     r1, =srcstr             ; Pointer to first string
        LDR     r0, =dststr             ; Pointer to second string
        BL      strcopy                 ; Call subroutine to do copy
stop
        MOV     r0, #0x18               ; angel_SWIreason_ReportException
        LDR     r1, =0x20026            ; ADP_Stopped_ApplicationExit
        SVC     #0x123456               ; ARM semihosting
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
```

7.12.2 See also

Concepts

- *Load immediate 32-bit values to a register using LDR Rd, =const* on page 7-11.

Reference

armasm Reference Guide:

- *LDR pseudo-instruction* on page 3-83
- *DCB* on page 10-24.

7.13 Other ways to load and store registers

You can load any 32-bit value from memory into a register with an `LDR` data load instruction. To store registers into memory you can use the `STR` data store instruction.

You can use the `MOV` instruction to move any 32-bit data from one register to another.

7.13.1 See also

Concepts

- *Load and store multiple register instructions* on page 7-21
- *A32 and T32 load and store multiple instructions* on page 7-22.

Reference

armasm Reference Guide:

- *Memory access instructions* on page 3-9
- *MOV and MVN* on page 3-93.

7.14 Load and store multiple register instructions

The A32 and T32 instruction sets include instructions that load and store multiple registers to and from memory.

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

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

Note

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

You can use the `--diag_warning 1206` `armasm` command-line option to check that registers in register lists are specified in increasing order.

7.14.1 See also

Concepts

- *A32 and T32 load and store multiple instructions* on page 7-22
- *Stack implementation using LDM and STM* on page 7-23
- *Stack operations for nested subroutines* on page 7-25
- *Block copy with LDM and STM* on page 7-26.

7.15 A32 and T32 load and store multiple instructions

The following instructions are available in both A32 and T32 instruction sets:

`LDM`	Load Multiple registers.
`STM`	Store Multiple registers.
`PUSH`	Store multiple registers onto the stack and update the stack pointer.
`POP`	Load multiple registers off the stack, and update the stack pointer.

In `LDM` and `STM` instructions:

- The list of registers loaded or stored can include:
  - In A32 instructions, any or all of R0-R12, SP, LR, and PC.
  - In 32-bit T32 instructions, any or all of R0-R12, and optionally LR or PC (`LDM` only) with some restrictions.
  - In 16-bit T32 instructions, any or all of R0-R7.
- The address can be:
  - Incremented after each transfer.
  - Incremented before each transfer (A32 instructions only).
  - Decremented after each transfer (A32 instructions only).
  - Decremented before each transfer (not in 16-bit encoded T32 instructions).
- The base register can be either:
  - Updated to point to the next block of data in memory.
  - Left as it was before the instruction.

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

In `PUSH` and `POP` instructions:

- The stack pointer (SP) is the base register, and is always updated.
- The address is incremented after each transfer in `POP` instructions, and decremented before each transfer in `PUSH` instructions.
- The list of registers loaded or stored can include:
  - In A32 instructions, any or all of R0-R12, SP, LR, and PC.
  - In 32-bit T32 instructions, any or all of R0-R12, and optionally LR or PC (`POP` only) with some restrictions.
  - In 16-bit T32 instructions, any or all of R0-R7, and optionally LR (`PUSH` only) or PC (`POP` only).

Note

Use of SP in the list of registers in any of these A32 instructions is deprecated.

A32 `STM` and `PUSH` instructions that use PC in the list of registers, and A32 `LDM` and `POP` instructions that use both PC and LR in the list of registers are deprecated.

7.15.1 See also

Concepts

- *Load and store multiple register instructions* on page 7-21.

7.16 Stack implementation using LDM and STM

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

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

Descending or ascending

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

Full or empty

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

To make it easier for the programmer, stack-oriented suffixes can be used instead of the increment or decrement, and before or after suffixes. Table 7-6 shows the stack-oriented suffixes and their equivalent addressing mode suffixes for load and store instructions.

Table 7-6 Stack-oriented suffixes and equivalent addressing mode suffixes

Stack-oriented suffix	For store or push instructions	For load or pop instructions
FD (Full Descending stack)	DB (Decrement Before)	IA (Increment After)
FA (Full Ascending stack)	IB (Increment Before)	DA (Decrement After)
ED (Empty Descending stack)	DA (Decrement After)	IB (Increment Before)
EA (Empty Ascending stack)	IA (Increment After)	DB (Decrement Before)

Table 7-7 shows the load and store multiple instructions with the stack-oriented suffixes for the various stack types.

Table 7-7 Suffixes for load and store multiple instructions

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

For example:

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

——— **Note** ———

The *Procedure Call Standard for the ARM Architecture* (AAPCS), and `armclang` always use a full descending stack.

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

7.16.1 See also

Concepts

- *Load and store multiple register instructions* on page 7-21.

Other information

- *Procedure Call Standard for the ARM Architecture*
  `http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042-/index.html`.

7.17 Stack operations for nested subroutines

Stack operations are very useful at subroutine entry and exit. At the start of a subroutine, any working registers required can be stored on the stack, and at exit they can be popped off again.

In addition, if the link register is pushed onto the stack at entry, additional subroutine calls can be made safely without causing the return address to be lost. If you do this, you can also return from a subroutine by popping PC off the stack at exit, instead of popping LR and then moving that value into PC. For example:

```
subroutine  PUSH    {r5-r7,lr} ; Push work registers and lr
            ; code
            BL      somewhere_else
            ; code
            POP     {r5-r7,pc} ; Pop work registers and pc
```

7.17.1 See also

Concepts

- *Subroutine calls* on page 7-5.

7.18 Block copy with LDM and STM

Example 7-6 is an A32 code routine that copies a set of words from a source location to a destination by copying a single word at a time.

Example 7-6 Block copy without LDM and STM

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

You can make this module 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 available registers. You can find the number of eight-word multiples in the block to be copied (if `R2` = number of words to be copied) using:

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

You can use this value to control the number of iterations through a loop that copies eight words per iteration. When there are fewer than eight words left, you can find the number of words left (assuming that `R2` has not been corrupted) using:

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

Example 7-7 lists the block copy module rewritten to use `LDM` and `STM` for copying.

Example 7-7 Block copy using LDM and STM

```
            AREA    Block, CODE, READONLY    ; name this block of code
num         EQU     20                       ; set number of words to be copied
            ENTRY                            ; mark the first instruction called
start
            LDR     r0, =src                 ; r0 = pointer to source block
            LDR     r1, =dst                 ; r1 = pointer to destination block
            MOV     r2, #num                 ; r2 = number of words to copy
            MOV     sp, #0x400               ; Set up stack pointer (sp)
blockcopy
            MOVS    r3,r2, LSR #3            ; Number of eight word multiples
            BEQ     copywords                ; Fewer than eight words to move?
            PUSH    {r4-r11}                 ; Save some working registers
octcopy
            LDM     r0!, {r4-r11}            ; Load 8 words from the source
            STM     r1!, {r4-r11}            ; and put them at the destination
            SUBS    r3, r3, #1               ; Decrement the counter
            BNE     octcopy                  ; ... copy more
            POP     {r4-r11}                 ; Don't need these now - restore
                                             ; originals
copywords
            ANDS    r2, r2, #7               ; Number of odd words to copy
            BEQ     stop                     ; No words left to copy?
wordcopy
            LDR     r3, [r0], #4             ; Load a word from the source and
            STR     r3, [r1], #4             ; store it to the destination
            SUBS    r2, r2, #1               ; Decrement the counter
            BNE     wordcopy                 ; ... copy more
stop
            MOV     r0, #0x18                ; angel_SWIreason_ReportException
            LDR     r1, =0x20026             ; ADP_Stopped_ApplicationExit
            SVC     #0x123456                ; ARM semihosting
            AREA    BlockData, DATA, READWRITE
src         DCD     1,2,3,4,5,6,7,8,1,2,3,4,5,6,7,8,1,2,3,4
dst         DCD     0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
            END
```

7.19 Memory accesses

The following addressing modes are commonly permitted for memory access instructions:

Offset addressing

The offset value is applied to an address obtained from the base register. The result is used as the address for the memory access. The base register is unchanged. The assembly language syntax for this mode is:

```
[Rn, offset]
```

Pre-indexed addressing

The offset value is applied to an address obtained from the base register. The result is used as the address for the memory access, and written back into the base register. The assembly language syntax for this mode is:

```
[Rn, offset]!
```

Post-indexed addressing

The address obtained from the base register is used, unchanged, as the address for the memory access. The offset value is applied to the address, and written back into the base register. The assembly language syntax for this mode is:

```
[Rn], offset
```

In each case, `Rn` is the base register and `offset` can be:

- An immediate constant.
- An index register, `Rm`.
- A shifted index register, such as `Rm`, `LSL #shift`.

7.19.1 See also

Concepts

- *Registers in AArch32 state* on page 4-4
- *Address alignment in A32/T32 code* on page 9-19.

Reference

armasm Reference Guide:

- *Memory access instructions* on page 3-9.

7.20 Read-Modify-Write procedure

When you want to modify specific bits in a system register, you must ensure that you do not modify the other bits in the same register. This is because individual bits in a system register control different system functionality, and modifying them might cause your program to behave incorrectly. You must use a read-modify-write procedure to ensure that you modify only the bits you want to change.

To read-modify-write a system register, the instruction sequence must be:

1. The first instruction copies the value from the target system register to a temporary general-purpose register.
2. The next one or more instructions modify the required bits in the general-purpose register. This can be one or both of:
   - `BIC` to clear the bits that must be cleared to 0.
   - `ORR` to set the bits that must be set to 1.
3. The final instruction writes the value from the general-purpose register to the target system register.

7.20.1 Example

This example shows the read-modify-write procedure to change some bits of a SIMD and floating-point system register, FPSCR without affecting the other bits.

```
VMRS    r10,FPSCR           ; copy FPSCR into the general-purpose r10
BIC     r10,r10,#0x00370000 ; clears STRIDE bits[21:20] and LEN bits[18:16]
ORR     r10,r10,#0x00030000 ; sets bits[17:16] (STRIDE =1 and LEN = 4)
VMSR    FPSCR,r10           ; copy r10 back into FPSCR
```

7.20.2 See also

Concepts

- *Register accesses in AArch32 state* on page 4-7
- *The Q flag in AArch32 state* on page 4-12.

Reference

armasm Reference Guide:

- *VMRS and VMSR* on page 4-74
- *MRS (PSR to general-purpose register)* on page 3-100
- *MSR (general-purpose register to PSR)* on page 3-103.

7.21 Optional hash

You do not need to specify a hash before immediate constants in any instruction syntax (including A32, T32, Advanced SIMD, and floating-point instructions). For example, the following are valid instructions:

```
BKPT 100
MOVT R1, 256
VCEQ.I8 Q1, Q2, 0
```

By default, `armasm` warns if you do not specify a hash:

```
WARNING: A1865W: '#' not seen before constant expression.
```

This can be suppressed with `--diag_suppress=1865`.

If you use the assembly code with another assembler, you are advised to use the # before all immediates. The disassembler always shows the # for clarity.

7.21.1 See also

Reference

armasm Reference Guide:

- *A32 and T32 instruction summary* on page 3-2
- *Advanced SIMD and floating-point instruction summary* on page 4-2.

7.22 Use of macros

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

7.22.1 See also

Concepts

- *Test-and-branch macro example* on page 7-32
- *Unsigned integer division macro example* on page 7-33.

Reference

armasm Reference Guide:

- *MACRO and MEND* on page 10-67.

7.23 Test-and-branch macro example

In A32 code, a test-and-branch operation requires two instructions to implement.

You can define a macro definition such as this:

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

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

7.23.1 See also

Concepts

- *Use of macros* on page 7-31
- *Unsigned integer division macro example* on page 7-33
- *Numeric local labels* on page 10-12.

7.24 Unsigned integer division macro example

Example 7-8 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 7-8 Unsigned integer division with a macro

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

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

To avoid multiple definitions of labels if `DivMod` is used more than once in the assembly source, the macro uses numeric local labels (90, 91).

Example 7-9 on page 7-34 shows the code that this macro produces if it is invoked as follows:

```
ratio  DivMod  R0,R5,R4,R2
```

Example 7-9 Output from division macro

```
        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
```

7.24.1 See also

Concepts

- *Use of macros* on page 7-31
- *Test-and-branch macro example* on page 7-32
- *Numeric local labels* on page 10-12.

7.25 Instruction and directive relocations

A relocation is a directive embedded in the object file that enables source code to refer to a label whose target address is unknown or cannot be calculated at assembly time. `armasm` emits a relocation in the object file, and the linker resolves this to the address where the target is placed.

`armasm` relocates the data directives `DCB`, `DCW`, `DCWU`, `DCD`, and `DCDU` if their syntax contains an external symbol, that is a symbol declared using `IMPORT` or `EXTERN`. This causes the bottom 8, 16, or 32 bits of the address to be used at link-time.

The `REQUIRE` directive emits a relocation to signal to the linker that the target label must be present if the current section is present.

`armasm` is permitted to emit a relocation for these instructions:

`LDR` **(PC-relative)**

All A32 and T32 instructions, except the T32 doubleword instruction, can be relocated.

`PLD`**,** `PLDW`**, and** `PLI`

All A32 and T32 instructions can be relocated.

`B`**,** `BL`**, and** `BLX`

All A32 and T32 instructions can be relocated.

`CBZ` **and** `CBNZ`

All T32 instructions can be relocated but this is discouraged because of the limited branch range of these instructions.

`LDC` **and** `LDC2`

Only A32 instructions can be relocated.

`VLDR`

Only A32 instructions can be relocated.

`armasm` emits a relocation for these instructions if the label used meets any of the following requirements, as appropriate for the instruction type:

- The label is `WEAK`.
- The label is not in the same `AREA`.
- The label is external to the object (`IMPORT` or `EXTERN`).

For `B`, `BL`, and `BX` instructions, `armasm` emits a relocation also if:

- The label is a function.
- The label is exported using `EXPORT` or `GLOBAL`.

——— **Note** ———

You can use the `RELOC` directive to control the relocation at a finer level, but this requires knowledge of the ABI.

7.25.1 Example

