ARM® Compiler toolchain

Version 5.01

Assembler Reference

ARM Compiler toolchain
Assembler Reference

Release Information

The following changes have been made to this book.

Change History

Date	Issue	Confidentiality	Change
May 2010	A	Non-Confidential	ARM Compiler toolchain v4.1 Release
30 September 2010	B	Non-Confidential	Update 1 for ARM Compiler toolchain v4.1
28 January 2011	C	Non-Confidential	Update 2 for ARM Compiler toolchain v4.1 Patch 3
30 April 2011	D	Non-Confidential	ARM Compiler toolchain v5.0 Release
29 July 2011	E	Non-Confidential	Update 1 for ARM Compiler toolchain v5.0
30 September 2011	F	Non-Confidential	ARM Compiler toolchain v5.01 Release
29 February 2012	G	Non-Confidential	Document update 1 for ARM Compiler toolchain v5.01 Release

Proprietary Notice

Confidentiality Status

Product Status

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

Web Address

`http://www.arm.com`

Contents
ARM Compiler toolchain Assembler Reference

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.
`monospace` (with `mono` underlined)	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 0489G
- 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
Assembler command line options

The following topics describe the ARM® Compiler toolchain assembler command line syntax and the command line options accepted by the assembler, `armasm`:

2.1 Assembler command line syntax

The command for invoking the ARM assembler is:

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

where:

`options`	are commands to the assembler. You can invoke the assembler with any combination of options separated by spaces. You can specify values for some options. To specify a value for an option, use either '`=`' (*`option`*`=`*`value`*) or a space character (*`option value`*).
`inputfile`	can be one or more assembly source files separated by spaces. Input files must be UAL, or pre-UAL ARM or Thumb® assembly language source files.

See also

Using the Compiler:

- *Order of compiler command-line options* on page 3-11.

2.2 Assembler command line options

The following command line options are supported by the assembler:

- *--16* on page 2-4
- *--32* on page 2-5
- *--apcs=qualifier…qualifier* on page 2-5
- *--arm* on page 2-6
- *--arm_only* on page 2-6
- *--bi* on page 2-6
- *--bigend* on page 2-6
- *--brief_diagnostics* on page 2-7
- *--checkreglist* on page 2-7
- *--compatible=name* on page 2-7
- *--cpreproc* on page 2-7
- *--cpreproc_opts=options* on page 2-8
- *--cpu=list* on page 2-8
- *--cpu=name* on page 2-8
- *--debug* on page 2-9
- *--depend=dependfile* on page 2-9
- *--depend_format=string* on page 2-9
- *--device=list* on page 2-9
- *--device=name* on page 2-10
- *--diag_error=tag{, tag}* on page 2-10
- *--diag_remark=tag{, tag}* on page 2-11
- *--diag_style=style* on page 2-11
- *--diag_suppress=tag{, tag}* on page 2-11
- *--diag_warning=tag{, tag}* on page 2-12
- *--dllexport_all* on page 2-12
- *--dwarf2* on page 2-12
- *--dwarf3* on page 2-12
- *--errors=errorfile* on page 2-12
- *--execstack* on page 2-13
- *--exceptions* on page 2-13
- *--exceptions_unwind* on page 2-13
- *--fpmode=model* on page 2-13
- *--fpu=list* on page 2-14
- *--fpu=name* on page 2-14
- *-g* on page 2-16
- *--help* on page 2-16
- *-idir{,dir, …}* on page 2-16
- *--keep* on page 2-16
- *--length=n* on page 2-16
- *--li* on page 2-16
- *--library_type=lib* on page 2-16
- *--licretry* on page 2-17
- *--list=file* on page 2-17
- *--list=* on page 2-18
- *--littleend* on page 2-18

2.2.1 --16

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

See also

- *--thumb* on page 2-24
- *ARM, THUMB, THUMBX, CODE16 and CODE32* on page 6-56.

2.2.2 --32

This option is a synonym for `--arm`.

See also

- *--arm* on page 2-6.

2.2.3 --apcs=*qualifier…qualifier*

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

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

Note

AAPCS qualifiers do not affect the code produced by the assembler. They are an assertion by the programmer that the code in *`inputfile`* complies with a particular variant of AAPCS. They cause attributes to be set in the object file produced by the assembler. The linker uses these attributes to check compatibility of files, and to select appropriate library variants.

Values for *`qualifier`* are:

`none`	Specifies that *`inputfile`* does not use AAPCS. AAPCS registers are not set up. Other qualifiers are not permitted if you use `none`.
`/interwork, /nointerwork`	`/interwork` specifies that the code in the *inputfile* can interwork between ARM and Thumb safely. The default is `/nointerwork`.
`/inter, /nointer`	Are synonyms for `/interwork` and `/nointerwork`.
`/ropi, /noropi`	`/ropi` specifies that the code in *inputfile* is Read-Only Position-Independent (ROPI). The default is `/noropi`.
`/pic, /nopic`	Are synonyms for `/ropi` and `/noropi`.
`/rwpi, /norwpi`	`/rwpi` specifies that the code in *inputfile* is Read-Write Position-Independent (RWPI). The default is `/norwpi`.
`/pid, /nopid`	Are synonyms for `/rwpi` and `/norwpi`.
`/fpic, /nofpic`	`/fpic` specifies that the code in *inputfile* is read-only independent and references to addresses are suitable for use in a Linux shared object. The default is `/nofpic`.
`/hardfp, /softfp`	Requests hardware or software floating-point linkage. This enables the procedure call standard to be specified separately from the version of the floating-point hardware available through the `--fpu` option. It is still possible to specify the procedure call standard by using the `--fpu` option, but ARM recommends you use `--apcs`. If floating-point support is not permitted (for example, because `--fpu=none` is specified, or because of other means), then `/hardfp` and `/softfp` are ignored. If floating-point support is permitted and the *softfp* calling convention is used (`--fpu=softvfp` or `--fpu=softvfp+vfp...`), then `/hardfp` gives an error.

Note

You must specify at least one *qualifier*. If you specify more than one *qualifier*, ensure that there are no spaces or commas between the individual qualifiers in the list.

Example

```
armasm --apcs=/inter/ropi inputfile.s
```

See also

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

Compiler Reference:
- *--apcs=qualifer...qualifier* on page 3-11.

Developing Software for ARM Processors:
- Chapter 5 *Interworking ARM and Thumb*.

2.2.4 --arm

This option instructs the assembler to interpret instructions as ARM instructions. It does not, however, guarantee ARM-only code in the object file. This is the default. Using this option is equivalent to specifying the `ARM` or `CODE32` directive at the start of the source file.

See also
- *--32* on page 2-5
- *--arm_only*
- *ARM, THUMB, THUMBX, CODE16 and CODE32* on page 6-56.

2.2.5 --arm_only

This option instructs the assembler to only generate ARM code. This is similar to `--arm` but also has the property that the assembler does not permit the generation of any Thumb code.

See also
- *--arm*.

2.2.6 --bi

This option is a synonym for `--bigend`.

See also
- *--bigend*
- *--littleend* on page 2-18

2.2.7 --bigend

This option instructs the assembler to assemble code suitable for a big-endian ARM. The default is `--littleend`.

See also

- *--littleend* on page 2-18.

2.2.8 --brief_diagnostics

This option instructs the assembler to use a shorter form of the diagnostic output. In this form, the original source line is not displayed and the error message text is not wrapped when it is too long to fit on a single line. The default is `--no_brief_diagnostics`.

See also

- *--diag_error=tag{, tag}* on page 2-10
- *--diag_warning=tag{, tag}* on page 2-12.

2.2.9 --checkreglist

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

This option is deprecated. Use `--diag_warning 1206` instead.

See also

- *--diag_warning=tag{, tag}* on page 2-12.

2.2.10 --compatible=*name*

This option specifies a second processor or architecture, *`name`*, for which the assembler generates compatible code.

When you specify a processor or architecture name using `--compatible`, valid values of *`name`* for both the `--cpu` and `--compatible` options are restricted to those shown in Table 2-1 and must not be from the same group.

Table 2-1 Compatible processor or architecture combinations

Group 1	`ARM7TDMI`, `4T`
Group 2	`Cortex`™`-M0`, `Cortex-M1`, `Cortex-M3`, `Cortex-M4`, `7-M`, `6-M`, `6S-M`, `SC300`, `SC000`

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

Example

```
armasm --cpu=arm7tdmi --compatible=cortex-m3 inputfile.s
```

See also

- *--cpu=name* on page 2-8.

2.2.11 --cpreproc

This option instructs the assembler to call `armcc` to preprocess the input file before assembling it.

See also

- *--cpreproc_opts=options*.

Using the Assembler:

- *Using the C preprocessor* on page 7-22.

2.2.12 --cpreproc_opts=*options*

This option enables the assembler to pass compiler options to `armcc` when using the C preprocessor.

options is a comma-separated list of options and their values.

Example

```
armasm --cpreproc --cpreproc_opts='-DDEBUG=1' inputfile.s
```

See also

- *--cpreproc* on page 2-7.

Using the Assembler:

- *Using the C preprocessor* on page 7-22.