```
    IMPORT sym     ; sym is an external symbol
    DCW sym        ; Because DCW only outputs 16 bits, only the lower 16 bits
                   ; of the address of sym are inserted at link-time.
```

7.25.2 See also

Reference

armasm Reference Guide:

- *AREA* on page 10-14
- *EXPORT or GLOBAL* on page 10-37
- *IMPORT and EXTERN* on page 10-58
- *REQUIRE* on page 10-78
- *RELOC* on page 10-77
- *DCB* on page 10-24
- *DCD and DCDU* on page 10-25
- *DCW and DCWU* on page 10-32
- *LDR (PC-relative)* on page 3-79
- *ADR (PC-relative)* on page 3-32
- *PLD, PLDW, and PLI* on page 3-114
- *B, BL, BX, BLX, and BXJ* on page 3-44
- *CBZ and CBNZ* on page 3-49
- *LDC and STC* on page 3-66
- *VLDR and VSTR* on page 4-54.

Other information

- *ELF for the ARM Architecture*
  `http://infocenter.arm.com/help/topic/com.arm.doc.ihi0044-/index.html`

7.26 Symbol versions

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

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

- `name@ver` if *ver* is a non default version of *name*
- `name@@ver` if *ver* is the default version of *name*.

The version symbols must be enclosed in vertical bars.

For example, to define a default version:

```
|my_versioned_symbol@@ver2|   ; Default version
my_asm_function PROC
                ...
                BX lr
                ENDP
```

To define a non default version:

```
|my_versioned_symbol@ver1|    ; Non default version
my_old_asm_function     PROC
                        ...
                        BX lr
                        ENDP
```

7.26.1 See also

Concepts

armlink User Guide:

- Chapter 7 *Accessing and managing symbols with armlink*.

Other information

- *Base Platform ABI for the ARM Architecture*
  `http://infocenter.arm.com/help/topic/com.arm.doc.ihi0037-/index.html`.

7.27 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.

`armasm` uses frame directives to insert DWARF debug frame information into the object file in ELF format that it produces. This information is required by a debugger for stack unwinding and for profiling.

Be aware of the following:

- Frame directives do not affect the code that `armasm` produces.
- `armasm` does not validate the information in frame directives against the instructions emitted.

7.27.1 See also

Concepts

- *Exception tables and Unwind tables* on page 7-39.

Reference

armasm Reference Guide:

- *About frame directives* on page 10-7.

Other information

- *Procedure Call Standard for the ARM Architecture* `http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042-/index.html`.

7.28 Exception tables and Unwind tables

Exception tables are necessary to handle exceptions thrown by functions in high-level languages such as C++. *Unwind* tables contain debug frame information which are also necessary for the handling of such exceptions. An exception can only propagate through a function with an *unwind* table.

Functions written in C++ have unwind information by default. However, for assembly language functions (code encased by `PROC`/`ENDP` or `FUNC`/`ENDFUNC`) that are called from C++ code, you must ensure that there are exception tables and *unwind* tables to enable the exceptions to propagate through them.

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.

`armasm` can generate *nounwind* table entries for all functions and non-functions. It 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. To be able to create an *unwind* table for a function, each `POP` or `PUSH` instruction must be followed by a `FRAME POP` or `FRAME PUSH` directive respectively. Functions must conform to the conditions set out in the *Exception Handling ABI for the ARM Architecture* (EHABI), section 9.1 *Constraints on Use*. If `armasm` cannot generate an *unwind* table it generates a *nounwind* table.

7.28.1 See also

Concepts

- *Frame directives* on page 7-38.

Reference

armasm Reference Guide:

- *About frame directives* on page 10-7
- *--no_exceptions_unwind* on page 2-51
- *--exceptions* on page 2-30
- *--no_exceptions* on page 2-50
- *FRAME UNWIND ON* on page 10-51
- *FRAME UNWIND OFF* on page 10-52
- *FUNCTION or PROC* on page 10-53
- *ENDFUNC or ENDP* on page 10-34.

Chapter 8
Condition Codes

The following topics describe condition codes and conditional execution of A64, A32 and T32 code:

8.1 Conditional instructions

You can execute an instruction conditionally based on the condition flags set by another instruction, either:

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

In AArch32 state, whether an instruction can be conditional or not depends on the instruction set state that the processor is in. Very few A64 instructions can be conditionally executed.

To make an instruction conditional, you add a condition code suffix to the instruction mnemonic. The condition code suffix enables the processor to test a condition based on the flags. If the test fails, the instruction:

- Does not execute.
- Does not write any value to its destination register.
- Does not affect any of the flags.
- Does not generate any exception.

8.1.1 See also

Concepts

- *Condition code suffixes* on page 8-12
- *Conditional execution in A32 code* on page 8-3
- *Conditional execution in T32 code* on page 8-4
- *Conditional execution in A64 code* on page 8-5.

8.2 Conditional execution in A32 code

Almost all A32 instructions can be executed conditionally on the value of the condition flags in the APSR. You can either add a condition code suffix to the instruction or you can conditionally skip over the instruction using a conditional branch instruction.

Using conditional branch instructions to control the flow of execution can be more efficient when a series of instructions depend on the same condition.

Example 8-1 Conditional instructions to control execution

```
    ; flags set by a previous instruction
    LSLEQ r0, r0, #24
    ADDEQ r0, r0, #2
    ;…
```

Example 8-2 Conditional branch to control execution

```
    ; flags set by a previous instruction
    BNE over
    LSL r0, r0, #24
    ADD r0, r0, #2
over
    ;…
```

8.2.1 See also

Concepts

- *Conditional execution in T32 code* on page 8-4.

8.3 Conditional execution in T32 code

In T32 code, there are several ways to achieve conditional execution. You can conditionally skip over the instruction using a conditional branch instruction. Instructions can also be conditionally executed by using either of the following:

- `CBZ` and `CBNZ`.
- The `IT` (If-Then) instruction.

The T32 `CBZ` (Conditional Branch on Zero) and `CBNZ` (Conditional Branch on Non-Zero) instructions compare the value of a register against zero and branch on the result.

`IT` is a 16-bit instruction that enables a single subsequent 16-bit T32 instruction from a restricted set to be conditionally executed, based on the value of the condition flags, and the condition code suffix specified.

Example 8-3 Conditional instructions using IT block

```
; flags set by a previous instruction
IT   EQ
LSLEQ r0, r0, #24
;…
```

The use of the IT instruction is deprecated when any of the following are true:

- There is more than one instruction in the IT block.
- There is a 32-bit instruction in the IT block.
- The instruction in the IT block references the PC.

8.3.1 See also

Concepts

- *Conditional execution in A32 code* on page 8-3.

Reference

- *IT* on page 3-63
- *CBZ and CBNZ* on page 3-49.

8.4 Conditional execution in A64 code

In the A64 instruction set, there are a small number of instructions that are truly conditional. Truly conditional means that when the condition is false, the instruction advances the program counter but has no other effect. The conditional branch, `B.`*`cond`* is a truly conditional instruction. The condition code is appended to the instruction with a '.' delimiter, for example `B.EQ`.

There are other truly conditional branch instructions that execute depending on the value of the Zero condition flag. You cannot append any condition code suffix to them. These instructions are:

- `CBNZ`.
- `CBZ`.
- `TBNZ`.
- `TBZ`.

There is a small number of A64 instructions that are unconditionally executed but use the condition code as a source operand. These instructions always execute but the operation depends on the value of the condition code. These instructions can be categorized as:

- Conditional data processing instructions, for example `CSEL`.
- Conditional comparison instructions, `CCMN` and `CCMP`.

In these instructions, you specify the condition code in the final operand position, for example `CSEL Wd,Wm,Wn,NE`.

8.4.1 See also

Concepts

- *Condition flags* on page 8-6.

Other information

- *ARMv8-A Architecture Reference Manual* `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0487a.b/index.html`.

8.5 Condition flags

The condition flags are:

N	Set to 1 when the result of the operation is negative, cleared to 0 otherwise.
Z	Set to 1 when the result of the operation is zero, cleared to 0 otherwise.
C	Set to 1 when the operation results in a carry, cleared to 0 otherwise.
V	Set to 1 when the operation causes overflow, cleared to 0 otherwise.

8.5.1 See also

Concepts

- *Updates to the condition flags in A32/T32 code* on page 8-7
- *Condition code suffixes* on page 8-12
- *Read-Modify-Write procedure* on page 7-29
- *Floating-point instructions that update the condition flags* on page 8-9
- *Updates to the condition flags in A64 code* on page 8-8
- *Carry flag* on page 8-10
- *Overflow flag* on page 8-11.

8.6 Updates to the condition flags in A32/T32 code

In AArch32 state, the condition flags are held in the *Application Program Status Register* (APSR). You can read and modify the flags using the read-modify-write procedure.

Most A32 and T32 data processing instructions have an option to update the condition flags according to the result of the operation. Instructions with the optional S suffix update the flags. Conditional instructions that are not executed have no effect on the flags.

Which flags are updated depends on the instruction. Some instructions update all flags, and some update a subset of the flags. If a flag is not updated, the original value is preserved. In the *armasm Reference Guide*, the description of each instruction mentions the effect it has on the flags.

——— **Note** ———

Most instructions update the condition flags only if the S suffix is specified. The instructions `CMP`, `CMN`, `TEQ`, and `TST` always update the flags.

———————————

8.6.1 See also

Concepts

- *Condition code suffixes* on page 8-12
- *Read-Modify-Write procedure* on page 7-29
- *Floating-point instructions that update the condition flags* on page 8-9
- *Updates to the condition flags in A64 code* on page 8-8
- *Carry flag* on page 8-10
- *Overflow flag* on page 8-11.

8.7 Updates to the condition flags in A64 code

In AArch64 state, the N, Z, C, and V condition flags are held in the NZCV system register, which is part of the process state. You can access the flags using the `MSR` and `MRS` instructions.

Note

An instruction updates the condition flags only if the S suffix is specified, except the instructions `CMP`, `CMN`, `CCMP`, `CCMN`, and `TST`, which always update the condition flags. The instruction also determines which flags get updated. Conditional instructions do not affect the flags if the instruction does not execute.

8.7.1 Example

This example shows the read-modify-write procedure to change some of the condition flags in A64 code.

```
MRS  x1, NZCV          ; copy N, Z, C, and V flags into general-purpose x1
MOV  x2, #0x30000000
BIC  x1,x1,x2          ; clears the C and V flags (bits 29,28)
ORR  x1,x1,#0xC0000000 ; sets the N and Z flags (bits 31,30)
MSR  NZCV, x1          ; copy x1 back into NZCV register to update the condition flags
```

8.7.2 See also

Concepts

- *Condition code suffixes* on page 8-12
- *Read-Modify-Write procedure* on page 7-29
- *Floating-point instructions that update the condition flags* on page 8-9
- *Updates to the condition flags in A32/T32 code* on page 8-7.

8.8 Floating-point instructions that update the condition flags

The only A32/T32 floating-point instructions that can update the condition flags are `VCMP` and `VCMPE`. Other floating-point or Advanced SIMD instructions cannot modify the flags.

`VCMP` and `VCMPE` do not update the flags directly, but update a separate set of flags in the *Floating-Point Status and Control Register* (FPSCR). To use these flags to control conditional instructions, including conditional floating-point instructions, you must first update the condition flags yourself. To do this, you must copy the flags from the FPSCR into the APSR using a `VMRS` instruction:

```
    VMRS APSR_nzcv, FPSCR
```

All A64 floating-point comparison instructions can update the condition flags. These instructions update the flags directly in the NZCV register.

8.8.1 See also

Concepts

- *Read-Modify-Write procedure* on page 7-29
- *Floating-point instructions that update the condition flags*
- *Updates to the condition flags in A64 code* on page 8-8
- *Carry flag* on page 8-10
- *Overflow flag* on page 8-11.

Reference

- *VCMP, VCMPE* on page 4-39
- *VMRS and VMSR* on page 4-74.

Other information

- *ARMv8-A Architecture Reference Manual*
  `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0487a.b/index.html`.

8.9 Carry flag

The carry (C) flag is set when an operation results in a carry, or when a subtraction results in no borrow.

In A32/T32 code, C is set in one of the following ways:

- For an addition, including the comparison instruction `CMN`, C is set to 1 if the addition produced a carry (that is, an unsigned overflow), and to 0 otherwise.
- For a subtraction, including the comparison instruction `CMP`, C is set to 0 if the subtraction produced a borrow (that is, an unsigned underflow), and to 1 otherwise.
- For non-additions/subtractions that incorporate a shift operation, C is set to the last bit shifted out of the value by the shifter.
- For other non-additions/subtractions, C is normally left unchanged, but see the individual instruction descriptions for any special cases.
- The floating-point compare instructions, `VCMP` and `VCMPE` set the C flag and the other condition flags in the FPSCR to the result of the comparison.

In A64 code, C is set in one of the following ways:

- For an addition, including the comparison instruction `CMN`, C is set to 1 if the addition produced a carry (that is, an unsigned overflow), and to 0 otherwise.
- For a subtraction, including the comparison instruction `CMP` and the negate instructions `NEGS` and `NGCS`, C is set to 0 if the subtraction produced a borrow (that is, an unsigned underflow), and to 1 otherwise.
- For the integer and floating-point conditional compare instructions `CCMP`, `CCMN`, `FCCMP`, and `FCCMPE`, C and the other condition flags are set either to the result of the comparison, or directly from an immediate value.
- For the floating-point compare instructions, `FCMP` and `FCMPE`, C and the other condition flags are set to the result of the comparison.
- For other instructions, C is normally left unchanged, but see the individual instruction descriptions for any special cases.

8.9.1 See also

Concepts

- *Condition flags* on page 8-6
- *Predeclared core register names in AArch32 state* on page 4-8
- *Predeclared core register names in AArch64 state* on page 5-6
- *Condition code suffixes* on page 8-12
- *Updates to the condition flags in A32/T32 code* on page 8-7
- *Updates to the condition flags in A64 code* on page 8-8
- *Overflow flag* on page 8-11.

8.10 Overflow flag

In A32/T32 code, overflow occurs if the result of an add, subtract, or compare is greater than or equal to 2^{31}, or less than -2^{31}.

In A64 instructions that use the 64-bit X registers, overflow occurs if the result of an add, subtract, or compare is greater than or equal to 2^{63}, or less than -2^{63}.

In A64 instructions that use the 32-bit W registers, overflow occurs if the result of an add, subtract, or compare is greater than or equal to 2^{31}, or less than -2^{31}.

8.10.1 See also

Concepts

- *Condition flags* on page 8-6
- *Condition code suffixes* on page 8-12
- *Updates to the condition flags in A32/T32 code* on page 8-7
- *Updates to the condition flags in A64 code* on page 8-8
- *Carry flag* on page 8-10.

8.11 Condition code suffixes

The instructions that can be conditional have an optional condition code, shown in syntax descriptions as `{cond}`. This condition is encoded in A32 instructions and in A64 instructions. The condition is encoded in a preceding `IT` instruction for T32 instructions. An instruction with a condition code is only executed if the condition flags meet the specified condition.

The following table shows the condition codes that you can use and the flags they depend on.

Table 8-1 Condition code suffixes

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

The following example shows conditional execution in A32 code.

Example 8-4

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

8.11.1 See also

Concepts

- *Updates to the condition flags in A32/T32 code* on page 8-7
- *Updates to the condition flags in A64 code* on page 8-8
- *Comparison of condition code meanings in integer and floating-point code* on page 8-14

- *Conditional instructions* on page 8-2.

8.12 Comparison of condition code meanings in integer and floating-point code

The meaning of the condition code mnemonic suffixes depends on whether the condition flags were set by a floating-point instruction or by an integer data processing instruction. This is because:

- Floating-point values are never unsigned, so the unsigned conditions are not required.
- *Not-a-Number* (NaN) values have no ordering relationship with numbers or with each other, so additional conditions are required to account for *unordered* results.

The meanings of the condition code mnemonics are shown in Table 8-2.

Table 8-2 Condition codes

Mnemonic	Meaning after integer data processing instruction	Meaning after floating-point instruction
`EQ`	Equal	Equal
`NE`	Not equal	Not equal, or unordered
`CS`	Carry set	Greater than or equal, or unordered
`HS`	Unsigned higher or same	Greater than or equal, or unordered
`CC`	Carry clear	Less than
`LO`	Unsigned lower	Less than
`MI`	Negative	Less than
`PL`	Positive or zero	Greater than or equal, or unordered
`VS`	Overflow	Unordered (at least one NaN operand)
`VC`	No overflow	Not unordered
`HI`	Unsigned higher	Greater than, or unordered
`LS`	Unsigned lower or same	Less than or equal
`GE`	Signed greater than or equal	Greater than or equal
`LT`	Signed less than	Less than, or unordered
`GT`	Signed greater than	Greater than
`LE`	Signed less than or equal	Less than or equal, or unordered
`AL`	Always (normally omitted)	Always (normally omitted)

Note

The type of the instruction that last updated the condition flags determines the meaning of the condition codes.

8.12.1 See also

Concepts

- *Floating-point instructions that update the condition flags* on page 8-9
- *Condition code suffixes* on page 8-12.

Reference

armasm Reference Guide:

- *IT* on page 3-63
- *VMRS and VMSR* on page 4-74.

Other information

- *ARM Architecture Reference Manual* `http://infocenter.arm.com/help/topic/com.arm.doc.subset.architecture.reference/`.

8.13 Benefits of using conditional execution in A32 and T32 code

You can use conditional execution of A32 instructions to reduce the number of branch instructions in your code. This improves code density. The `IT` instruction in T32 achieves a similar improvement.

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

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

8.13.1 See also

Concepts

- *Illustration of the benefits of conditional instructions in A32 and T32 code* on page 8-17.

8.14 Illustration of the benefits of conditional instructions in A32 and T32 code

This illustrates the difference between using branches and using conditional instructions. It uses the Euclid algorithm for the *Greatest Common Divisor* (gcd) to demonstrate how conditional instructions improve code size and speed.

In C the gcd algorithm can be expressed as:

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

The following examples show implementations of the gcd algorithm with and without conditional instructions.

Note

The detailed analysis of execution speed only applies to an ARM7™ processor. The code density calculations apply to all ARM processors.

8.14.1 Example of conditional execution using branches in A32 code

This is an A32 code implementation of the gcd algorithm using branches, without using any other conditional instructions. Conditional execution is achieved by using conditional branches, rather than individual conditional instructions:

```
gcd     CMP      r0, r1
        BEQ      end
        BLT      less
        SUBS     r0, r0, r1  ; could be SUB r0, r0, r1 for A32
        B        gcd
less
        SUBS     r1, r1, r0  ; could be SUB r1, r1, r0 for A32
        B        gcd
end
```

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

The following table shows the number of cycles this implementation uses on an ARM7 processor when R0 equals 1 and R1 equals 2.

Table 8-3 Conditional branches only

R0: a	R1: b	Instruction	Cycles (ARM7)
1	2	`CMP r0, r1`	1
1	2	`BEQ end`	1 (not executed)
1	2	`BLT less`	3

Table 8-3 Conditional branches only (continued)

R0: a	R1: b	Instruction	Cycles (ARM7)
1	2	`SUB r1, r1, r0`	1
1	2	`B gcd`	3
1	1	`CMP r0, r1`	1
1	1	`BEQ end`	3
			Total = 13

8.14.2 Example of conditional execution using conditional instructions in A32 code

This is an A32 code implementation of the gcd algorithm using individual conditional instructions in A32 code. The gcd algorithm only takes four instructions:

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

In addition to improving code size, in most cases this code executes faster than the version that uses only branches.

The following table shows the number of cycles this implementation uses on an ARM7 processor when R0 equals 1 and R1 equals 2.

Table 8-4 All instructions conditional

R0: a	R1: b	Instruction	Cycles (ARM7)
1	2	`CMP r0, r1`	1
1	2	`SUBGT r0,r0,r1`	1 (not executed)
1	1	`SUBLT r1,r1,r0`	1
1	1	`BNE gcd`	3
1	1	`CMP r0,r1`	1
1	1	`SUBGT r0,r0,r1`	1 (not executed)
1	1	`SUBLT r1,r1,r0`	1 (not executed)
1	1	`BNE gcd`	1 (not executed)
			Total = 10

Comparing this with the example that uses only branches:

- Replacing branches with conditional execution of all instructions saves three cycles.
- Where R0 equals R1, both implementations execute in the same number of cycles. For all other cases, the implementation that uses conditional instructions executes in fewer cycles than the implementation that uses branches only.

8.14.3 Example of conditional execution using conditional instructions in T32 code

You can use the `IT` instruction to write conditional instructions in T32 code. The T32 code implementation of the gcd algorithm using conditional instructions is very similar to the implementation in A32 code. The implementation in T32 code is:

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

This assembles equally well to A32 or T32 code. The assembler checks the `IT` instructions, but omits them on assembly to A32 code.

It requires one more instruction in T32 code (the `IT` instruction) than in A32 code, but the overall code size is 10 bytes in T32 code compared with 16 bytes in A32 code.

8.14.4 Example of conditional execution code using branches in T32 code

In architectures before ARMv6T2, there is no `IT` instruction and therefore T32 instructions cannot be executed conditionally except for the `B` branch instruction. The gcd algorithm must be written with conditional branches and is very similar to the A32 code implementation using branches, without conditional instructions.

The T32 code implementation of the gcd algorithm without conditional instructions requires seven instructions. The overall code size is 14 bytes. This is even less than the A32 implementation that uses conditional instructions, which uses 16 bytes.

In addition, on a system using 16-bit memory this T32 implementation runs faster than both A32 implementations because only one memory access is required for each 16-bit T32 instruction, whereas each 32-bit A32 instruction requires two fetches.

8.14.5 See also

Concepts

- *Benefits of using conditional execution in A32 and T32 code* on page 8-16
- *Condition code suffixes* on page 8-12
- *Optimization for execution speed* on page 8-20.

Reference

armasm Reference Guide:

- *IT* on page 3-63.

Other information

- *ARM Architecture Reference Manual*
  `http://infocenter.arm.com/help/topic/com.arm.doc.subset.architecture.reference/`
- *Technical Reference Manual* for your processor.

8.15 Optimization for execution speed

To optimize code for execution speed you must have detailed knowledge of the instruction timings, branch prediction logic, and cache behavior of your target system.

8.15.1 See also

Other information

- *ARM Architecture Reference Manual*
  `http://infocenter.arm.com/help/topic/com.arm.doc.subset.architecture.reference/`
- *Technical Reference Manual* for your processor.

Chapter 9
Using armasm

The following topics describe how to use `armasm`:

9.1 armasm command-line syntax

The command for invoking `armasm` is:

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

where *inputfile* is an assembly source file and `options` instruct `armasm` how to assemble the *inputfile*. You can invoke `armasm` with any combination of options separated by spaces.

The `armasm` command line is case-insensitive, except in filenames and where specified. If the command line contains options that conflict with each other, then the last option found always takes precedence.

9.1.1 See also

Reference

- *armasm commands listed in groups* on page 9-3.

9.2 armasm commands listed in groups

See the following command-line options in the *armasm Reference Guide*.

Help

- *--help* on page 2-35
- *--version_number* on page 2-69
- *--vsn* on page 2-71.

Source

- *--16* on page 2-5
- *--32* on page 2-6
- *--arm* on page 2-9
- *--arm_only* on page 2-10
- *-i* on page 2-36
- *--maxcache* on page 2-45
- *--no_esc* on page 2-48
- *--no_regs* on page 2-54
- *--pd* on page 2-59
- *--predefine* on page 2-60
- *--reduce_paths* on page 2-61
- *--regnames* on page 2-62
- *--thumb* on page 2-65
- *--unsafe* on page 2-67.

Output

- *--debug* on page 2-17
- *--depend* on page 2-18
- *--depend_format* on page 2-19
- *--dllexport_all* on page 2-25
- *--dwarf2* on page 2-26
- *--dwarf3* on page 2-27
- *--execstack* on page 2-29
- *-g* on page 2-34
- *--keep* on page 2-37
- *--length* on page 2-38
- *--list* on page 2-42
- *-m* on page 2-44
- *--md* on page 2-46
- *--no_code_gen* on page 2-47
- *--no_execstack* on page 2-49
- *--no_hide_all* on page 2-52
- *--no_terse* on page 2-55
- *-o* on page 2-58
- *--width* on page 2-72

- *--xref* on page 2-73
- *--untyped_local_labels* on page 2-68.

Target

- *--apcs* on page 2-7
- *--bi* on page 2-11
- *--bigend* on page 2-12
- *--cpu* on page 2-15
- *--fpmode* on page 2-32
- *--fpu* on page 2-33
- *--li* on page 2-39
- *--littleend* on page 2-43
- *--no_unaligned_access* on page 2-56
- *--unaligned_access* on page 2-66.

Diagnostics

- *--brief_diagnostics* on page 2-13
- *--checkreglist* on page 2-14
- *--diag_error* on page 2-20
- *--diag_remark* on page 2-21
- *--diag_warning* on page 2-24
- *--diag_suppress* on page 2-23
- *--diag_style* on page 2-22
- *--errors* on page 2-28
- *--no_warn* on page 2-57
- *--report-if-not-wysiwyg* on page 2-63.

Exception table generation

- *--exceptions* on page 2-30
- *--exceptions_unwind* on page 2-31.

License

- *--licretry* on page 2-41.

Command-line options in a text file

- *--via* on page 2-70.

9.3 Specify command-line options with an environment variable

You can specify command-line options by setting the value of the `ARMCOMPILER6_ASMOPT` environment variable. The syntax is identical to the command-line syntax. The assembler reads the value of `ARMCOMPILER6_ASMOPT` and inserts it at the front of the command string. This means that options specified in `ARMCOMPILER6_ASMOPT` can be overridden by arguments on the command line.

9.3.1 See also

Concepts

- *armasm command-line syntax* on page 9-2.

9.4 Using `stdin` to input source code to armasm

Instead of creating a file for your source code, you can use `stdin` to pipe output from another program into `armasm` or to input source code directly on the command line. This is useful if you want to test a short piece of code without having to create a file for it.

To use `stdin` to pipe output from another program into `armasm`, invoke the program and the assembler using the pipe character (|). Use the minus character (`-`) as the source filename to instruct the assembler to take input from `stdin`. You must specify the output filename using the `-o` option. You can specify the command-line options you want to use. For example to pipe output from `fromelf`:

```
fromelf --disassemble input.o | armasm -o output.o -
```

To use `stdin` to input source code directly on the command line:

1. Invoke the assembler with the command-line options you want to use. Use the minus character (`-`) as the source filename to instruct the assembler to take input from `stdin`. You must specify the output filename using the `-o` option. For example:

   ```
   armasm --bigend -o output.o -
   ```

2. Enter your input. For example:

   ```
           AREA     A32ex, CODE, READONLY
                                   ; Name this block of code A32ex
           ENTRY                   ; Mark first instruction to execute
   start
           MOV      r0, #10        ; Set up parameters
           MOV      r1, #3
           ADD      r0, r0, r1     ; r0 = r0 + r1
   stop
           MOV      r0, #0x18      ; angel_SWIreason_ReportException
           LDR      r1, =0x20026   ; ADP_Stopped_ApplicationExit
           SVC      #0x123456      ; ARM semihosting
           END                     ; Mark end of file
   ```

3. Terminate your input by entering:
   - `Ctrl+Z` then `Return` on Microsoft Windows systems
   - `Ctrl+D` on Unix-based operating systems.

Note

The source code from `stdin` is stored in an internal cache that can hold up to 8 MB. You can increase this cache size using the `--maxcache` command-line option.

9.4.1 See also

Reference

- *armasm command-line syntax* on page 9-2
- *armasm commands listed in groups* on page 9-3.

armasm Reference Guide:

- *--maxcache* on page 2-45.

9.5 Built-in variables and constants

Table 9-1 lists the built-in variables defined by `armasm`.

Table 9-1 Built-in variables

`{ARCHITECTURE}`	Holds the name of the selected ARM architecture.
`{AREANAME}`	Holds the name of the current `AREA`.
`{ARMASM_VERSION}`	Holds an integer that increases with each version of `armasm`. The format of the version number is *PVbbbb* where: **P** is the major version; **V** is the minor version; **bbbb** is the build number. — **Note** — The built-in variable `|ads$version|` is deprecated.
`{CODESIZE}`	Is a synonym for `{CONFIG}`.
`{COMMANDLINE}`	Holds the contents of the command line.
`{CONFIG}`	Has the value: • 64 if the assembler is assembling A64 code • 32 if the assembler is assembling A32 code • 16 if the assembler is assembling T32 code.
`{CPU}`	Holds the name of the selected processor. The value of `{CPU}` is derived from the value specified in the `--cpu` option on the command line.
`{ENDIAN}`	Has the value "`big`" if the assembler is in big-endian mode, or "`little`" if it is in little-endian mode.
`{FPU}`	Holds the name of the selected FPU. The default in AArch32 state is `"FP-ARMv8"`. The default in AArch64 state is `"A64"`.
`{INPUTFILE}`	Holds the name of the current source file.
`{INTER}`	Has the boolean value `{True}` if `--apcs=/inter` is set. The default is `{False}`.
`{LINENUM}`	Holds an integer indicating the line number in the current source file.
`{LINENUMUP}`	When used in a macro, holds an integer indicating the line number of the current macro. The value is the same as `{LINENUM}` when used in a non-macro context.
`{LINENUMUPPER}`	When used in a macro, holds an integer indicating the line number of the top macro. The value is the same as `{LINENUM}` when used in a non-macro context.
`{OPT}`	Value of the currently-set listing option. You can use the `OPT` directive to save the current listing option, force a change in it, or restore its original value.
`{PC}` or `.`	Address of current instruction.
`{PCSTOREOFFSET}`	Is the offset between the address of the `STR PC,[…]` or `STM Rb,{…, PC}` instruction and the value of PC stored out. This varies depending on the processor or architecture specified.
`{VAR}` or `@`	Current value of the storage area location counter.

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

```
IF {ARCHITECTURE} = "8-A"
```

The names of the built-in variables can be in uppercase, lowercase, or mixed. For example:

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

Note

All built-in string variables contain case-sensitive values. Relational operations on these built-in variables do not match with strings that contain an incorrect case. Use the command-line options `--cpu` and `--fpu` to determine valid values for `{CPU}`, `{ARCHITECTURE}`, and `{FPU}`.

Table 9-2 lists the built-in Boolean constants defined by `armasm`.

Table 9-2 Built-in Boolean constants

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

Table 9-3 lists the target processor related built-in variables that are predefined by `armasm`. Where the value field is empty, the symbol is a boolean value and the meaning column describes when its value is `{TRUE}`.

Table 9-3 Predefined macros

Name	Value	Meaning
`{TARGET_ARCH_AARCH32}`	*boolean*	`{TRUE}` when assembling for AArch32 state. `{FALSE}` when assembling for AArch64 state.
`{TARGET_ARCH_AARCH64}`	*boolean*	`{TRUE}` when assembling for AArch64 state. `{FALSE}` when assembling for AArch32 state.
`{TARGET_ARCH_ARM}`	*num*	The number of the ARM base architecture of the target processor irrespective of whether the assembler is assembling for A32 or T32. The value is defined as zero when assembling for A64, and eight when assembling for A32/T32.
`{TARGET_ARCH_THUMB}`	*num*	The number of the T32 base architecture of the target processor irrespective of whether the assembler is assembling for A32 or T32. The value is defined as zero when assembling for A64 and five when assembling for A32/T32.
`{TARGET_FEATURE_EXTENSION_REGISTER_COUNT}`	*num*	The number of SIMD or floating-point 64-bit extension registers available.
`{TARGET_FEATURE_CLZ}`	–	If the target processor supports the `CLZ` instruction.
`{TARGET_FEATURE_DIVIDE}`	–	If the target processor supports the hardware divide instructions `SDIV` and `UDIV`.
`{TARGET_FEATURE_DOUBLEWORD}`	–	If the target processor supports doubleword load and store instructions, for example the A32 and T32 instructions `LDRD` and `STRD`.
`{TARGET_FEATURE_DSPMUL}`	–	If the DSP-enhanced multiplier (for example the `SMLA`*`xy`* instruction) is available.
`{TARGET_FEATURE_MULTIPLY}`	–	If the target processor supports long multiply instructions, for example the A32 and T32 instructions `SMULL`, `SMLAL`, `UMULL`, and `UMLAL`.
`{TARGET_FEATURE_MULTIPROCESSING}`	–	If assembling for a target processor with Multiprocessing Extensions.
`{TARGET_FEATURE_NEON}`	–	If the target processor has Advanced SIMD.

Table 9-3 Predefined macros (continued)

Name	Value	Meaning
`{TARGET_FEATURE_NEON_FP16}`	–	If the target processor has Advanced SIMD with half-precision floating-point operations.
`{TARGET_FEATURE_NEON_FP32}`	–	If the target processor has Advanced SIMD with single-precision floating-point operations.
`{TARGET_FEATURE_NEON_INTEGER}`	–	If the target processor has Advanced SIMD with integer operations.
`{TARGET_FEATURE_UNALIGNED}`	–	If the target processor has support for unaligned accesses.
`{TARGET_FPU_SOFTVFP}`	–	If assembling with the option `--fpu=softvfp`.
`{TARGET_FPU_SOFTVFP_VFP}`	–	If assembling for a target processor with softvfp and floating-point hardware, for example `--fpu=softvfp+fp-armv8.`
`{TARGET_FPU_VFP}`	–	If assembling for a target processor with floating-point hardware, without using softvfp, for example `--fpu=fp-armv8`.
`{TARGET_FPU_VFPV2}`	–	If assembling for a target processor with VFPv2.
`{TARGET_FPU_VFPV3}`	–	If assembling for a target processor with VFPv3.
`{TARGET_PROFILE_A}`	–	If assembling for a Cortex™-A profile processor.
`{TARGET_PROFILE_M}`	–	If assembling for a Cortex-M profile processor.
`{TARGET_PROFILE_R}`	–	If assembling for a Cortex-R profile processor.

9.5.1 See also

Reference

armasm Reference Guide:

- *--cpu* on page 2-15
- *--fpu* on page 2-33
- *Versions of armasm* on page 9-10.

9.6 Versions of `armasm`

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

P	is the major version
V	is the minor version
bbbb	is the build number

——— **Note** ———

The built-in variable `|ads$version|` is deprecated.

———————————

9.6.1 See also

Concepts

- *Built-in variables and constants* on page 9-7.

9.7 Diagnostic messages

In addition to the default error, warning, and remark messages, `armasm` can provide more diagnostic messages. By default, these additional diagnostic messages are not displayed. However, you can enable them using the command-line options `--diag_error`, `--diag_warning` and `--diag_remark`.

9.7.1 See also

Concepts

- *Interlocks diagnostics* on page 9-12
- *Automatic IT block generation in T32 code* on page 9-13
- *T32 branch target alignment* on page 9-14
- *T32 code size diagnostics* on page 9-15
- *A32 and T32 instruction portability diagnostics* on page 9-16
- *T32 instruction width* on page 9-17
- *Two pass assembler diagnostics* on page 9-18.

Reference

armasm Reference Guide:

- *--diag_error* on page 2-20.

9.8 Interlocks diagnostics

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

```
armasm --diag_warning 1563
```

Note

- `armasm` does not have an accurate model of the target processor, so these messages are not reliable when used with a multi-issue processor such as Cortex-A8.
- Interlocks diagnostics apply to A32 and T32 code, but not to A64 code.

9.8.1 See also

Concepts

- *Diagnostic messages* on page 9-11
- *Automatic IT block generation in T32 code* on page 9-13
- *T32 branch target alignment* on page 9-14
- *T32 instruction width* on page 9-17.

Reference

armasm Reference Guide:

- *--diag_warning* on page 2-24.

9.9 Automatic IT block generation in T32 code

If you write the following code:

```
AREA x, CODE
THUMB
MOVNE   r0,r1
NOP
IT      NE
MOVNE   r0,r1
END
```

`armasm` generates the following instructions:

```
IT      NE
MOVNE   r0,r1
NOP
IT      NE
MOVNE   r0,r1
```

You can receive warning messages about the automatic generation of `IT` blocks when assembling T32 code. To do this, use the following command-line option when invoking `armasm`:

```
armasm --diag_warning 1763
```

9.9.1 See also

Concepts

- *Diagnostic messages* on page 9-11.

Reference

armasm Reference Guide:

- *--diag_warning* on page 2-24.

9.10 T32 branch target alignment

On some processors, non word-aligned T32 instructions sometimes take one or more additional cycles to execute in loops. This means that it can be an advantage to ensure that branch targets are word-aligned. `armasm` can issue warnings when branch targets in T32 code are not word-aligned. To do this, use the following command-line option when invoking `armasm`:

```
armasm --diag_warning 1604
```

9.10.1 See also

Concepts

- *Diagnostic messages* on page 9-11.

Reference

armasm Reference Guide:

- *--diag_warning* on page 2-24.

9.11 T32 code size diagnostics

In T32 code, some instructions, for example a branch or `LDR` (PC-relative), can be encoded as either a 32-bit or 16-bit instruction. `armasm` chooses the size of the encoding as described in *Instruction width selection in T32*.

`armasm` can issue warnings when an instruction is assembled to a 32-bit T32 instruction where a 16-bit T32 instruction could have been used. To enable this warning, use the following command-line option when invoking `armasm`:

```
armasm --diag_warning 1813
```

9.11.1 See also

Concepts

- *Diagnostic messages* on page 9-11
- *Instruction width selection in T32 code* on page 9-21
- *A64, A32, and T32 instruction sets* on page 3-3.

Reference

armasm Reference Guide:

- *--diag_warning* on page 2-24.

9.12 A32 and T32 instruction portability diagnostics

There are a few UAL instructions that can assemble as either A32 code or T32 code, but not both. You can identify these instructions in the source code using the following command-line option when invoking `armasm`:

```
armasm --diag_warning 1812
```

It warns for any instruction that cannot be assembled in the other instruction set. This is only a hint, and other factors, like relocation availability or target distance might affect the accuracy of the message.

9.12.1 See also

Concepts

- *Diagnostic messages* on page 9-11
- *A64, A32, and T32 instruction sets* on page 3-3.

Reference

armasm Reference Guide:

- *--diag_warning* on page 2-24.

9.13 T32 instruction width

If you use the `.W` specifier, the instruction is encoded in 32 bits even if it could be encoded in 16 bits. You can use a diagnostic warning to detect when a branch instruction could have been encoded in 16 bits, but has been encoded in 32 bits. To do this, use the following command-line option when invoking `armasm`:

```
armasm --diag_warning 1607
```

Note

This diagnostic does not produce a warning for relocated branch instructions, because the final address is not known. The linker might even insert a veneer, if the branch is out of range for a 32-bit instruction.

9.13.1 See also

Concepts

- *Diagnostic messages* on page 9-11.

Reference

armasm Reference Guide:

- *--diag_warning* on page 2-24.

9.14 Two pass assembler diagnostics

armasm is a two pass assembler and the input code that it reads must be identical in both passes. If a symbol is defined after the :DEF: test for that symbol, then the code read in pass one might be different from the code read in pass two. armasm can warn in this situation.

To do this, use the following command-line option when invoking armasm:

```
armasm --diag_warning 1907
```

Example 9-1 shows that the symbol foo is defined after the :DEF: foo test. Assembling this code with --diag_warning 1907 generates the message:

```
Warning A1907W: Test for this symbol has been seen and may cause failure in the second
pass.
```

Example 9-1 Symbol test before symbol definition