2.2.13 --cpu=list

This option lists the supported CPU and architecture names that can be used with the `--cpu` *`name`* option.

Example

```
armasm --cpu=list
```

See also

- *--cpu=name*.

2.2.14 --cpu=*name*

This option sets the target CPU. Some instructions produce either errors or warnings if assembled for the wrong target CPU.

Valid values for *`name`* are architecture names such as `4T`, `5TE`, or `6T2`, or part numbers such as `ARM7TDMI`®. The default is `ARM7TDMI`.

Note

ARMv7 is not a recognized ARM architecture. Using `--cpu=7` generates only Thumb instructions that are common in the ARMv7-A, ARMv7-R, and ARMv7-M architectures.

When you specify an alternative processor or architecture name using `--compatible`, valid values of *`name`* for both the `--cpu` and `--compatible` options are restricted to those shown in Table 2-1 on page 2-7.

Example

```
armasm --cpu=Cortex-M3 inputfile.s
```

See also

- *--cpu=list* on page 2-8
- *--unsafe* on page 2-24
- *--compatible=name* on page 2-7
- *ARM Architecture Reference Manual*, `http://infocenter.arm.com/help/topic/com.arm.doc.subset.architecture.reference/index.html`.

2.2.15 --debug

This option instructs the assembler to generate DWARF debug tables. `--debug` is a synonym for `-g`. The default is DWARF 3.

Note

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

See also

- *--dwarf2* on page 2-12
- *--dwarf3* on page 2-12
- *--keep* on page 2-16.

2.2.16 --depend=*dependfile*

This option instructs the assembler to save source file dependency lists to *`dependfile`*. These are suitable for use with make utilities.

See also

- *--md* on page 2-18
- *--depend_format=string*.

2.2.17 --depend_format=*string*

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

The value of *`string`* can be one of:

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

See also

- *--depend=dependfile*.

2.2.18 --device=list

This option lists the supported device names that can be used with the `--device=`*`name`* option.

Note

This option is deprecated.

See also

- *--device=name*.

2.2.19 --device=*name*

This option selects a specified device as the target and sets the associated processor settings.

Note

This option is deprecated.

See also

- *--device=list* on page 2-9
- *--cpu=name* on page 2-8
- *--device=name* on page 3-68 in the *Compiler Reference*.

2.2.20 --diag_error=*tag*{, *tag*}

Diagnostic messages output by the assembler can be identified by a tag in the form of `{prefix}number`, where the `prefix` is A. The `--diag_error` option sets the diagnostic messages that have the specified tags to the error severity.

You can specify more than one tag with these options by separating each tag using a comma.You can specify the optional assembler prefix `A` before the tag number. If any prefix other than A is included, the message number is ignored.

Table 2-2 shows the meaning of the term *severity* used in the option descriptions.

Table 2-2 Severity of diagnostic messages

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

You can set the `tag` to `warning` to treat all warnings as errors.

See also

- *--brief_diagnostics* on page 2-7
- *--diag_warning=tag{, tag}* on page 2-12
- *--diag_suppress=tag{, tag}* on page 2-11.

2.2.21 --diag_remark=*tag*{, *tag*}

Diagnostic messages output by the assembler can be identified by a tag in the form of {`prefix`}`number`, where the `prefix` is A. The `--diag_remark` option sets the diagnostic messages that have the specified tags to the remark severity.

You can specify more than one tag with these options by separating each tag using a comma.You can specify the optional assembler prefix `A` before the tag number. If any prefix other than A is included, the message number is ignored.

See also

- *--brief_diagnostics* on page 2-7
- *--diag_error=tag{, tag}* on page 2-10.

2.2.22 --diag_style=*style*

This option instructs the assembler to display diagnostic messages using the specified `style`, where `style` is one of:

`arm`	Display messages using the ARM assembler style. This is the default if `--diag_style` is not specified.
`ide`	Include the line number and character count for the line that is in error. These values are displayed in parentheses.
`gnu`	Display messages using the GNU style.

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

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

See also

- *--brief_diagnostics* on page 2-7
- *--diag_style=style*.

2.2.23 --diag_suppress=*tag*{, *tag*}

Diagnostic messages output by the assembler can be identified by a tag in the form of {`prefix`}`number`, where the `prefix` is A. The `--diag_suppress` option disables the diagnostic messages that have the specified tags.

You can specify more than one tag with these options by separating each tag using a comma.

For example, to suppress the warning messages that have numbers `1293` and `187`, use the following command:

```
armasm --diag_suppress=1293,187
```

You can specify the optional assembler prefix `A` before the tag number. For example:

```
armasm --diag_suppress=A1293,A187
```

If any prefix other than A is included, the message number is ignored. Diagnostic message tags can be cut and pasted directly into a command line.

You can also set the `tag` to:

- `warning`, to suppress all warnings
- `error`, to suppress all downgradeable errors.

See also

- *--diag_error=tag{, tag}* on page 2-10.

2.2.24 --diag_warning=*tag*{, *tag*}

Diagnostic messages output by the assembler can be identified by a tag in the form of {`prefix`}`number`, where the `prefix` is A. The `--diag_warning` option sets the diagnostic messages that have the specified tags to the warning severity.

You can specify more than one tag with these options by separating each tag using a comma. You can specify the optional assembler prefix `A` before the tag number. If any prefix other than A is included, the message number is ignored.

You can set the `tag` to `error` to downgrade the severity of all downgradeable errors to warnings.

See also

- *--diag_error=tag{, tag}* on page 2-10.

2.2.25 --dllexport_all

This option gives all exported global symbols `STV_PROTECTED` visibility in ELF rather than `STV_HIDDEN`, unless overridden by source directives.

See also

- *EXPORT or GLOBAL* on page 6-67.

2.2.26 --dwarf2

This option can be used with `--debug`, to instruct the assembler to generate DWARF 2 debug tables.

See also

- *--debug* on page 2-9
- *--dwarf3*.

2.2.27 --dwarf3

This option can be used with `--debug`, to instruct the assembler to generate DWARF 3 debug tables. This is the default if `--debug` is specified.

See also

- *--debug* on page 2-9
- *--dwarf2*.

2.2.28 --errors=*errorfile*

This option instructs the assembler to output error messages to `errorfile`.

2.2.29 --execstack

This option generates a .note.GNU-stack section marking the stack as executable.

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

```
    AREA    |.note.GNU-stack|,ALIGN=0,READONLY,NOALLOC,CODE
```

In the absence of `--execstack` and `--no_execstack`, the .note.GNU-stack section is not generated unless it is specified by the `AREA` directive.

See also

- *--no_execstack* on page 2-18
- *AREA* on page 6-61.

2.2.30 --exceptions

This option instructs the assembler to switch on exception table generation for all functions defined by `FUNCTION` (or `PROC`) and `ENDFUNC` (or `ENDP`).

See also

- *--no_exceptions* on page 2-19
- *--exceptions_unwind*
- *--no_exceptions_unwind* on page 2-19
- *FRAME UNWIND ON* on page 6-47
- *FUNCTION or PROC* on page 6-47
- *ENDFUNC or ENDP* on page 6-49
- *FRAME UNWIND OFF* on page 6-47.

2.2.31 --exceptions_unwind

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

For finer control, use `FRAME UNWIND ON` and `FRAME UNWIND OFF` directives.

See also

- *--no_exceptions_unwind* on page 2-19
- *--exceptions*
- *--no_exceptions* on page 2-19
- *FRAME UNWIND ON* on page 6-47
- *FRAME UNWIND OFF* on page 6-47
- *FUNCTION or PROC* on page 6-47
- *ENDFUNC or ENDP* on page 6-49.

2.2.32 --fpmode=*model*

This option specifies the floating-point model, and sets library attributes and floating-point optimizations to select the most suitable library when linking.

Note

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

`model` can be one of:

`none`	Source code is not permitted to use any floating-point type or floating point instruction. This option overrides any explicit --fpu=*name* option.
`ieee_full`	All facilities, operations, and representations guaranteed by the IEEE standard are available in single and double-precision. Modes of operation can be selected dynamically at runtime.
`ieee_fixed`	IEEE standard with round-to-nearest and no inexact exception.
`ieee_no_fenv`	IEEE standard with round-to-nearest and no exceptions. This mode is compatible with the Java floating-point arithmetic model.
`std`	IEEE finite values with denormals flushed to zero, round-to-nearest and no exceptions. It is C and C++ compatible. This is the default option. Finite values are as predicted by the IEEE standard. It is not guaranteed that NaNs and infinities are produced in all circumstances defined by the IEEE model, or that when they are produced, they have the same sign. Also, it is not guaranteed that the sign of zero is that predicted by the IEEE model.
`fast`	Some value altering optimizations, where accuracy is sacrificed to fast execution. This is not IEEE compatible, and is not standard C.

Example

```
armasm --fpmode ieee_full inputfile.s
```

See also

- *--fpu=name*.

2.2.33 --fpu=list

This option lists the supported FPU names that can be used with the `--fpu=name` option.

Example

```
armasm --fpu=list
```

See also

- *--fpu=name*
- *--fpmode=model* on page 2-13.