```
    AREA x,CODE
    [ :DEF: foo
    ]
foo MOV r3, r4
    END
```

9.14.1 See also

Concepts

- *How the assembler works* on page 2-4
- *Diagnostic messages* on page 9-11
- *Automatic IT block generation in T32 code* on page 9-13
- *T32 branch target alignment* on page 9-14
- *T32 instruction width* on page 9-17.

Reference

armasm Reference Guide:

- *--diag_warning* on page 2-24.

9.15 Address alignment in A32/T32 code

The A bit in the *System Control Register* (SCTLR) controls whether alignment checking is enabled or disabled. If enabled, all unaligned word and halfword transfers cause an alignment exception. If disabled, unaligned accesses are permitted for the `LDR`, `LDRH`, `STR`, `STRH`, `LDRSH`, `LDRT`, `STRT`, `LDRSHT`, `LDRHT`, `STRHT`, and `TBH` instructions. Other data-accessing instructions always cause an alignment exception for unaligned data.

For `STRD` and `LDRD`, the specified address must be word-aligned.

If all your data accesses are aligned, you can use the `--no_unaligned_access` command-line option, to avoid linking in any library functions that support unaligned accesses.

9.15.1 See also

Reference

armasm Reference Guide:

- *--no_unaligned_access* on page 2-56.

9.16 Address alignment in A64 code

If alignment checking is not enabled, then unaligned accesses are permitted for all load and store instructions other than exclusive load, exclusive store, load acquire, and store release instructions. If alignment checking is enabled, then unaligned accesses are not permitted. This means all load and store instructions must use addresses that are aligned to the size of the data being accessed. In other words, addresses for 8-byte transfers must be 8-byte aligned, addresses for 4-byte transfers are 4-byte word aligned, and addresses for 2-byte transfers are 2-byte aligned. Unaligned accesses cause an alignment exception.

For any memory access, if the stack pointer is used as the base register, then it must be quadword aligned. Otherwise it generates a stack alignment exception.

9.16.1 See also

Reference

armasm Reference Guide:

- *--no_unaligned_access* on page 2-56.

9.17 Instruction width selection in T32 code

Some T32 instructions can have either a 16-bit encoding or a 32-bit encoding.

If you do not specify the instruction size, by default:

- For forward reference `LDR`, `ADR`, and `B` instructions, `armasm` always generates a 16-bit instruction, even if that results in failure for a target that could be reached using a 32-bit instruction.
- For external reference `LDR` and `B` instructions, `armasm` always generates a 32-bit instruction.
- In all other cases, `armasm` generates the smallest size encoding that can be output.

If you want to override this behavior, you can use the `.W` or `.N` width specifier to ensure a particular instruction size. `armasm` faults if it cannot generate an instruction with the specified width.

The `.W` specifier is ignored when assembling to A32 code, so you can safely use this specifier in code that might assemble to either A32 or T32 code. However, the `.N` specifier is faulted when assembling to A32 code.

9.17.1 See also

Concepts

- *T32 code size diagnostics* on page 9-15.

Reference

armasm Reference Guide:

- *Instruction width specifiers* on page 3-8.

Chapter 10
Symbols, Literals, Expressions, and Operators

The following topics describe how you can use symbols to represent variables, addresses and constants in code. It also describes how you can combine these with operators to create numeric or string expressions:

10.1 Symbol naming rules

The following general rules apply to symbol names:

- Symbol names must be unique within their scope.
- You can use uppercase letters, lowercase letters, numeric characters, or the underscore character in symbol names. Symbol names are case-sensitive, and all characters in the symbol name are significant.
- Do not use numeric characters for the first character of symbol names, except in numeric local labels.
- Symbols must not use the same name as built-in variable names or predefined symbol names.
- If you use the same name as an instruction mnemonic or directive, use double bars to delimit the symbol name. For example:

  ```
  ||ASSERT||
  ```

  The bars are not part of the symbol.
- You must not use the symbols `|$a|`, `|$t|`, or `|$d|` as program labels. These are mapping symbols that mark the beginning of A32, T32, and A64 code, and data within the object file. You must not use `|$x|`in A64 code.
- Symbols beginning with the characters $v are mapping symbols that relate to floating-point code. ARM recommends you avoid using symbols beginning with $v in your source code.

If you have to use a wider range of characters in symbols, for example, when working with compilers, use single bars to delimit the symbol name. For example:

```
|.text|
```

The bars are not part of the symbol. You cannot use bars, semicolons, or newlines within the bars.

10.1.1 See also

Concepts

- *Numeric local labels* on page 10-12
- *Predeclared core register names in AArch32 state* on page 4-8
- *Predeclared core register names in AArch64 state* on page 5-6
- *Predeclared extension register names in AArch32 state* on page 4-9
- *Predeclared extension register names in AArch64 state* on page 5-7
- *Built-in variables and constants* on page 9-7.

10.2 Variables

The value of a variable can be changed as the assembly proceeds. Variables are local to the assembler. This means that in the generated code or data, every instance of the variable has a fixed value.

Variables are one of the following 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.

The possible values of a logical variable are {TRUE} or {FALSE}.

The range of possible values of a string variable is the same as the range of values of a string expression.

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.

10.2.1 Example

```
a    SETA 100;
L1   MOV R1, #(a*5)  ; In the object file, this is MOV R1, #500
a    SETA 200        ; Value of ‘a’ is 200 only after this point.
                     ; The previous instruction is always MOV R1, #500
     …
     BNE L1          ; When the processor branches to L1, it executes MOV R1, #500
```

10.2.2 See also

Concepts

- *Numeric constants* on page 10-5
- *Numeric expressions* on page 10-16
- *String expressions* on page 10-14
- *Logical expressions* on page 10-19.

Reference

armasm Reference Guide:

- *GBLA, GBLL, and GBLS* on page 10-55
- *LCLA, LCLL, and LCLS* on page 10-65
- *SETA, SETL, and SETS* on page 10-84.

10.3 Numeric constants

Numeric constants are 32-bit integers in A32 and T32 code. 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, `armasm` makes no distinction between $-n$ and $2^{32}-n$.

In A64 code, numeric constants are 64-bit integers. You can set them using unsigned numbers in the range 0 to $2^{64}-1$, or signed numbers in the range -2^{63} to $2^{63}-1$. However, `armasm` makes no distinction between $-n$ and $2^{64}-n$.

`armasm` produces a Numeric Overflow message if you use a constant too large for the instruction set.

Relational operators such as >= use the unsigned interpretation. This means that 0 > –1 is `{FALSE}`.

Use the `EQU` directive to define constants. You cannot change the value of a numeric constant after you define it. You can construct expressions by combining numeric constants and binary operators.

10.3.1 See also

Concept

- *Numeric expressions* on page 10-16
- *Numeric literals* on page 10-17
- *Relational operators* on page 10-27.

Reference

armasm Reference Guide:

- *EQU* on page 10-36.

10.4 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 `armasm` to substitute the string into the source code line before checking the syntax of the line. `armasm` faults if the substituted line is larger than the source line limit.

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. You must set the contents of the variable before you can use it.

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

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

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

10.4.1 Example

```
    ; 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    string1
           GBLS    string2
           GBLS    fixup
           GBLA    count
           ;
count      SETA    14
string1    SETS    "a$$b$count" ; string1 now has value a$b0000000E
string2    SETS    "abc"
fixup      SETS    "|xy$string2.z|"  ; fixup now has value |xyabcz|
|C$$code|  MOV     r4,#16      ; but the label here is C$$code
```

10.4.2 See also

Concept

- *Syntax of source lines in assembly language* on page 6-2
- *Symbol naming rules* on page 10-3.

10.5 Register-relative and PC-relative expressions

armasm supports PC-relative and register-relative expressions.

A register-relative expression evaluates to a named register combined with a numeric expression.

You write a PC-relative expression in source code as a label or the PC, optionally combined with a numeric expression. Some instructions can also accept PC-relative expressions in the form [PC, #number].

If you specify a label, the assembler calculates the offset from the PC value of the current instruction to the address of the label. The assembler encodes the offset in the instruction. If the offset is too large, the assembler produces an error. The offset is either added to or subtracted from the PC value to form the required address.

ARM recommends you write PC-relative expressions using labels rather than PC because the value of PC depends on the instruction set.

Note

- In A32 code, the value of the PC is the address of the current instruction plus 8 bytes.
- In T32 code:
  - For B, BL, CBNZ, and CBZ instructions, the value of the PC is the address of the current instruction plus 4 bytes.
  - For all other instructions that use labels, the value of the PC is the address of the current instruction plus 4 bytes, with bit[1] of the result cleared to 0 to make it word-aligned.
- In A64 code, the value of the PC is the address of the current instruction.

10.5.1 Example

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

10.5.2 See also

Concepts

- *Labels* on page 10-8.

Reference

armasm Reference Guide:

- *MAP* on page 10-70.

10.6 Labels

Labels are symbols representing the memory addresses of instructions or data. The address can be PC-relative, register-relative, or absolute. Labels are local to the source file unless you make them global using the `EXPORT` directive.

The address given by a label is calculated during assembly. `armasm` calculates the address of a label relative to the origin of the section where the label is defined. A reference to a label within the same section can use the PC plus or minus an offset. This is called *PC-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.

10.6.1 See also

Concept

- *Labels for PC-relative addresses* on page 10-9
- *Labels for register-relative addresses* on page 10-10
- *Labels for absolute addresses* on page 10-11.

Reference

armasm Reference Guide:

- *EXPORT or GLOBAL* on page 10-37.

10.7 Labels for PC-relative addresses

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

You can also use the section name of an `AREA` directive as a label for PC-relative addresses. In this case the label points to the first byte of the specified `AREA`. ARM does not recommend using `AREA` names as branch targets because when branching from A32 to T32 state or T32 to A32 state in this way, the processor does not change the state properly.

10.7.1 See also

Reference

armasm Reference Guide:

- *AREA* on page 10-14
- *DCB* on page 10-24
- *DCD and DCDU* on page 10-25
- *DCFD and DCFDU* on page 10-27
- *DCFS and DCFSU* on page 10-28
- *DCI* on page 10-29
- *DCQ and DCQU* on page 10-31
- *DCW and DCWU* on page 10-32.

10.8 Labels for register-relative addresses

These represent a named register plus a numeric value. They are most commonly 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.

Note

Register-relative addresses are not supported in A64 code.

Example 10-1 Storage map definitions

```
        MAP     0,r9
        MAP     0xff,r9
```

10.8.1 See also

Reference

armasm Reference Guide:

- *MAP* on page 10-70
- *SPACE or FILL* on page 10-86
- *DCDO* on page 10-26
- *EQU* on page 10-36.

10.9 Labels for absolute addresses

These are numeric constants. In A32 and T32 code they are integers in the range 0 to 2^{32}–1. In A64 code, they are integers in the range 0 to 2^{64}–1. They address the memory directly. You can use labels to represent absolute addresses using the `EQU` directive. You can specify the absolute address as A32, T32, or data to ensure that the labels are used correctly when referenced in code.

Example 10-2 Defining labels for absolute address

```
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 A32 code
```

10.9.1 See also

Concepts

- *Labels* on page 10-8
- *Labels for PC-relative addresses* on page 10-9
- *Labels for register-relative addresses* on page 10-10.

Reference

armasm Reference Guide:

- *EQU* on page 10-36.

10.10 Numeric local labels

Numeric local labels are a subclass of label. A numeric local label is a number in the range 0-99, optionally followed by a name. Unlike other labels, a numeric local label can be defined many times and the same number can be used for more than one numeric local label in an area.

Numeric local labels do not appear in the object file. This means that, for example, a debugger cannot set a breakpoint directly on a numeric local label, like it can for named local labels kept using the `KEEP` directive.

A numeric local label can be used in place of *`symbol`* in source lines in an assembly language module:

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

A numeric local label is generally used where you might use a PC-relative label.

Numeric 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 when you are generating labels in macros.

The scope of numeric local labels is limited by the `AREA` directive. Use the `ROUT` directive to limit the scope of numeric local labels more tightly. A reference to a numeric local label refers to a matching label within the same scope. If there is no matching label within the scope in either direction, `armasm` generates an error message and the assembly fails.

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

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

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

10.10.1 See also

Concepts

- *Syntax of numeric local labels* on page 10-13
- *Labels* on page 10-8
- *Syntax of source lines in assembly language* on page 6-2.

Reference

armasm Reference Guide:

- *MACRO and MEND* on page 10-67
- *KEEP* on page 10-64
- *ROUT* on page 10-83.

10.11 Syntax of numeric local labels

The syntax of a numeric local label is:

`n{routname}`

The syntax of a reference to a numeric local label is:

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

where:

`n`	is the number of the numeric local label in the range 0-99.
`routname`	is the name of the current scope.
`%`	introduces the reference.
`F`	instructs `armasm` to search forwards only.
`B`	instructs `armasm` to search backwards only.
`A`	instructs `armasm` to search all macro levels.
`T`	instructs `armasm` to look at this macro level only.

If neither `F` nor `B` is specified, `armasm` searches backwards first, then forwards.

If neither `A` nor `T` is specified, `armasm` 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, `armasm` checks it against the name of the nearest preceding `ROUT` directive. If it does not match, `armasm` generates an error message and the assembly fails.

10.11.1 See also

Concepts

- *Numeric local labels* on page 10-12.

Reference

armasm Reference Guide:

- *ROUT* on page 10-83.

10.12 String expressions

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

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

The value of a string expression cannot exceed 5120 characters in length. It can be of zero length.

10.12.1 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
```

10.12.2 See also

Concepts

- *Variables* on page 10-4
- *String literals* on page 10-15
- *Unary operators* on page 10-21
- *String manipulation operators* on page 10-24.

Reference

armasm Reference Guide:

- *SETA, SETL, and SETS* on page 10-84.

10.13 String literals

String literals consist of a series of characters or spaces contained between double quote characters. The length of a string literal is restricted by the length of the input line.

To include a double quote character or a dollar character within the string literal, include the character twice as a pair. For example, you must use `$$` if you require a single `$` in the string.

C string escape sequences are also enabled and can be used within the string, unless `--no_esc` is specified.

10.13.1 Examples

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

10.13.2 See also

Concepts

- *Syntax of source lines in assembly language* on page 6-2.

Reference

armasm Reference Guide:

- *--no_esc* on page 2-48.

10.14 Numeric expressions

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

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

Numeric expressions evaluate to 32-bit integers in A32 and T32 code. 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, `armasm` makes no distinction between $–n$ and $2^{32}–n$.