2.2.34 --fpu=*name*

This option selects the target *floating-point unit* (FPU) architecture. If you specify this option it overrides any implicit FPU set by the `--cpu` option. The assembler produces an error if the FPU you specify explicitly is incompatible with the CPU. Floating-point instructions also produce either errors or warnings if assembled for the wrong target FPU.

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

Valid values for `name` are:

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

`vfpv3`	Selects hardware floating-point unit conforming to architecture VFPv3.
`vfpv3_fp16`	Selects hardware floating-point unit conforming to architecture VFPv3 with half-precision floating-point extension.
`vfpv3_d16`	Selects hardware floating-point unit conforming to architecture VFPv3-D16.
`vfpv3_d16_fp16`	Selects hardware floating-point unit conforming to architecture VFPv3-D16 with half-precision floating-point extension.
`vfpv4`	Selects hardware floating-point unit conforming to architecture VFPv4.
`vfpv4_d16`	Selects hardware floating-point unit conforming to architecture VFPv4-D16.
`fpv4-sp`	Selects hardware floating-point unit conforming to the single precision variant of architecture FPv4.
`vfpv2`	Selects hardware floating-point unit conforming to architecture VFPv2.
`softvfp`	Selects software floating-point linkage. This is the default if you do not specify a `--fpu` option and the `--cpu` option selected does not imply a particular FPU.
`softvfp+vfpv2`	Selects a floating-point library with software floating-point linkage that uses VFP instructions. This is otherwise equivalent to using `--fpu vfpv2`.
`softvfp+vfpv3`	Selects a floating-point library with software floating-point linkage that uses VFP instructions. This is otherwise equivalent to using `--fpu vfpv3`.
`softvfp+vfpv3_fp16`	Selects a floating-point library with software floating-point linkage that uses VFP instructions. This is otherwise equivalent to using `--fpu vfpv3_fp16`.
`softvfp+vfpv3_d16`	Selects a floating-point library with software floating-point linkage that uses VFP instructions. This is otherwise equivalent to using `--fpu vfpv3_d16`.
`softvfp+vfpv3_d16_fp16`	Selects a floating-point library with software floating-point linkage that uses VFP instructions. This is otherwise equivalent to using `--fpu vfpv3_d16_fp16`.
`softvfp+vfpv4`	Selects a floating-point library with software floating-point linkage that uses VFP instructions. This is otherwise equivalent to using `--fpu vfpv4`.
`softvfp+vfpv4_d16`	Selects a floating-point library with software floating-point linkage that uses VFP instructions. This is otherwise equivalent to using `--fpu vfpv4_d16`.
`softvfp+fpv4-sp`	Selects a floating-point library with software floating-point linkage that uses VFP instructions. This is otherwise equivalent to using `--fpu fpv4-sp`.

See also

- *--fpmode=model* on page 2-13.

2.2.35 -g

This option is a synonym for `--debug`.

See also

- *--debug* on page 2-9.

2.2.36 --help

This option instructs the assembler to show a summary of the available command line options.

2.2.37 -i*dir*{,*dir*, …}

This option adds directories to the source file include path. Any directories added using this option have to be fully qualified.

See also

- *GET or INCLUDE* on page 6-70.

2.2.38 --keep

This option instructs the assembler to keep local labels in the symbol table of the object file, for use by the debugger.

2.2.39 --length=*n*

This option sets the listing page length to *n*. Length zero means an unpaged listing. The default is 66 lines.

See also

- *--list=file* on page 2-17.

2.2.40 --li

This option is a synonym for `--littleend`.

See also

- *--littleend* on page 2-18
- *--bigend* on page 2-6.

2.2.41 --library_type=*lib*

This option enables the relevant library selection to be used at link time.

Where `lib` can be one of:

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

——— **Note** ———

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

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

See also

- *Building an application with microlib* on page 3-7 in the *Using ARM C and C++ Libraries and Floating Point Support*
- *--library_type=lib* on page 3-129 in the *Compiler Reference.*

2.2.42 --licretry

If you are using floating licenses, this option makes up to 10 attempts to obtain a license when you invoke `armasm`.

Use this option if your builds are failing to obtain a license from your license server, and only after you have ruled out any other problems with the network or the license server setup.

ARM recommends that you place this option in the `ARMCC`*`n`*`_ASMOPT` environment variable. In this way, you do not have to modify your build files.

See also

- *Toolchain environment variables* on page 2-14 in the *Introducing the ARM Compiler toolchain*
- *--licretry* on page 3-130 in the *Compiler Reference*
- *--licretry* on page 2-99 in the *Linker Reference*
- *--licretry on page 4-51* in *Using the fromelf Image Converter*
- *ARM® DS-5™ License Management Guide*, `http://infocenter.arm.com/help/topic/com.arm.doc.dui0577-/index.html`.

2.2.43 --list=*file*

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

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

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

- `--no_terse`
- `--width`
- `--length`
- `--xref`.

See also

- *--no_terse* on page 2-20
- *--width=n* on page 2-26
- *--length=n* on page 2-16
- *--xref* on page 2-26.

2.2.44 --list=

This option instructs the assembler to send the detailed assembly language listing to `*inputfile*.lst`.

Note

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

See also

- *--list=file* on page 2-17.

2.2.45 --littleend

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

See also

- *--bigend* on page 2-6.

2.2.46 -m

This option instructs the assembler to write source file dependency lists to `stdout`.

See also

- *--md*.

2.2.47 --maxcache=*n*

This option sets the maximum source cache size to *n* bytes. The default is 8MB. `armasm` gives a warning if size is less than 8MB.

2.2.48 --md

This option instructs the assembler to write source file dependency lists to `*inputfile*.d`.

See also

- *-m*.

2.2.49 --no_code_gen

This option instructs the assembler to exit after pass 1. No object file is generated. This option is useful if you only want to check the syntax of the source code or directives.

2.2.50 --no_esc

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

2.2.51 --no_execstack

This option generates a .note.GNU-stack section marking the stack as non-executable.

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

```
    AREA    |.note.GNU-stack|,ALIGN=0,READONLY,NOALLOC
```

In the absence of --execstack and --no_execstack, the .note.GNU-stack section is not generated unless it is specified by the AREA directive.

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

Table 2-3 Specifying a command line option and an AREA directive for GNU-stack sections

	--execstack **command line option**	--no_execstack **command line option**
execstack AREA directive	execstack	execstack
no_execstack AREA directive	execstack	no_execstack

See also

- *--execstack* on page 2-13
- *AREA* on page 6-61.

2.2.52 --no_exceptions

This option instructs the assembler to switch off exception table generation. No tables are generated. This is the default.

See also

- *--exceptions* on page 2-13
- *--exceptions_unwind* on page 2-13
- *--no_exceptions_unwind*
- *FRAME UNWIND ON* on page 6-47
- *FRAME UNWIND OFF* on page 6-47.

2.2.53 --no_exceptions_unwind

This option instructs the assembler to produce *nounwind* tables for every function.

See also

- *--exceptions* on page 2-13
- *--no_exceptions*
- *--exceptions_unwind* on page 2-13.

2.2.54 --no_hide_all

This option gives all exported and imported global symbols STV_DEFAULT visibility in ELF rather than STV_HIDDEN, unless overridden by source directives.

See also

- *EXPORT or GLOBAL* on page 6-67
- *IMPORT and EXTERN* on page 6-71.

2.2.55 --no_project

This option disables the use of a project template file.

Note

This option is deprecated.

See also

- *--project=filename, --no_project* on page 3-174 in the *Compiler Reference*
- *--project=filename* on page 2-21
- *--reinitialize_workdir* on page 2-23
- *--workdir=directory* on page 2-26.

2.2.56 --no_reduce_paths

This option disables the elimination of redundant pathname information in file paths. This is the default setting. This option is valid for Windows systems only.

See also

- *--reduce_paths* on page 2-22
- *--reduce_paths, --no_reduce_paths* on page 3-177 in the *Compiler Reference*.

2.2.57 --no_regs

This option instructs the assembler not to predefine register names.

This option is deprecated. Use `--regnames=none` instead.

See also

- *--regnames=none* on page 2-22
- *Predeclared core register names* on page 3-13 in *Using the Assembler*
- *Predeclared extension register names* on page 3-14 in *Using the Assembler*
- *Predeclared XScale register names* on page 3-15 in *Using the Assembler*
- *Predeclared coprocessor names* on page 3-16 in *Using the Assembler*.

2.2.58 --no_terse

This option instructs the assembler to show the lines of assembly code that have been skipped because of conditional assembly in the list file. When this option is not specified on the command line, the assembler does not output the skipped assembly code to the list file.

This option turns off the terse flag. By default the terse flag is on.

See also

- *--list=file* on page 2-17.

2.2.59 --no_unaligned_access

This option instructs the assembler to set an attribute in the object file to disable the use of unaligned accesses.

See also

- *--unaligned_access* on page 2-24.

2.2.60 --no_warn

This option turns off warning messages.

See also

- *--diag_warning=tag{, tag}* on page 2-12.

2.2.61 -o *filename*

This option names the output object file. If this option is not specified, the assembler creates an object filename of the form `inputfilename.o`. This option is case-sensitive.