In A64 code, numeric expressions evaluate to 64-bit integers. You can interpret them as unsigned numbers in the range 0 to $2^{64}–1$, or signed numbers in the range $–2^{63}$ to $2^{63}–1$. However, `armasm` makes no distinction between $–n$ and $2^{64}–n$.

Relational operators such as >= use the unsigned interpretation. This means that 0 > –1 is `{FALSE}`.

10.14.1 Example

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

10.14.2 See also

Concepts

- *Numeric constants* on page 10-5
- *Variables* on page 10-4
- *Numeric literals* on page 10-17
- *Relational operators* on page 10-27
- *Binary operators* on page 10-22.

Reference

armasm Reference Guide:

- *SETA, SETL, and SETS* on page 10-84.

10.15 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 the standard C escape character (\') if you require a single quote. The character must be enclosed within opening and closing single quotes. 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 A32/T32 code, the integer range is 0 to 2^{32}–1, except for the `DCQ`, `DCQU`, `DCO`, and `DCOU` directives.

In A64 code, the integer range is 0 to 2^{64}–1, except for the `DCO` and `DCOU` directives.

Note

- In the `DCQ` and `DCQU` directives, the integer range is 0 to 2^{64} –1.
- In the `DCO` and `DCOU` directives, the integer range is 0 to 2^{128} –1.

10.15.1 Examples

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

10.15.2 See also

Concepts

- *Numeric constants* on page 10-5.

10.16 Floating-point literals

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

`{-}`*`digits`*`E {-}`*`digits`* `{-}{`*`digits`*`}.digits` `{-}{`*`digits`*`}.digitsE {-}`*`digits`* `0x`*`hexdigits`* `&`*`hexdigits`* `0f_`*`hexdigits`* `0d_`*`hexdigits`*.

where:

`digits`	Are sequences of characters using only the digits 0 to 9. You can write `E` in uppercase or lowercase. These forms correspond to normal floating-point notation.
`hexdigits`	Are sequences of characters using only the digits 0 to 9 and the letters A to F or a to f. These forms correspond to the internal representation of the numbers in the computer. Use these forms to enter infinities and NaNs, or if you want to be sure of the exact bit patterns you are using.

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

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

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

The range for single-precision floating-point values is:
- maximum 3.40282347e+38
- minimum 1.17549435e–38.

The range for double-precision floating-point values is:
- maximum 1.79769313486231571e+308
- minimum 2.22507385850720138e–308.

Floating-point numbers are only available if your system has Advanced SIMD or floating-point support.

10.16.1 Examples

```
    DCFD    1E308,-4E-100
    DCFS    1.0
    DCFS    0.02
    DCFD    3.725e15
    DCFS    0x7FC00000              ; Quiet NaN
    DCFD    &FFF0000000000000       ; Minus infinity
```

10.16.2 See also

Concepts
- *Numeric constants* on page 10-5
- *Numeric literals* on page 10-17.

10.17 Logical expressions

Logical expressions consist of combinations of logical literals (`{TRUE}` or `{FALSE}`), logical variables, Boolean operators, relations, and parentheses.

Relations consist of combinations of variables, literals, constants, or expressions with appropriate relational operators.

10.17.1 See also

Concepts

- *Boolean operators* on page 10-28
- *Relational operators* on page 10-27.

10.18 Logical literals

The logical or boolean literals can have one of two values:

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

10.18.1 See also

Concepts

- *Numeric literals* on page 10-17
- *String literals* on page 10-15.

10.19 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 10-1 lists the unary operators that return strings.

Table 10-1 Unary operators that return strings

Operator	Usage	Description
`:CHR:`	`:CHR:A`	Returns the character with ASCII code A.
`:LOWERCASE:`	`:LOWERCASE:string`	Returns the given string, with all uppercase characters converted to lowercase.
`:REVERSE_CC:`	`:REVERSE_CC:cond_code`	Returns the inverse of the condition code in `cond_code`, or an error if `cond_code` does not contain a valid condition code.
`:STR:`	`:STR:A`	In A32 and T32 code, returns an 8-digit hexadecimal string corresponding to a numeric expression, or the string `"T"` or `"F"` if used on a logical expression. In A64 code, returns a 16-digit hexadecimal string.
`:UPPERCASE:`	`:UPPERCASE:string`	Returns the given string, with all lowercase characters converted to uppercase.

Table 10-2 lists the unary operators that return numeric values.

Table 10-2 Unary operators that return numeric or logical values

Operator	Usage	Description
`?`	`?A`	Number of bytes of executable code generated by line defining symbol A.
`+` and `-`	`+A` `-A`	Unary plus. Unary minus. + and – can act on numeric and PC-relative expressions.
`:BASE:`	`:BASE:A`	If A is a PC-relative or register-relative expression, `:BASE:` returns the number of its register component. `:BASE:` is most useful in macros.
`:CC_ENCODING:`	`:CC_ENCODING:cond_code`	Returns the numeric value of the condition code in `cond_code`, or an error if `cond_code` does not contain a valid condition code.
`:DEF:`	`:DEF:A`	`{TRUE}` if A is defined, otherwise `{FALSE}`.
`:INDEX:`	`:INDEX:A`	If A is a register-relative expression, `:INDEX:` returns the offset from that base register. `:INDEX:` is most useful in macros.
`:LEN:`	`:LEN:A`	Length of string A.
`:LNOT:`	`:LNOT:A`	Logical complement of A.
`:NOT:`	`:NOT:A`	Bitwise complement of A (~ is an alias, for example `~A`).
`:RCONST:`	`:RCONST:Rn`	Number of register. In A32/T32 code, 0-15 corresponds to R0-R15. In A64 code, 0-30 corresponds to W0-W30 or X0-X30.

10.19.1 See also

Concepts

- *Binary operators* on page 10-22.

10.20 Binary operators

Binary operators are written between the pair of sub-expressions 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.

10.20.1 See also

Concepts

- *Multiplicative operators* on page 10-23
- *String manipulation operators* on page 10-24
- *Shift operators* on page 10-25
- *Addition, subtraction, and logical operators* on page 10-26
- *Relational operators* on page 10-27
- *Boolean operators* on page 10-28
- *Difference between operator precedence in assembly language and C* on page 10-30.

10.21 Multiplicative operators

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

Table 10-3 shows the multiplicative operators.

Table 10-3 Multiplicative operators

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

You can use the `:MOD:` operator on PC-relative expressions to ensure code is aligned correctly. These alignment checks have the form *PC-relative*`:MOD:`*Constant*. For example:

```
    AREA x,CODE
    ASSERT ({PC}:MOD:4) == 0
    DCB 1
y   DCB 2
    ASSERT (y:MOD:4) == 1
    ASSERT ({PC}:MOD:4) == 2
    END
```

10.21.1 See also

Concepts

- *Register-relative and PC-relative expressions* on page 10-7
- *Binary operators* on page 10-22
- *Numeric literals* on page 10-17
- *Numeric expressions* on page 10-16.

10.22 String manipulation operators

Table 10-4 shows the string manipulation operators. In `CC`, both `A` and `B` must be strings. In the slicing operators `LEFT` and `RIGHT`:

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

Table 10-4 String manipulation operators

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

10.22.1 See also

Concepts

- *String expressions* on page 10-14
- *Numeric expressions* on page 10-16.

10.23 Shift operators

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

Table 10-5 shows the shift operators.

Table 10-5 Shift operators

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

Note

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

10.23.1 See also

Concepts

- *Binary operators* on page 10-22.

10.24 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 10-6 shows addition, subtraction, and logical operators.

Table 10-6 Addition, subtraction, and logical operators

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

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

10.24.1 See also

Concepts

- *Binary operators* on page 10-22.

10.25 Relational operators

Table 10-7 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.
- PC-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`.

Relational operators interpret arithmetic values as unsigned. So the value of `0>-1` is `{FALSE}`.

Table 10-7 Relational operators

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

10.25.1 See also

Concepts

- *Binary operators* on page 10-22
- *Numeric constants* on page 10-5
- *Numeric expressions* on page 10-16.

10.26 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 10-8 shows the Boolean operators.

Table 10-8 Boolean operators

Operator	Alias	Usage	Explanation
`:LAND:`	`&&`	`A:LAND:B`	Logical AND of A and B
`:LEOR:`		`A:LEOR:B`	Logical Exclusive OR of A and B
`:LOR:`	`||`	`A:LOR:B`	Logical OR of A and B

10.26.1 See also

Concepts

- *Binary operators* on page 10-22.

10.27 Operator precedence

`armasm` includes an extensive set of operators for use in expressions. Many of the operators resemble their counterparts in high-level languages such as C.

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.

10.27.1 See also

Concepts

- *Unary operators* on page 10-21
- *Binary operators* on page 10-22
- *Multiplicative operators* on page 10-23
- *String manipulation operators* on page 10-24
- *Shift operators* on page 10-25
- *Addition, subtraction, and logical operators* on page 10-26
- *Relational operators* on page 10-27
- *Boolean operators* on page 10-28
- *Difference between operator precedence in assembly language and C* on page 10-30.

10.28 Difference between operator precedence in assembly language and C

`armasm` does not follow exactly the same order of precedence when evaluating operators as a C compiler.

For example, `(1 + 2 :SHR: 3)` evaluates as `(1 + (2 :SHR: 3))` = 1 in assembly language. The equivalent expression in C evaluates as `((1 + 2) >> 3)` = 0.

ARM recommends you use brackets to make the precedence explicit.

If your code contains an expression that would parse differently in C, and you are not using the `--unsafe` option, `armasm` gives a warning:

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

Table 10-9 shows the order of precedence of operators in assembly language, and a comparison with the order in C (see Table 10-10).

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

Table 10-9 Operator precedence in armasm

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

Table 10-10 Operator precedence in C

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

Table 10-10 Operator precedence in C (continued)

C precedence
|
&&
||

10.28.1 See also

Concepts

- *Operator precedence* on page 10-29.

Chapter 11
Advanced SIMD and Floating-point Programming

The following topics describe Advanced SIMD and floating-point assembly language programming:

11.1 Architecture support for Advanced SIMD and floating-point

Advanced SIMD is optionally available for the ARMv8 architecture. All Advanced SIMD instructions are available on systems that support Advanced SIMD. In A32, some of these instructions are also available on systems that implement the floating-point extension without Advanced SIMD. These are called shared instructions.

The floating-point instruction set supported in A32 is based on VFPv4, but with the addition of some new instructions, including the following:

- Floating-point round to integral.
- Conversion from floating-point to integer with a directed rounding mode.
- Direct conversion between half-precision and double-precision floating-point.
- Floating-point conditional select.

In AArch32 state, Advanced SIMD and floating-point instructions share the same register bank. It consists of thirty-two 64-bit registers, and smaller registers are packed into larger ones, as in ARMv7 and earlier.

In AArch64 state, the SIMD and floating-point register bank is shared, as in AArch32 state, but it includes thirty-two 128-bit registers and has a new register packing model.

Advanced SIMD and floating point instructions in A64 are closely based on VFPv4 and A32, but with new instruction mnemonics and some functional enhancements.

11.1.1 See also

Concepts

ARM C and C++ Libraries and Floating-Point Support User Guide:

- Chapter 4 *Floating-point support*.

Other information

- *Technical Reference Manual* for your processor.

11.2 Extension register bank mapping in AArch32 state

Advanced SIMD and floating-point instructions use the same extension register bank, which is distinct from the ARM register bank. The extension register bank is a collection of registers which can be accessed as either 32-bit, 64-bit, or 128-bit, depending on whether the instruction is Advanced SIMD or floating-point.

Figure 11-1 on page 11-5 shows the three views of the extension register bank, and the overlap between the different size registers. For example, the 128-bit register `Q0` is an alias for two consecutive 64-bit registers `D0` and `D1`, and is also an alias for four consecutive 32-bit registers `S0`, `S1`, `S2`, and `S3`. The 128-bit register `Q8` is an alias for 2 consecutive 64-bit registers `D16` and `D17` but does not have an alias using the 32-bit `S`*n* registers.

Note

If your processor supports both Advanced SIMD and floating-point, all the Advanced SIMD registers overlap with the floating-point registers.

The aliased views enable half-precision, single-precision, and double-precision values, and Advanced SIMD vectors to coexist in different non-overlapped registers at the same time.

You can also use the same overlapped registers to store half-precision, single-precision, and double-precision values, and Advanced SIMD vectors at different times.

Do not attempt to use overlapped 32-bit and 64-bit, or 128-bit registers at the same time because it creates meaningless results.

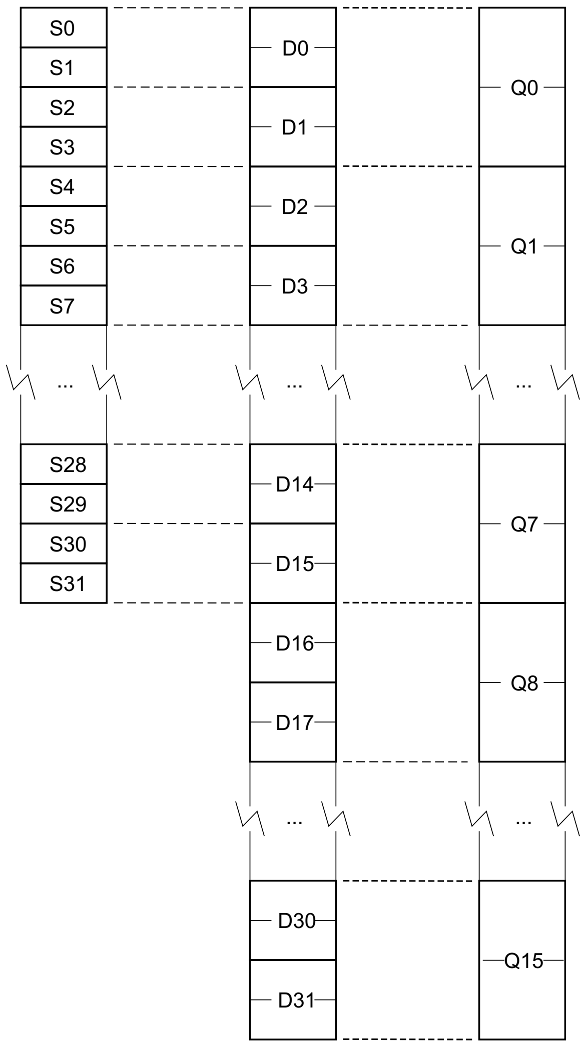


Figure 11-1 Extension register bank in AArch32 state

The mapping between the registers is as follows:

- `S<2n>` maps to the least significant half of `D<n>`
- `S<2n+1>` maps to the most significant half of `D<n>`
- `D<2n>` maps to the least significant half of `Q<n>`
- `D<2n+1>` maps to the most significant half of `Q<n>`.