2.2.62 --pd

This option is a synonym for `--predefine`.

See also

- *--predefine "directive"*.

2.2.63 --predefine "*directive*"

This option instructs the assembler to pre-execute one of the `SET` directives. This is useful for conditional assembly.

The `directive` is one of the `SETA`, `SETL`, or `SETS` directives. You must enclose `directive` in quotes, for example:

```
armasm --predefine "VariableName SETA 20" inputfile.s
```

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

The variable name is case-sensitive. The variables defined using the command line are global to the assembler source files specified on the command line.

Note

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

See also

- *--pd*
- *GBLA, GBLL, and GBLS* on page 6-4
- *SETA, SETL, and SETS* on page 6-7
- *Assembly conditional on a variable being defined* on page 6-35.

2.2.64 --project=*filename*

This option enables the use of a project template file.

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

Note

This option is deprecated.

See also

- *--project=filename, --no_project* on page 3-174 in the *Compiler Reference*
- *--no_project* on page 2-20
- *--reinitialize_workdir* on page 2-23
- *--workdir=directory* on page 2-26.

2.2.65 --reduce_paths

This option enables the elimination of redundant pathname information in file paths. This option is valid for Windows systems only.

Windows systems impose a 260 character limit on file paths. Where relative pathnames exist whose absolute names expand to longer than 260 characters, you can use the `--reduce_paths` option to reduce absolute pathname length by matching up directories with corresponding instances of `..` and eliminating the directory/`..` sequences in pairs.

Note

ARM recommends that you avoid using long and deeply nested file paths, in preference to minimizing path lengths using the `--reduce_paths` option.

See also

- *--no_reduce_paths* on page 2-20
- *--reduce_paths, --no_reduce_paths* on page 3-177 in the *Compiler Reference*.

2.2.66 --regnames=none

This option instructs the assembler not to predefine register names.

See also

- *--regnames=callstd*
- *--regnames=all* on page 2-23
- *--no_regs* on page 2-20
- *Predeclared core register names* on page 3-13 in *Using the Assembler*
- *Predeclared extension register names* on page 3-14 in *Using the Assembler*
- *Predeclared XScale register names* on page 3-15 in *Using the Assembler*
- *Predeclared coprocessor names* on page 3-16 in *Using the Assembler*.

2.2.67 --regnames=callstd

This option defines additional register names based on the AAPCS variant that you are using as specified by the `--apcs` option.

See also

- *--apcs=qualifier...qualifier* on page 2-5

- *--regnames=none* on page 2-22
- *--regnames=all*.

2.2.68 --regnames=all

This option defines all AAPCS registers regardless of the value of `--apcs`.

- *--apcs=qualifier...qualifier* on page 2-5
- *--regnames=none* on page 2-22
- *--regnames=callstd* on page 2-22.

2.2.69 --reinitialize_workdir

This option enables you to re-initialize the project template working directory.

Note

This option is deprecated.

See also

- *--reinitialize_workdir* on page 3-178 in the *Compiler Reference*.
- *--project=filename* on page 2-21
- *--no_project* on page 2-20
- *--workdir=directory* on page 2-26

2.2.70 --report-if-not-wysiwyg

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

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

2.2.71 --show_cmdline

This option outputs the command line used by the assembler. It shows the command line after processing by the assembler, and can be useful to check:

- the command line a build system is using
- how the assembler is interpreting the supplied command line, for example, the ordering of command line options.

The commands are shown normalized, and the contents of any via files are expanded.

The output is sent to the standard output stream (stdout).

See also

- *--via=file* on page 2-25.

2.2.72 --split_ldm

This option instructs the assembler to fault `LDM` and `STM` instructions with a large number of registers. Use of this option is deprecated.

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

- 5, for `LDM`s that do not load the PC
- 4, for `LDM`s that load the PC.

This option faults `STM` instructions if the maximum number of registers transferred exceeds 5.

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

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

Also, avoiding large multiple register transfers:

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

2.2.73 --thumb

This option instructs the assembler to interpret instructions as Thumb instructions, using the UAL syntax. This is equivalent to a `THUMB` directive at the start of the source file.

See also

- *--arm* on page 2-6
- *ARM, THUMB, THUMBX, CODE16 and CODE32* on page 6-56.

2.2.74 --thumbx

This option instructs the assembler to interpret instructions as ThumbEE instructions, using the UAL syntax. This is equivalent to a `THUMBX` directive at the start of the source file.

See also

- *ARM, THUMB, THUMBX, CODE16 and CODE32* on page 6-56.

2.2.75 --unaligned_access

This option instructs the assembler to set an attribute in the object file to enable the use of unaligned accesses.

See also

- *--no_unaligned_access* on page 2-20.

2.2.76 --unsafe

This option enables instructions from differing architectures to be assembled without error. It changes corresponding error messages to warning messages. It also suppresses warnings about operator precedence.

See also

- *--diag_error=tag{, tag}* on page 2-10
- *--diag_warning=tag{, tag}* on page 2-12
- *Binary operators* on page 8-22 in *Using the Assembler*.

2.2.77 --untyped_local_labels

This option causes the assembler not to set the Thumb bit for the address of a local label referenced in an `LDR` pseudo instruction.

When this option is not used, if you reference a local label in an `LDR` pseudo-instruction, and the label is in Thumb code, then the assembler sets the Thumb bit (bit 0) of the address. You can then use the address as the target for a `BX` or `BLX` instruction.

If you require the actual address of the local label, without the Thumb bit set, then use the `--untyped_local_labels` option. Note that if you then use the address in a branch (register) instruction, the assembler treats it as an ARM code address, causing the branch to arrive in ARM state, meaning it would interpret this code as ARM instructions.

Example

```
    THUMB
    ...
1
    ...
    LDR r0,=%B1 ; r0 contains the address of local label "1",
                ; Thumb bit is not set if --untyped_local_labels was used
    ...
```

See also

- *LDR pseudo-instruction* on page 3-159
- *B, BL, BX, BLX, and BXJ* on page 3-117
- *Local labels* on page 8-12 in *Using the Assembler*.

2.2.78 --version_number

This option displays the version of `armasm` being used. The format is *PVVbbbb*, where:

P	is the major version
VV	is the minor version
bbbb	is the build number.

For example, version 5.01 build 0019 is displayed as `5010019`.

See also

- *--vsn* on page 2-26
- *--help* on page 2-16.

2.2.79 --via=*file*

This option instructs the assembler to open *`file`* and read in command line arguments to the assembler.

See also

- Appendix B *Via File Syntax* in the *Compiler Reference*.

2.2.80 --vsn

This option displays the version information and license details. For example:

```
>armasm --vsn
ARM Assembler, N.nn [Build num]
license_type
Software supplied by: ARM Limited
```

See also

- *--version_number* on page 2-25
- *--help* on page 2-16.

2.2.81 --width=*n*

This option sets the listing page width to *n*. The default is 79 characters.

See also

- *--list=file* on page 2-17.

2.2.82 --workdir=*directory*

This option enables you to provide a working directory for a project template.

Note

This option is deprecated.

See also

- *--project=filename* on page 2-21
- *--no_project* on page 2-20
- *--reinitialize_workdir* on page 2-23
- *--workdir=directory* on page 3-226 in the *Compiler Reference.*

2.2.83 --xref

This option instructs the assembler to list cross-referencing information on symbols, including where they were defined and where they were used, both inside and outside macros. The default is off.

See also

- *--list=file* on page 2-17.

Chapter 3
ARM and Thumb Instructions

The following topics describe the ARM, Thumb (all versions), and ThumbEE instructions supported by the ARM assembler:

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

3.1 Instruction summary

Table 3-1 gives an overview of the instructions available in the ARM, Thumb, and ThumbEE instruction sets. Use it to locate individual instructions and pseudo-instructions.