For example, you can access the least significant half of the elements of a vector in `Q6` by referring to `D12`, and the most significant half of the elements by referring to `D13`.

11.2.1 See also

Concepts

- *Views of the Advanced SIMD register bank in AArch32 state* on page 11-8
- *Views of the floating-point extension register bank in AArch32 state* on page 11-10.

11.3 Extension register bank mapping in AArch64 state

Advanced SIMD and floating-point instructions use the same extension register bank, which is distinct from the ARM register bank. The extension register bank is a collection of registers which can be accessed as 8-bit, 16-bit, 32-bit, 64-bit, or 128-bit, depending on whether the instruction is Advanced SIMD or floating-point.

Figure 11-2 shows the views of the extension register bank, and the overlap between the different size registers.

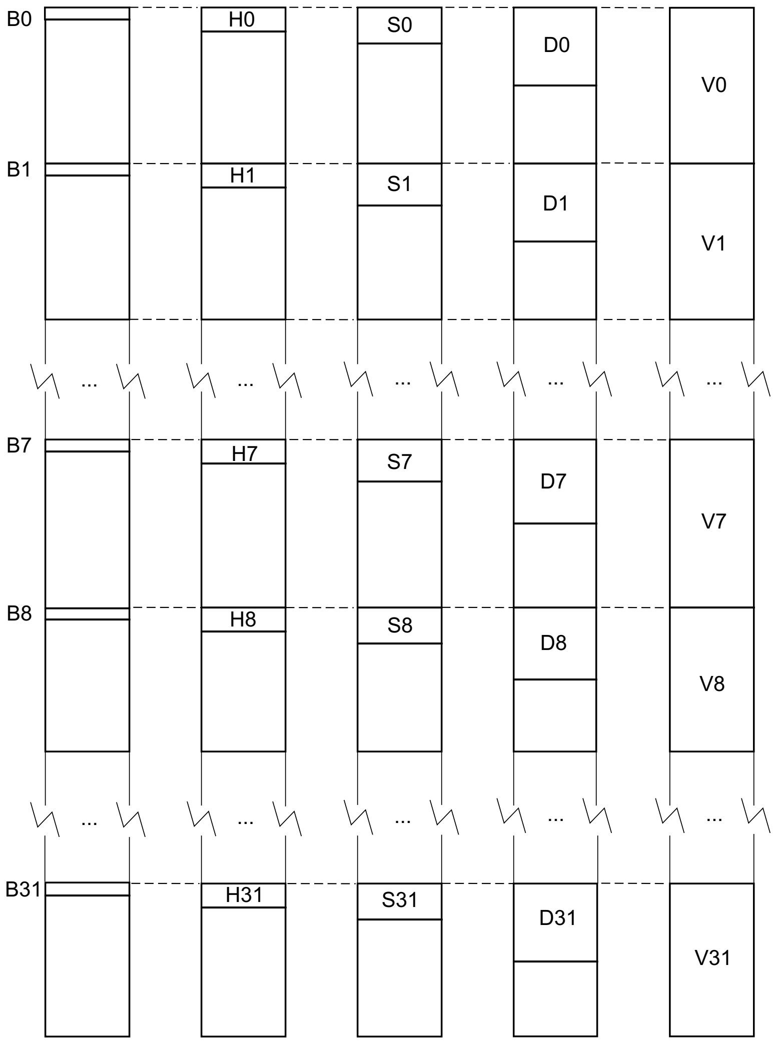


Figure 11-2 Extension register bank in AArch64 state

The mapping between the registers is as follows:

- `D<n>` maps to the least significant half of `V<n>`
- `S<n>` maps to the least significant half of `D<n>`
- `H<n>` maps to the least significant half of `S<n>`
- `B<n>` maps to the least significant half of `H<n>`.

For example, you can access the least significant half of the elements of a vector in `V7` by referring to `D7`.

Registers `Q0`-`Q31` map directly to registers `V0`-`V31`.

11.3.1 See also

Concepts

- *Extension register bank mapping in AArch32 state* on page 11-4
- *Views of the Advanced SIMD register bank in AArch64 state* on page 11-9
- *Views of the floating-point extension register bank in AArch64 state* on page 11-11.

11.4 Views of the Advanced SIMD register bank in AArch32 state

Advanced SIMD can view the extension register bank as:

- Sixteen 128-bit registers, `Q0`-`Q15`.
- Thirty-two 64-bit registers, `D0`-`D31`.
- A combination of registers from these views.

Advanced SIMD views each register as containing a *vector* of 1, 2, 4, 8, or 16 elements, all of the same size and type. Individual elements can also be accessed as *scalars*.

In Advanced SIMD, the 64-bit registers are called doubleword registers and the 128-bit registers are called quadword registers.

11.4.1 See also

Concepts

- *Views of the floating-point extension register bank in AArch32 state* on page 11-10
- *Extension register bank mapping in AArch32 state* on page 11-4.

11.5 Views of the Advanced SIMD register bank in AArch64 state

Advanced SIMD can view the extension register bank as:

- Thirty-two 128-bit registers `V0-V31`.
- Thirty-two 64-bit registers `D0-D31`.
- Thirty-two 32-bit registers `S0-S31`.
- Thirty-two 16-bit registers `H0-H31`.
- Thirty-two 8-bit registers `B0-B31`.
- A combination of registers from these views.

11.5.1 See also

Concepts

- *Views of the floating-point extension register bank in AArch64 state* on page 11-11
- *Extension register bank mapping in AArch64 state* on page 11-6.

11.6 Views of the floating-point extension register bank in AArch32 state

The floating-point extension register bank can be viewed as:

- Thirty-two 64-bit registers, `D0-D31`.
- Thirty-two 32-bit registers, `S0-S31`. Only half of the register bank is accessible in this view.
- A combination of registers from these views.

64-bit floating-point registers are called double-precision registers and can contain double-precision floating-point values. 32-bit floating-point registers are called single-precision registers and can contain either a single-precision or two half-precision floating-point values.

11.6.1 See also

Concepts

- *Views of the Advanced SIMD register bank in AArch32 state* on page 11-8
- *Extension register bank mapping in AArch32 state* on page 11-4.

11.7 Views of the floating-point extension register bank in AArch64 state

The floating-point extension register bank can be viewed as:

- Thirty-two 64-bit registers `D0-D31`.
- Thirty-two 32-bit registers `S0-S31`.
- Thirty-two 16-bit registers `H0-H31`.
- A combination of registers from these views.

11.7.1 See also

Concepts

- *Views of the Advanced SIMD register bank in AArch64 state* on page 11-9
- *Extension register bank mapping in AArch64 state* on page 11-6.

11.8 Differences between A32/T32 and A64 Advanced SIMD and floating-point instruction syntax

The syntax and mnemonics of A64 Advanced SIMD and floating-point instructions are based on those in A32/T32 but with some differences. Table 11-1 describes the main differences.

Table 11-1 Differences in syntax and mnemonics between A32/T32 and A64

A32/T32	A64
All Advanced SIMD and floating-point instruction mnemonics begin with `V`, for example `VMAX`.	The first letter of the instruction mnemonic indicates whether the data type of the instruction is signed, unsigned, floating-point, polynomial or irrelevant. For example `SMAX`, `UMAX`, or `FMAX`.
A mnemonic qualifier specifies the type and width of elements in a vector. For example, in the following instruction, `U32` means 32-bit unsigned integers: `VMAX.U32 Q0, Q1, Q2`	A register qualifier specifies the data width and the number of elements in the register. For example, in the following instruction `.4S` means 4 32-bit elements: `UMAX V0.4S, V1.4S, V2.4S`
The 128-bit vector registers are named Q0-Q15 and the 64-bit vector registers are named D0-D31.	All vector registers are named Vn , where n is a register number between 0 and 31. You only use one of the qualified register names Qn, Dn, Sn, Hn or Bn when referring to a scalar register, to indicate the number of significant bits.
You load a single element into one or more vector registers by appending an index to each register individually, for example: `VLD4.8 {D0[3], D1[3], D2[3], D3[3]}, [R0]`	You load a single element into one or more vector registers by appending the index to the register list, for example: `LD4 {V0.B, V1.B, V2.B, V3.B}[3], [X0]`
You can append a condition code to most floating-point and Advanced SIMD instruction mnemonics to make them conditional.	A64 has no conditionally executed floating-point or Advanced SIMD instructions.
The floating-point select instruction, `VSEL`, is unconditionally executed but uses a condition code as an operand. You append the condition code to the mnemonic, for example: `VSELEQ.F32 S1,S2,S3`	There are several floating-point instructions that use a condition code as an operand. You specify the condition code in the final operand position, for example: `FCSEL S1,S2,S3,EQ`
`L`, `W` and `N` suffixes indicate long, wide and narrow variants of Advanced SIMD data processing instructions. A32/T32 Advanced SIMD does not include vector narrowing or widening second part instructions.	`L`, `W` and `N` suffixes indicate long, wide and narrow variants of Advanced SIMD data processing instructions. You can additionally append a `2` to implement the second part of a narrowing or widening operation, for example: `UADDL2 V0.4S, V1.8H, V2.8H ; take input from 4 high-numbered lanes of V1 and V2`
A32/T32 Advanced SIMD does not include vector reduction instructions.	The `V` Advanced SIMD mnemonic suffix identifies vector reduction instructions, in which the operand is a vector and the result a scalar, for example: `ADDV S0, V1.4S`
The `P` mnemonic qualifier which indicates pairwise instructions is a prefix, for example, `VPADD`.	The `P` mnemonic qualifier is a suffix, for example `ADDP`.

11.8.1 See also

Concepts

- *Advanced SIMD and floating-point data types in A32/T32 instructions* on page 11-17
- *Conditional execution of T32 Advanced SIMD and floating-point instructions* on page 11-15

- *Advanced SIMD scalars* on page 11-24
- *Long Advanced SIMD instructions* on page 11-20
- *Wide Advanced SIMD instructions* on page 11-21
- *Narrow Advanced SIMD instructions* on page 11-22
- *Syntax differences between UAL and A64 assembly language* on page 7-4.

Reference

- *VSEL* on page 4-89
- *FCSEL* on page 7-10.

11.9 Load values to SIMD and floating-point registers

To load a register with a floating-point immediate value, use `VMOV` in A32 or `FMOV` in A64. Both instructions exist in scalar and vector forms.

The A32 Advanced SIMD instructions `VMOV` and `VMVN` can also load integer immediates. The A64 Advanced SIMD instructions to load integer immediates are `MOVI` and `MVNI`.

You can load any 64-bit integer, single-precision, or double-precision floating-point value from a literal pool using the `VLDR` pseudo-instruction.

11.9.1 See also

Reference

armasm Reference Guide:

- *VMOV, VMVN (immediate)* on page 4-67
- *VMOV* on page 4-66
- *FMOV (scalar, immediate)* on page 7-32
- *MOVI (vector)* on page 9-135
- *MVNI (vector)* on page 9-139
- *VLDR pseudo-instruction* on page 4-61.

11.10 Conditional execution of T32 Advanced SIMD and floating-point instructions

You cannot use any of the following Advanced SIMD and floating-point instructions in an IT block:

- `VRINT{A, N, P, M}` (floating-point).
- `VSEL` (floating-point).
- `VCVT{A, N, P, M}` (Advanced SIMD and floating-point).
- `VMAXNM` (Advanced SIMD and floating-point).
- `VMINNM` (Advanced SIMD and floating-point).
- `VRINT{N, X, A, Z, M, P}` (Advanced SIMD).
- All instructions in the Crypto extension.

In addition, specifying any other Advanced SIMD or floating-point instruction in an IT block is deprecated.

11.10.1 See also

Reference

armasm Reference Guide:

- *IT* on page 3-63.

Other information

- *ARMv8-A Architecture Reference Manual*
  `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0487a.b/index.html`.

11.11 Floating-point exceptions in A32/T32 instructions

The Advanced SIMD and floating-point extensions record the following floating-point exceptions in the FPSCR cumulative flags:

Invalid operation

The exception is caused if the result of an operation has no mathematical value or cannot be represented.

Division by zero

The exception is caused if a divide operation has a zero divisor and a dividend that is not zero, an infinity or a NaN.

Overflow

The exception is caused if the absolute value of the result of an operation, produced after rounding, is greater than the maximum positive normalized number for the destination precision.

Underflow

The exception is caused if the absolute value of the result of an operation, produced before rounding, is less than the minimum positive normalized number for the destination precision, and the rounded result is inexact.

Inexact

The exception is caused if the result of an operation is not equivalent to the value that would be produced if the operation were performed with unbounded precision and exponent range.

Input denormal

The exception is caused if a denormalized input operand is replaced in the computation by a zero.