Note

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

Table 3-1 Location of instructions

Mnemonic	Brief description	See	Arch. [a]
`ADC`, `ADD`	Add with Carry, Add	page 3-51	All
`ADR`	Load program or register-relative address (short range)	page 3-24	All
`ADRL` pseudo-instruction	Load program or register-relative address (medium range)	page 3-156	x6M
`AND`	Logical AND	page 3-57	All
`ASR`	Arithmetic Shift Right	page 3-72	All
`B`	Branch	page 3-117	All
`BFC`, `BFI`	Bit Field Clear and Insert	page 3-110	T2
`BIC`	Bit Clear	page 3-57	All
`BKPT`	Breakpoint	page 3-135	5
`BL`	Branch with Link	page 3-117	All
`BLX`	Branch with Link, change instruction set	page 3-117	T
`BX`	Branch, change instruction set	page 3-117	T
`BXJ`	Branch, change to Jazelle®	page 3-117	J, x7M
`CBZ, CBNZ`	Compare and Branch if {Non}Zero	page 3-123	T2
`CDP`	Coprocessor Data Processing operation	page 3-126	x6M
`CDP2`	Coprocessor Data Processing operation	page 3-126	5, x6M
`CHKA`	Check array	page 3-153	EE
`CLREX`	Clear Exclusive	page 3-43	K, x6M
`CLZ`	Count leading zeros	page 3-59	5, x6M
`CMN`, `CMP`	Compare Negative, Compare	page 3-60	All
`CPS`	Change Processor State	page 3-141	6
`DBG`	Debug	page 3-147	7
`DMB`, `DSB`	Data Memory Barrier, Data Synchronization Barrier	page 3-148	7, 6M
`ENTERX`, `LEAVEX`	Change state to or from ThumbEE	page 3-152	EE
`EOR`	Exclusive OR	page 3-57	All
`ERET`	Exception Return	page 3-37	7VE
`HB`, `HBL`, `HBLP`, `HBP`	Handler Branch, branches to a specified handler	page 3-154	EE

Table 3-1 Location of instructions (continued)

Mnemonic	Brief description	See	Arch. [a]
`ISB`	Instruction Synchronization Barrier	page 3-148	7, 6M
`IT`	If-Then	page 3-120	T2
`LDC`	Load Coprocessor	page 3-132	x6M
`LDC2`	Load Coprocessor	page 3-132	5, x6M
`LDM`	Load Multiple registers	page 3-30	All
`LDR`	Load Register with word	page 3-9	All
`LDR` pseudo-instruction	Load Register pseudo-instruction	page 3-159	All
`LDRB`	Load Register with byte	page 3-9	All
`LDRBT`	Load Register with byte, user mode	page 3-9	x6M
`LDRD`	Load Registers with two words	page 3-9	5E, x6M
`LDREX`	Load Register Exclusive	page 3-40	6, x6M
`LDREXB`, `LDREXH`	Load Register Exclusive Byte, Halfword	page 3-40	K, x6M
`LDREXD`	Load Register Exclusive Doubleword	page 3-40	K, x7M
`LDRH`	Load Register with halfword	page 3-9	All
`LDRHT`	Load Register with halfword, user mode	page 3-9	T2
`LDRSB`	Load Register with signed byte	page 3-9	All
`LDRSBT`	Load Register with signed byte, user mode	page 3-9	T2
`LDRSH`	Load Register with signed halfword	page 3-9	All
`LDRSHT`	Load Register with signed halfword, user mode	page 3-9	T2
`LDRT`	Load Register with word, user mode	page 3-9	x6M
`LSL`, `LSR`	Logical Shift Left, Logical Shift Right	page 3-72	All
`MAR`	Move from Registers to 40-bit Accumulator	page 3-150	XScale
`MCR`	Move from Register to Coprocessor	page 3-127	x6M
`MCR2`	Move from Register to Coprocessor	page 3-127	5, x6M
`MCRR`	Move from Registers to Coprocessor	page 3-127	5E, x6M
`MCRR2`	Move from Registers to Coprocessor	page 3-127	6, x6M
`MIA`, `MIAPH`, `MIAxy`	Multiply with Internal 40-bit Accumulate	page 3-95	XScale
`MLA`	Multiply Accumulate	page 3-77	x6M
`MLS`	Multiply and Subtract	page 3-77	T2
`MOV`	Move	page 3-62	All
`MOVT`	Move Top	page 3-65	T2
`MOV32` pseudo-instruction	Move 32-bit immediate to register	page 3-158	T2
`MRA`	Move from 40-bit Accumulator to Registers	page 3-150	XScale

Table 3-1 Location of instructions (continued)

Mnemonic	Brief description	See	Arch. [a]
`MRC`	Move from Coprocessor to Register	page 3-128	x6M
`MRC2`	Move from Coprocessor to Register	page 3-128	5, x6M
`MRRC`	Move from Coprocessor to Registers	page 3-128	5E, x6M
`MRRC2`	Move from Coprocessor to Registers	page 3-128	6, x6M
`MRS`	Move from PSR to register	page 3-137	All
`MRS`	Move from system Coprocessor to Register	page 3-130	7A, 7R
`MSR`	Move from register to PSR	page 3-139	All
`MSR`	Move from Register to system Coprocessor	page 3-129	7A, 7R
`MUL`	Multiply	page 3-77	All
`MVN`	Move Not	page 3-62	All
`NOP`	No Operation	page 3-144	All
`ORN`	Logical OR NOT	page 3-57	T2
`ORR`	Logical OR	page 3-57	All
`PKHBT`, `PKHTB`	Pack Halfwords	page 3-114	6, 7EM
`PLD`	Preload Data	page 3-28	5E, x6M
`PLDW`	Preload Data with intent to Write	page 3-28	7MP
`PLI`	Preload Instruction	page 3-28	7
`PUSH`, `POP`	PUSH registers to stack, POP registers from stack	page 3-33	All
`QADD`, `QDADD`, `QDSUB`, `QSUB`	Saturating Arithmetic	page 3-98	5E, 7EM
`QADD8`, `QADD16`, `QASX`, `QSUB8`, `QSUB16`, `QSAX`	Parallel signed Saturating Arithmetic	page 3-103	6, 7EM
`RBIT`	Reverse Bits	page 3-70	T2
`REV`, `REV16`, `REVSH`	Reverse byte order	page 3-70	6
`RFE`	Return From Exception	page 3-35	T2, x7M
`ROR`	Rotate Right Register	page 3-72	All
`RRX`	Rotate Right with Extend	page 3-72	x6M
`RSB`	Reverse Subtract	page 3-51	All
`RSC`	Reverse Subtract with Carry	page 3-51	x7M
`SADD8`, `SADD16`, `SASX`	Parallel signed arithmetic	page 3-103	6, 7EM
`SBC`	Subtract with Carry	page 3-51	All
`SBFX`, `UBFX`	Signed, Unsigned Bit Field eXtract	page 3-111	T2
`SDIV`	Signed divide	page 3-75	7M, 7R
`SEL`	Select bytes according to APSR GE flags	page 3-68	6, 7EM

Table 3-1 Location of instructions (continued)

Mnemonic	Brief description	See	Arch. [a]
`SETEND`	Set Endianness for memory accesses	page 3-143	6, x7M
`SEV`	Set Event	page 3-145	K, 6M
`SHADD8`, `SHADD16`, `SHASX`, `SHSUB8`, `SHSUB16`, `SHSAX`	Parallel signed Halving arithmetic	page 3-103	6, 7EM
`SMC`	Secure Monitor Call	page 3-142	Z
`SMLAxy`	Signed Multiply with Accumulate (32 <= 16 x 16 + 32)	page 3-81	5E, 7EM
`SMLAD`	Dual Signed Multiply Accumulate (32 <= 32 + 16 x 16 + 16 x 16)	page 3-90	6, 7EM
`SMLAL`	Signed Multiply Accumulate (64 <= 64 + 32 x 32)	page 3-79	x6M
`SMLALxy`	Signed Multiply Accumulate (64 <= 64 + 16 x 16)	page 3-84	5E, 7EM
`SMLALD`	Dual Signed Multiply Accumulate Long (64 <= 64 + 16 x 16 + 16 x 16)	page 3-92	6, 7EM
`SMLAWy`	Signed Multiply with Accumulate (32 <= 32 x 16 + 32)	page 3-83	5E, 7EM
`SMLSD`	Dual Signed Multiply Subtract Accumulate (32 <= 32 + 16 x 16 – 16 x 16)	page 3-90	6, 7EM
`SMLSLD`	Dual Signed Multiply Subtract Accumulate Long (64 <= 64 + 16 x 16 – 16 x 16)	page 3-92	6, 7EM
`SMMLA`	Signed top word Multiply with Accumulate (32 <= TopWord(32 x 32 + 32))	page 3-88	6, 7EM
`SMMLS`	Signed top word Multiply with Subtract (32 <= TopWord(32 - 32 x 32))	page 3-88	6, 7EM
`SMMUL`	Signed top word Multiply (32 <= TopWord(32 x 32))	page 3-88	6, 7EM
`SMUAD`, `SMUSD`	Dual Signed Multiply, and Add or Subtract products	page 3-86	6, 7EM
`SMULxy`	Signed Multiply (32 <= 16 x 16)	page 3-81	5E, 7EM
`SMULL`	Signed Multiply (64 <= 32 x 32)	page 3-79	x6M
`SMULWy`	Signed Multiply (32 <= 32 x 16)	page 3-83	5E, 7EM
`SRS`	Store Return State	page 3-38	T2, x7M
`SSAT`	Signed Saturate	page 3-100	6, x6M
`SSAT16`	Signed Saturate, parallel halfwords	page 3-107	6, 7EM
`SSUB8`, `SSUB16`, `SSAX`	Parallel signed arithmetic	page 3-103	6, 7EM
`STC`	Store Coprocessor	page 3-132	x6M
`STC2`	Store Coprocessor	page 3-132	5, x6M
`STM`	Store Multiple registers	page 3-30	All
`STR`	Store Register with word	page 3-9	All

Table 3-1 Location of instructions (continued)

Mnemonic	Brief description	See	Arch. [a]
STRB	Store Register with byte	page 3-9	All
STRBT	Store Register with byte, user mode	page 3-9	x6M
STRD	Store Registers with two words	page 3-9	5E, x6M
STREX	Store Register Exclusive	page 3-40	6, x6M
STREXB, STREXH	Store Register Exclusive Byte, Halfword	page 3-40	K, x6M
STREXD	Store Register Exclusive Doubleword	page 3-40	K, x7M
STRH	Store Register with halfword	page 3-9	All
STRHT	Store Register with halfword, user mode	page 3-9	T2
STRT	Store Register with word, user mode	page 3-9	x6M
SUB	Subtract	page 3-51	All
SUBS pc, lr	Exception return, no stack	page 3-55	T2, x7M
SVC (formerly SWI)	SuperVisor Call	page 3-136	All
SWP, SWPB	Swap registers and memory (ARM only)	page 3-44	All, x7M
SXTAB, SXTAB16, SXTAH	Signed extend, with Addition	page 3-112	6, 7EM
SXTB, SXTH	Signed extend	page 3-112	6
SXTB16	Signed extend	page 3-112	6, 7EM
SYS	Execute system coprocessor instruction	page 3-131	7A, 7R
TBB, TBH	Table Branch Byte, Halfword	page 3-124	T2
TEQ	Test Equivalence	page 3-66	x6M
TST	Test	page 3-66	All
UADD8, UADD16, UASX	Parallel Unsigned Arithmetic	page 3-103	6, 7EM
UDIV	Unsigned divide	page 3-75	7M, 7R
UHADD8, UHADD16, UHASX, UHSUB8, UHSUB16, UHSAX	Parallel Unsigned Halving Arithmetic	page 3-103	6, 7EM
UMAAL	Unsigned Multiply Accumulate Accumulate Long (64 <= 32 + 32 + 32 x 32)	page 3-94	6, 7EM
UMLAL, UMULL	Unsigned Multiply Accumulate, Unsigned Multiply (64 <= 32 x 32 + 64), (64 <= 32 x 32)	page 3-79	x6M
UQADD8, UQADD16, UQASX, UQSUB8, UQSUB16, UQSAX	Parallel Unsigned Saturating Arithmetic	page 3-103	6, 7EM
USAD8	Unsigned Sum of Absolute Differences	page 3-105	6, 7EM
USADA8	Accumulate Unsigned Sum of Absolute Differences	page 3-105	6, 7EM
USAT	Unsigned Saturate	page 3-100	6, x6M

Table 3-1 Location of instructions (continued)

Mnemonic	Brief description	See	Arch. [a]
`USAT16`	Unsigned Saturate, parallel halfwords	page 3-107	6, 7EM
`USUB8`, `USUB16`, `USAX`	Parallel unsigned arithmetic	page 3-103	6, 7EM
`UXTAB, UXTAB16, UXTAH`	Unsigned extend with Addition	page 3-112	6, 7EM
`UXTB, UXTH`	Unsigned extend	page 3-112	6
`UXTB16`	Unsigned extend	page 3-112	6, 7EM
`V*`	See Chapter 4 *NEON and VFP Programming*		
`WFE`, `WFI`, `YIELD`	Wait For Event, Wait For Interrupt, Yield	page 3-145	T2, 6M

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

All	These instructions are available in all versions of the ARM architecture.
5	These instructions are available in the ARMv5T*, ARMv6*, and ARMv7 architectures.
5E	These instructions are available in the ARMv5TE, ARMv6*, and ARMv7 architectures.
6	These instructions are available in the ARMv6* and ARMv7 architectures.
6M	These instructions are available in the ARMv6-M and ARMv7 architectures.
x6M	These instructions are not available in the ARMv6-M architecture.
7	These instructions are available in the ARMv7 architectures.
7M	These instructions are available in the ARMv7-M architecture, including ARMv7E-M implementations.
x7M	These instructions are not available in the ARMv6-M or ARMv7-M architecture, or any ARMv7E-M implementation.
7EM	These instructions are available in ARMv7E-M implementations but not in the ARMv7-M or ARMv6-M architecture.
7R	These instructions are available in the ARMv7-R architecture.
7MP	These instructions are available in the ARMv7 architectures that implement the Multiprocessing Extensions.
7VE	These instructions are available in the ARMv7 architectures that implement the Virtualization Extensions.
EE	These instructions are available in ThumbEE variants of the ARM architecture.
J	This instruction is available in the ARMv5TEJ, ARMv6*, and ARMv7 architectures.
K	These instructions are available in the ARMv6K, and ARMv7 architectures.
T	These instructions are available in ARMv4T, ARMv5T*, ARMv6*, and ARMv7 architectures.
T2	These instructions are available in the ARMv6T2 and above architectures.
XScale	These instructions are available in XScale versions of the ARM architecture.
Z	This instruction is available if Security Extensions are implemented.

3.2 Instruction width specifiers

The instruction width specifiers `.W` and `.N` control the instruction size of Thumb code assembled for ARMv6T2 or later.

In Thumb code (ARMv6T2 or later) the `.W` width specifier forces the assembler to generate a 32-bit encoding, even if a 16-bit encoding is available. The `.W` specifier has no effect when assembling to ARM code.

In Thumb code the `.N` width specifier forces the assembler to generate a 16-bit encoding. In this case, if the instruction cannot be encoded in 16 bits or if `.N` is used in ARM code, the assembler generates an error.

If you use an instruction width specifier, you must place it immediately after the instruction mnemonic and any condition code, for example:

```
BCS.W   label   ; forces 32-bit instruction even for a short branch
B.N     label   ; faults if label out of range for 16-bit instruction
```

3.3 Memory access instructions

This section contains the following subsections:

Note

There is also an `LDR` pseudo-instruction. This pseudo-instruction either assembles to an `LDR` instruction, or to a `MOV` or `MVN` instruction.

See also

Concepts

Using the Assembler:

- *Memory accesses* on page 5-27.

Reference

- *LDR pseudo-instruction* on page 3-159.

3.3.1 LDR and STR (immediate offset)

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

Syntax

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

where:

op can be either:

LDR	Load Register
STR	Store Register.

type can be any one of:

B	unsigned Byte (Zero extend to 32 bits on loads.)
SB	signed Byte (LDR only. Sign extend to 32 bits.)
H	unsigned Halfword (Zero extend to 32 bits on loads.)
SH	signed Halfword (LDR only. Sign extend to 32 bits.)
-	omitted, for Word.

cond is an optional condition code.

Rt is the register to load or store.

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

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

Rt2 is the additional register to load or store for doubleword operations.

Not all options are available in every instruction set and architecture.

Offset ranges and architectures

Table 3-2 shows the ranges of offsets and availability of these instructions.

Table 3-2 Offsets and architectures, LDR/STR, word, halfword, and byte

Instruction	Immediate offset	Pre-indexed	Post-indexed	Arch.
ARM, word or byte [a]	–4095 to 4095	–4095 to 4095	–4095 to 4095	All
ARM, signed byte, halfword, or signed halfword	–255 to 255	–255 to 255	–255 to 255	All
ARM, doubleword	–255 to 255	–255 to 255	–255 to 255	v5TE +
32-bit Thumb, word, halfword, signed halfword, byte, or signed byte [a]	–255 to 4095	–255 to 255	–255 to 255	v6T2, v7
32-bit Thumb, doubleword	–1020 to 1020 [c]	–1020 to 1020 [c]	–1020 to 1020 [c]	v6T2, v7
16-bit Thumb, word [b]	0 to 124 [c]	Not available	Not available	All T
16-bit Thumb, unsigned halfword [b]	0 to 62 [d]	Not available	Not available	All T
16-bit Thumb, unsigned byte [b]	0 to 31	Not available	Not available	All T
16-bit Thumb, word, `Rn` is SP [e]	0 to 1020 [c]	Not available	Not available	All T
16-bit ThumbEE, word [b]	–28 to 124 [c]	Not available	Not available	T-2EE
16-bit ThumbEE, word, `Rn` is `R9` [e]	0 to 252 [c]	Not available	Not available	T-2EE
16-bit ThumbEE, word, `Rn` is `R10` [e]	0 to 124 [c]	Not available	Not available	T-2EE

a. For word loads, Rt can be the PC. A load to the PC causes a branch to the address loaded. In ARMv4, bits[1:0] of the address loaded must be 0b00. In ARMv5T and above, bits[1:0] must not be 0b10, and if bit[0] is 1, execution continues in Thumb state, otherwise execution continues in ARM state.

b. Rt and Rn must be in the range R0-R7.

c. Must be divisible by 4.

d. Must be divisible by 2.

e. Rt must be in the range R0-R7.

Register restrictions

R*n* must be different from R*t* in the pre-index and post-index forms.

Doubleword register restrictions

R*n* must be different from R*t2* in the pre-index and post-index forms.

For Thumb instructions, you must not specify SP or PC for either *Rt* or *Rt2*.

For ARM instructions:

- *Rt* must be an even-numbered register
- *Rt* must not be LR
- it is strongly recommended that you do not use R12 for *Rt*
- *Rt2* must be R(*t* + 1).

Use of PC

In ARM instructions:

- You can use PC for `Rt` in `LDR` word instructions and PC for `Rn` in `LDR` instructions.
- You can use PC for `Rt` in `STR` word instructions and PC for `Rn` in `STR` instructions with immediate offset syntax (that is the forms that do not writeback to the `Rn`). However, these are deprecated in ARMv6T2 and above.