In the *armasm Reference Guide*, in the descriptions of the instructions that can cause floating-point exceptions, there is a subsection listing the exceptions. If there is no such subsection, that instruction cannot cause any floating-point exception.

11.11.1 See also

Concepts

- *Flush-to-zero mode* on page 11-28.

Other information

- *Technical Reference Manual* for your floating-point hardware
- *ARM Architecture Reference Manual* `http://infocenter.arm.com/help/topic/com.arm.doc.subset.architecture.reference/`.

11.12 Advanced SIMD and floating-point data types in A32/T32 instructions

Data type specifiers in Advanced SIMD and floating-point instructions consist of a letter indicating the type of data, usually followed by a number indicating the width. They are separated from the instruction mnemonic by a point. Table 11-2 shows the data types available in Advanced SIMD instructions. Table 11-3 shows the data types available in floating-point instructions.

Table 11-2 Advanced SIMD data types

	8-bit	16-bit	32-bit	64-bit
Unsigned integer	`U8`	`U16`	`U32`	`U64`
Signed integer	`S8`	`S16`	`S32`	`S64`
Integer of unspecified type	`I8`	`I16`	`I32`	`I64`
Floating-point number	not available	`F16`	`F32` (or `F`)	not available
Polynomial over {0,1}	`P8`	`P16`	not available	not available

Table 11-3 Floating-point data types

	16-bit	32-bit	64-bit
Unsigned integer	`U16`	`U32`	not available
Signed integer	`S16`	`S32`	not available
Floating-point number	`F16`	`F32` (or `F`)	`F64` (or `D`)

The datatype of the second (or only) operand is specified in the instruction.

Note

- Most instructions have a restricted range of permitted data types. See the instruction descriptions for details. However, the data type description is flexible:
  - — If the description specifies I, you can also use the S or U data types.
  - — If only the data size is specified, you can specify a type (I, S, U, P or F).
  - — If no data type is specified, you can specify a data type.

11.12.1 See also

Concepts

- *Polynomial arithmetic over {0,1}* on page 11-26.

11.13 Advanced SIMD vectors

An Advanced SIMD operand can be a vector or a scalar. An Advanced SIMD vector can be a 64-bit doubleword vector or a 128-bit quadword vector.

In A32/T32 Advanced SIMD instructions, the size of the elements in an Advanced SIMD vector is specified by a datatype suffix appended to the mnemonic. In A64 Advanced SIMD instructions, the size and number of the elements in an Advanced SIMD vector are specified by a suffix appended to the register.

Doubleword vectors can contain:

- Eight 8-bit elements.
- Four 16-bit elements.
- Two 32-bit elements.
- One 64-bit element.

Quadword vectors can contain:

- Sixteen 8-bit elements.
- Eight 16-bit elements.
- Four 32-bit elements.
- Two 64-bit elements.

11.13.1 See also

Concepts

- *Advanced SIMD scalars* on page 11-24
- *Extension register bank mapping in AArch32 state* on page 11-4
- *Extended notation in A32/T32 code* on page 11-25
- *Advanced SIMD and floating-point data types in A32/T32 instructions* on page 11-17.

11.14 Normal Advanced SIMD instructions

Many A32/T32 and A64 Advanced SIMD data processing instructions are available in Normal, Long, Wide, Narrow, and saturating variants.

Normal instructions can operate on any of the vector types, and produce result vectors the same size, and usually the same type, as the operand vectors.

You can specify that the operands and result of a normal A32/T32 Advanced SIMD instruction must all be quadwords by appending a `Q` to the instruction mnemonic. If you do this, `armasm` produces an error if the operands or result are not quadwords.

11.14.1 See also

Concepts

- *Advanced SIMD vectors* on page 11-18
- *Long Advanced SIMD instructions* on page 11-20
- *Wide Advanced SIMD instructions* on page 11-21
- *Narrow Advanced SIMD instructions* on page 11-22
- *Saturating Advanced SIMD instructions* on page 11-23.

11.15 Long Advanced SIMD instructions

Long instructions operate on doubleword vector operands and produce a quadword vector result. The elements of the result are usually twice the width of the elements of the operands, and the same type.

Long instructions are specified using an `L` appended to the instruction mnemonic.

11.15.1 See also

Concepts

- *Advanced SIMD vectors* on page 11-18
- *Normal Advanced SIMD instructions* on page 11-19
- *Wide Advanced SIMD instructions* on page 11-21
- *Narrow Advanced SIMD instructions* on page 11-22
- *Saturating Advanced SIMD instructions* on page 11-23.

11.16 Wide Advanced SIMD instructions

Wide instructions operate on one doubleword vector operand and one quadword vector operand. They produce a quadword vector result. The elements of the result and the first operand are twice the width of the elements of the second operand.

Wide instructions are specified using a `W` appended to the instruction mnemonic.

11.16.1 See also

Concepts

- *Advanced SIMD vectors* on page 11-18
- *Normal Advanced SIMD instructions* on page 11-19
- *Long Advanced SIMD instructions* on page 11-20
- *Narrow Advanced SIMD instructions* on page 11-22
- *Saturating Advanced SIMD instructions* on page 11-23.

11.17 Narrow Advanced SIMD instructions

Narrow instructions operate on quadword vector operands, and produce a doubleword vector result. The elements of the result are half the width of the elements of the operands.

Narrow instructions are specified using an `N` appended to the instruction mnemonic.

11.17.1 See also

Concepts

- *Advanced SIMD vectors* on page 11-18
- *Normal Advanced SIMD instructions* on page 11-19
- *Long Advanced SIMD instructions* on page 11-20
- *Wide Advanced SIMD instructions* on page 11-21
- *Saturating Advanced SIMD instructions* on page 11-23.

11.18 Saturating Advanced SIMD instructions

Saturating instructions saturate the result to the value of the upper limit or lower limit if the result overflows or underflows. The saturation limits depend on the datatype of the instruction. See Table 11-4 for the ranges that Advanced SIMD saturating instructions saturate to, where *x* is the result of the operation.

Table 11-4 Advanced SIMD saturation ranges

Data type	Saturation range of *x*
Signed byte (`S8`)	$-2^{7} <= x < 2^{7}$
Signed halfword (`S16`)	$-2^{15} <= x < 2^{15}$
Signed word (`S32`)	$-2^{31} <= x < 2^{31}$
Signed doubleword (`S64`)	$-2^{63} <= x < 2^{63}$
Unsigned byte (`U8`)	$0 <= x < 2^{8}$
Unsigned halfword (`U16`)	$0 <= x < 2^{16}$
Unsigned word (`U32`)	$0 <= x < 2^{32}$
Unsigned doubleword (`U64`)	$0 <= x < 2^{64}$

Saturating instructions are specified using a `Q` prefix. In A32/T32 Advanced SIMD instructions, this is inserted between the `V` and the instruction mnemonic, or between the S or U and the mnemonic in A64 Advanced SIMD instructions.

11.18.1 See also

Concepts

- *Advanced SIMD vectors* on page 11-18
- *Normal Advanced SIMD instructions* on page 11-19
- *Long Advanced SIMD instructions* on page 11-20
- *Wide Advanced SIMD instructions* on page 11-21
- *Narrow Advanced SIMD instructions* on page 11-22.

Reference

armasm Reference Guide:

- *Saturating instructions* on page 3-19.

11.19 Advanced SIMD scalars

Some Advanced SIMD instructions act on scalars in combination with vectors. Advanced SIMD scalars can be 8-bit, 16-bit, 32-bit, or 64-bit. In A32/T32 Advanced SIMD instructions, the instruction syntax refers to a single element in a vector register using an index, *x*, into the vector, so that `Dm[x]` is the *x*th element in vector `Dm`. In A64 Advanced SIMD instructions, you append the index to the element size specifier, so that `Vm.D[x]` is the *x*th doubleword element in vector `Vm`.

In A64 Advanced SIMD scalar instructions, you refer to registers using a name that indicates the number of significant bits. The names are `Bn`, `Hn`, `Sn`, or `Dn`, where `n` is the register number (0-31). The unused high bits are ignored on a read and set to zero on a write.

Other than A32/T32 Advanced SIMD multiply instructions, instructions that access scalars can access any element in the register bank.

A32/T32 Advanced SIMD multiply instructions only allow 16-bit or 32-bit scalars, and can only access the first 32 scalars in the register bank. That is, in multiply instructions:

- 16-bit scalars are restricted to registers D0-D7, with `x` in the range 0-3.
- 32-bit scalars are restricted to registers D0-D15, with `x` either 0 or 1.

11.19.1 See also

Concepts

- *Extension register bank mapping in AArch32 state* on page 11-4
- *Advanced SIMD vectors* on page 11-18.

11.20 Extended notation in A32/T32 code

`armasm` implements an extension to the architectural Advanced SIMD and floating-point assembly syntax, called *extended notation*. This extension allows you to include datatype information or scalar indexes in register names. If you do this, you do not need to include the datatype or scalar index information in every instruction.

Register names can be any of the following:

Untyped The register name specifies the register, but not what datatype it contains, nor any index to a particular scalar within the register.

Untyped with scalar index

The register name specifies the register, but not what datatype it contains, It specifies an index to a particular scalar within the register.

Typed The register name specifies the register, and what datatype it contains, but not any index to a particular scalar within the register.

Typed with scalar index

The register name specifies the register, what datatype it contains, and an index to a particular scalar within the register.

Use the `SN`, `DN`, and `QN` directives to create typed and scalar registers.

Note

Extended notation is not supported for A64 code.

11.20.1 See also

Concepts

- *Advanced SIMD and floating-point data types in A32/T32 instructions* on page 11-17
- *Advanced SIMD vectors* on page 11-18
- *Advanced SIMD scalars* on page 11-24.

Reference

armasm Reference Guide:

- *QN, DN, and SN* on page 10-75.

11.21 Polynomial arithmetic over {0,1}

The coefficients 0 and 1 are manipulated using the rules of Boolean arithmetic:

- 0 + 0 = 1 + 1 = 0
- 0 + 1 = 1 + 0 = 1
- 0 * 0 = 0 * 1 = 1 * 0 = 0
- 1 * 1 = 1.

That is, adding two polynomials over {0,1} is the same as a bitwise exclusive OR, and multiplying two polynomials over {0,1} is the same as integer multiplication except that partial products are exclusive-ORed instead of being added.

11.21.1 See also

Concepts

- *Advanced SIMD and floating-point data types in A32/T32 instructions* on page 11-17.

11.22 Advanced SIMD and floating-point system registers in AArch32 state

For exception levels using AArch32, the following Advanced SIMD and floating-point system registers are accessible in all Advanced SIMD and floating-point implementations:

- FPSCR, the floating-point status and control register.
- FPEXC, the floating-point exception register.
- FPSID, the floating-point system ID register.

A particular Advanced SIMD or floating-point implementation can have additional registers.

11.22.1 See also

Concepts

- *Read-Modify-Write procedure* on page 7-29.

Other information

- *Technical Reference Manual* for your floating-point hardware
- *ARM Architecture Reference Manual* `http://infocenter.arm.com/help/topic/com.arm.doc.subset.architecture.reference/`.

11.23 Flush-to-zero mode

Some floating-point implementations 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.

Advanced SIMD and floating-point flush-to-zero mode preserves the sign bit.

Advanced SIMD always uses flush-to-zero mode.

11.23.1 See also

Concepts

- *When to use flush-to-zero mode* on page 11-29
- *The effects of using flush-to-zero mode* on page 11-30
- *Operations not affected by flush-to-zero mode* on page 11-31.

11.24 When to use flush-to-zero mode

You must 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 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 select flush-to-zero mode in one of the following ways:

- In A32 code, by setting the FZ bit in the FPSCR to 1. You do this using the `VMRS` and `VMSR` instructions.
- In A64 code, by setting the FZ bit in the FPCR to 1. You do this using the `MRS` and `MSR` instructions.

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.

11.24.1 See also

Concepts

- *The effects of using flush-to-zero mode* on page 11-30
- *Flush-to-zero mode* on page 11-28.

Reference

armasm Reference Guide:

- *VMRS and VMSR* on page 4-74
- *MRS* on page 5-104
- *MSR (register)* on page 5-106

11.25 The effects of using flush-to-zero mode

With certain exceptions, 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.

In flush-to-zero mode, an Input Denormal exception occurs whenever a denormalized number is used as an operand. An Underflow exception occurs when a result is flushed-to-zero.

11.25.1 See also

Concepts

- *Operations not affected by flush-to-zero mode* on page 11-31
- *Flush-to-zero mode* on page 11-28.

11.26 Operations not affected by flush-to-zero mode

The following Advanced SIMD and floating-point 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 (`VMOV`, `VMVN`, `V{Q}ABS`, and `V{Q}NEG`)
- duplicate (`VDUP`)
- swap (`VSWP`)
- load and store (`VLDR` and `VSTR`)
- load multiple and store multiple (`VLDM` and `VSTM`)
- transfer between extension registers and ARM general-purpose registers (`VMOV`).

11.26.1 See also

Concepts

- *The effects of using flush-to-zero mode* on page 11-30
- *Flush-to-zero mode* on page 11-28.

Reference

armasm Reference Guide:

- *VDUP* on page 4-49
- *VSWP* on page 4-94
- *VLDR and VSTR* on page 4-54
- *VLDM, VSTM, VPOP, and VPUSH* on page 4-53
- *VMOV, VMVN (register)* on page 4-68
- *VABS, VNEG, and VSQRT* on page 4-28
- *V{Q}ABS and V{Q}NEG* on page 4-20
- *VMOV (between two ARM registers and an extension register)* on page 4-69
- *VMOV (between an ARM register and an Advanced SIMD scalar)* on page 4-70
- *VMOV (between one ARM register and single precision floating-point register)* on page 4-71.