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

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

Use of SP

You can use SP for `Rn`.

In ARM, you can use SP for `Rt` in word instructions. You can use SP for `Rt` in non-word instructions in ARM code but this is deprecated in ARMv6T2 and above.

In Thumb, you can use SP for `Rt` in word instructions only. All other use of SP for `Rt` in these instructions are not permitted in Thumb code.

Examples

```
    LDR     r8,[r10]            ; loads R8 from the address in R10.
    LDRNE   r2,[r5,#960]!       ; (conditionally) loads R2 from a word
                                ; 960 bytes above the address in R5, and
                                ; increments R5 by 960.
    STR     r2,[r9,#consta-struc]   ; consta-struc is an expression evaluating
                                    ; to a constant in the range 0-4095.
```

See also

Reference

- *Memory access instructions* on page 3-9
- *Condition codes* on page 3-163.

3.3.2 LDR and STR (register offset)

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

Syntax

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

where:

op can be either:

LDR	Load Register
STR	Store Register.

type can be any one of:

B	unsigned Byte (Zero extend to 32 bits on loads.)
SB	signed Byte (LDR only. Sign extend to 32 bits.)
H	unsigned Halfword (Zero extend to 32 bits on loads.)
SH	signed Halfword (LDR only. Sign extend to 32 bits.)
-	omitted, for Word.

cond is an optional condition code.

Rt is the register to load or store.

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

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

shift is an optional shift.

Rt2 is the additional register to load or store for doubleword operations.

Not all options are available in every instruction set and architecture.

Offset register and shift options

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

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

Instruction	+/–*Rm* [a]	shift			Arch.
ARM, word or byte [b]	+/–*Rm*	LSL #0-31	LSR #1-32		All
		ASR #1-32	ROR #1-31	RRX	
ARM, signed byte, halfword, or signed halfword	+/–*Rm*	Not available			All
ARM, doubleword	+/–*Rm*	Not available			v5TE +

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

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

a. Where +/–Rm is shown, you can use –Rm, +Rm, or Rm. Where +Rm is shown, you cannot use –Rm.

b. For word loads, Rt can be the PC. A load to the PC causes a branch to the address loaded. In ARMv4, bits[1:0] of the address loaded must be 0b00. In ARMv5T and above, bits[1:0] must not be 0b10, and if bit[0] is 1, execution continues in Thumb state, otherwise execution continues in ARM state.

c. Rt, Rn, and Rm must all be in the range R0-R7.

Register restrictions

In the pre-index and post-index forms:

- R*n* must be different from R*t*
- R*n* must be different from R*m* in architectures before ARMv6.

Doubleword register restrictions

For ARM instructions:

- *Rt* must be an even-numbered register
- *Rt* must not be LR
- it is strongly recommended that you do not use R12 for *Rt*
- *Rt2* must be R(*t* + 1)
- R*m* must be different from R*t* and R*t2* in LDRD instructions
- R*n* must be different from R*t2* in the pre-index and post-index forms.

Use of PC

In ARM instructions:

- You can use PC for R*t* in LDR word instructions, and you can use PC for R*n* in LDR instructions with register offset syntax (that is the forms that do not writeback to the R*n*).
- You can use PC for R*t* in STR word instructions, and you can use PC for R*n* in STR instructions with register offset syntax (that is the forms that do not writeback to the R*n*). However, these are deprecated in ARMv6T2 and above.

Other uses of PC are not permitted in ARM instructions.

In Thumb instructions you can use PC for R*t* in LDR word instructions. Other uses of PC in these Thumb instructions are not permitted.

Use of SP

You can use SP for R*n*.

In ARM, you can use SP for `Rt` in word instructions. You can use SP for `Rt` in non-word ARM instructions but this is deprecated in ARMv6T2 and above.

You can use SP for `Rm` in ARM instructions but this is deprecated in ARMv6T2 and above.

In Thumb, you can use SP for `Rt` in word instructions only. All other use of SP for `Rt` in these instructions are not permitted in Thumb code.

Use of SP for `Rm` is not permitted in Thumb state.

See also

Reference

- *Memory access instructions* on page 3-9
- *Condition codes* on page 3-163.

3.3.3 LDR and STR, unprivileged

Unprivileged load and Store, byte, halfword, or word.

When these instructions are executed by privileged software, they access memory with the same restrictions as they would have if they were executed by unprivileged software.

When executed by unprivileged software these instructions behave in exactly the same way as the corresponding load or store instruction, for example `LDRSBT` behaves in the same way as `LDRSB`.

Syntax

```
op{type}T{cond} Rt, [Rn {, #offset}]       ; immediate offset (32-bit Thumb only)
op{type}T{cond} Rt, [Rn] {, #offset}       ; post-indexed (ARM only)
op{type}T{cond} Rt, [Rn], +/-Rm {, shift}  ; post-indexed (register) (ARM only)
```

where:

op can be either:

`LDR`	Load Register
`STR`	Store Register.

type can be any one of:

`B`	unsigned Byte (Zero extend to 32 bits on loads.)
`SB`	signed Byte (`LDR` only. Sign extend to 32 bits.)
`H`	unsigned Halfword (Zero extend to 32 bits on loads.)
`SH`	signed Halfword (`LDR` only. Sign extend to 32 bits.)
`-`	omitted, for Word.

cond is an optional condition code.

Rt is the register to load or store.

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

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

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

shift is an optional shift.

Offset ranges and architectures

Table 3-2 on page 3-12 shows the ranges of offsets and availability of these instructions.

Table 3-4 Offsets and architectures, LDR/STR (User mode)

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

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

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

a. You can use –Rm, +Rm, or Rm.

See also

Reference

- *Memory access instructions* on page 3-9
- *Condition codes* on page 3-163.

3.3.4 LDR (PC-relative)

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

Syntax

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

LDRD{cond} Rt, Rt2, label       ; Doubleword
```

where:

type	can be any one of:	
	`B`	unsigned Byte (Zero extend to 32 bits on loads.)
	`SB`	signed Byte (`LDR` only. Sign extend to 32 bits.)
	`H`	unsigned Halfword (Zero extend to 32 bits on loads.)
	`SH`	signed Halfword (`LDR` only. Sign extend to 32 bits.)
	`-`	omitted, for Word.
cond	is an optional condition code.	
`.W`	is an optional instruction width specifier.	
Rt	is the register to load or store.	
Rt2	is the second register to load or store.	
label	is a PC-relative expression. *label* must be within a limited distance of the current instruction.	

Note

Equivalent syntaxes are available for the `STR` instruction in ARM code but they are deprecated in ARMv6T2 and above.

Offset range and architectures

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

Table 3-5 shows the possible offsets between label and the current instruction.

Table 3-5 PC-relative offsets

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

a. For word loads, Rt can be the PC. A load to the PC causes a branch to the address loaded. In ARMv4, bits[1:0] of the address loaded must be 0b00. In ARMv5T and above, bits[1:0] must not be 0b10, and if bit[0] is 1, execution continues in Thumb state, otherwise execution continues in ARM state.

b. Must be a multiple of 4.

c. Rt must be in the range R0-R7. There are no byte, halfword, or doubleword 16-bit instructions.

Note

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

LDR (PC-relative) in 32-bit Thumb

You can use the `.W` width specifier to force `LDR` to generate a 32-bit instruction in Thumb code. `LDR.W` always generates a 32-bit instruction, even if the target could be reached using a 16-bit `LDR`.

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

Doubleword register restrictions

For 32-bit Thumb instructions, you must not specify SP or PC for either *`Rt`* or *`Rt2`*.

For ARM instructions:

- *`Rt`* must be an even-numbered register
- *`Rt`* must not be LR
- it is strongly recommended that you do not use `R12` for *`Rt`*
- *`Rt2`* must be `R(`*`t`*` + 1).`

Use of SP

In ARM, you can use SP for `Rt` in `LDR` word instructions. You can use SP for `Rt` in `LDR` non-word ARM instructions but this is deprecated in ARMv6T2 and above.

In Thumb, you can use SP for `Rt` in `LDR` word instructions only. All other uses of SP in these instructions are not permitted in Thumb code.

See also

Concepts

Using the Assembler:

- *Register-relative and PC-relative expressions* on page 8-7.

Reference

- *Pseudo-instructions* on page 3-155
- *LDR (PC-relative) in 32-bit Thumb*
- *Memory access instructions* on page 3-9
- *Condition codes* on page 3-163.

3.3.5 LDR (register-relative)

Load register. The address is an offset from a base register.

Syntax

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

```
LDRD{cond} Rt, Rt2, label        ; Doubleword
```

where:

type	can be any one of:
	`B` unsigned Byte (Zero extend to 32 bits on loads.)
	`SB` signed Byte (`LDR` only. Sign extend to 32 bits.)
	`H` unsigned Halfword (Zero extend to 32 bits on loads.)
	`SH` signed Halfword (`LDR` only. Sign extend to 32 bits.)
	`-` omitted, for Word.
cond	is an optional condition code.
`.W`	is an optional instruction width specifier.
Rt	is the register to load or store.
Rt2	is the second register to load or store.
label	is a symbol defined by the FIELD directive. *label* specifies an offset from the base register which is defined using the MAP directive.
	label must be within a limited distance of the value in the base register.

Offset range and architectures

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

Table 3-5 on page 3-19 shows the possible offsets between label and the current instruction.

Table 3-6 register-relative offsets

Instruction	Offset range	Architectures
ARM `LDR`, `LDRB`[a]	+/– 4095	All
ARM `LDRSB`, `LDRH`, `LDRSH`	+/– 255	All
ARM `LDRD`	+/– 255	v5TE +
32-bit Thumb `LDR`, `LDRB`, `LDRSB`, `LDRH`, `LDRSH` [a]	–255 to 4095	v6T2, v7
32-bit Thumb `LDRD`	+/– 1020 [b]	v6T2, v7
16-bit Thumb `LDR` [c]	0 to 124 [b]	All T
16-bit Thumb `LDRH` [c]	0 to 62 [d]	All T
16-bit Thumb `LDRB` [c]	0 to 31	All T
16-bit Thumb `LDR`, base register is SP[e]	0 to 1020 [b]	All T
16-bit ThumbEE `LDR` [c]	–28 to 124 [b]	T-2EE
16-bit Thumb `LDR`, base register is `R9` [e]	0 to 252 [b]	T-2EE
16-bit ThumbEE `LDR`, base register is `R10` [e]	0 to 124 [b]	T-2EE

a. For word loads, Rt can be the PC. A load to the PC causes a branch to the address loaded. In ARMv4, bits[1:0] of the address loaded must be 0b00. In ARMv5T and above, bits[1:0] must not be 0b10, and if bit[0] is 1, execution continues in Thumb state, otherwise execution continues in ARM state.

b. Must be a multiple of 4.

c. Rt and base register must be in the range R0-R7.

d. Must be a multiple of 2.

e. Rt must be in the range R0-R7.

LDR (register-relative) in 32-bit Thumb

You can use the `.W` width specifier to force `LDR` to generate a 32-bit instruction in Thumb code. `LDR.W` always generates a 32-bit instruction, even if the target could be reached using a 16-bit `LDR`.

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

Doubleword register restrictions

For 32-bit Thumb instructions, you must not specify SP or PC for either *`Rt`* or *`Rt2`*.

For ARM instructions:

- *`Rt`* must be an even-numbered register
- *`Rt`* must not be LR

- it is strongly recommended that you do not use `R12` for *`Rt`*
- *`Rt2`* must be `R(`*`t`*` + 1)`.

Use of PC

You can use PC for `Rt` in word instructions. Other uses of PC are not permitted in these instructions.

Use of SP

In ARM, you can use SP for `Rt` in word instructions. You can use SP for `Rt` in non-word ARM instructions but this is deprecated in ARMv6T2 and above.

In Thumb, you can use SP for `Rt` in word instructions only. All other use of SP for `Rt` in these instructions are not permitted in Thumb code.

See also

Concepts

Using the Assembler:

- *Register-relative and PC-relative expressions* on page 8-7.

Reference

- *Memory access instructions* on page 3-9
- *Pseudo-instructions* on page 3-155
- *LDR (register-relative) in 32-bit Thumb* on page 3-22
- *FIELD* on page 6-18
- *MAP* on page 6-17
- *Condition codes* on page 3-163.

3.3.6 ADR (PC-relative)

`ADR` generates a PC-relative address in the destination register, for a label in the current area.

Syntax

`ADR{`*`cond`*`}{.W}` *`Rd`*`,`*`label`*

where:

`cond`	is an optional condition code.
`.W`	is an optional instruction width specifier.
`Rd`	is the destination register to load.
`label`	is a PC-relative expression. *`label`* must be within a limited distance of the current instruction.

Usage

`ADR` produces position-independent code, because the assembler generates an instruction that adds or subtracts a value to the PC.

Use the `ADRL` pseudo-instruction to assemble a wider range of effective addresses.

`label` must evaluate to an address in the same assembler area as the `ADR` instruction.

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

Offset range and architectures

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

Table 3-5 on page 3-19 shows the possible offsets between label and the current instruction.

Table 3-7 PC-relative offsets

Instruction	Offset range	Architectures
ARM `ADR`	See *Operand2 as a constant* on page 3-46	All
32-bit Thumb `ADR`	+/– 4095	v6T2, v7
16-bit Thumb `ADR` [a]	0-1020 [b]	All T

a. Rd must be in the range R0-R7.

b. Must be a multiple of 4.

ADR in 32-bit Thumb

You can use the `.W` width specifier to force `ADR` to generate a 32-bit instruction in Thumb code. `ADR` with `.W` always generates a 32-bit instruction, even if the address can be generated in a 16-bit instruction.

For forward references, `ADR` without `.W` always generates a 16-bit instruction in Thumb code, even if that results in failure for an address that could be generated in a 32-bit Thumb `ADD` instruction.

Restrictions

In Thumb code, `Rd` cannot be `PC` or `SP`.

In ARM code, `Rd` can be `PC` or `SP` but use of `SP` is deprecated in ARMv6T2 and above.

See also

Concepts

Using the Assembler:

- *Register-relative and PC-relative expressions* on page 8-7.

Reference

- *Memory access instructions* on page 3-9
- *ADRL pseudo-instruction* on page 3-156
- *AREA* on page 6-61
- *Condition codes* on page 3-163.

3.3.7 ADR (register-relative)

`ADR` generates a register-relative address in the destination register, for a label defined in a storage map.

Syntax

`ADR{cond}{.W} Rd,label`

where:

cond	is an optional condition code.
`.W`	is an optional instruction width specifier.
Rd	is the destination register to load.
label	is a symbol defined by the FIELD directive. *label* specifies an offset from the base register which is defined using the MAP directive. *label* must be within a limited distance from the base register.

Usage

`ADR` generates code to easily access named fields inside a storage map.

Use the `ADRL` pseudo-instruction to assemble a wider range of effective addresses.

Restrictions

In Thumb code:
- *Rd* cannot be `PC`
- *Rd* can be `SP` only if the base register is `SP`.

Offset range and architectures

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

Table 3-5 on page 3-19 shows the possible offsets between label and the current instruction.

Table 3-8 register-relative offsets

Instruction	Offset range	Architectures
ARM `ADR`	See *Operand2 as a constant* on page 3-46	All
32-bit Thumb `ADR`	+/– 4095	v6T2, v7
16-bit Thumb `ADR`, base register is SP [a]	0-1020 [b]	All T

a. *Rd* must be in the range R0-R7 or SP. If *Rd* is SP, the offset range is –508 to 508 and must be a multiple of 4

b. Must be a multiple of 4.

ADR in 32-bit Thumb

You can use the `.W` width specifier to force `ADR` to generate a 32-bit instruction in Thumb code. `ADR` with `.W` always generates a 32-bit instruction, even if the address can be generated in a 16-bit instruction.

For forward references, `ADR` without `.W`, with base register SP, always generates a 16-bit instruction in Thumb code, even if that results in failure for an address that could be generated in a 32-bit Thumb `ADD` instruction.

See also

Concepts

Using the Assembler:

- *Register-relative and PC-relative expressions* on page 8-7.

Reference

- *Memory access instructions* on page 3-9
- *MAP* on page 6-17
- *FIELD* on page 6-18
- *ADRL pseudo-instruction* on page 3-156
- *Condition codes* on page 3-163.

3.3.8 PLD, PLDW, and PLI

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

Syntax

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

where:

type	can be one of:
	`D` Data address
	`DW` Data address with intention to write
	`I` Instruction address.
	type cannot be `DW` if the syntax specifies *label*.
cond	is an optional condition code.
	Note: *cond* is permitted only in 32-bit Thumb code, using a preceding `IT` instruction. This is an unconditional instruction in ARM and you must not use *cond*.
Rn	is the register on which the memory address is based.
offset	is an immediate offset. If offset is omitted, the address is the value in *Rn*.
Rm	is a register containing a value to be used as the offset.
shift	is an optional shift.
label	is a PC-relative expression.

Range of offset

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

- –4095 to +4095 for ARM instructions
- –255 to +4095 for 32-bit Thumb instructions, when *Rn* is not PC.
- –4095 to +4095 for 32-bit Thumb instructions, when *Rn* is PC.

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

Register or shifted register offset

In ARM, the value in *Rm* is added to or subtracted from the value in *Rn*. In 32-bit Thumb, the value in *Rm* can only be added to the value in *Rn*. The result used as the memory address for the preload.

The range of shifts permitted is:

- `LSL` #0 to #3 for 32-bit Thumb instructions

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

Address alignment for preloads

No alignment checking is performed for preload instructions.

Register restrictions

`Rm` must not be PC. For Thumb instructions *`Rm`* must also not be SP.

`Rn` must not be PC for Thumb instructions of the syntax `PL`*`type`*`{`*`cond`*`} [R`*`n`*`, +/-R`*`m`*`{, #`*`shift`*`}]`.

Architectures

ARM `PLD` is available in ARMv5TE and above.

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

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

`PLI` is available only in ARMv7 and above.

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

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

See also

Concepts

Using the Assembler:

- *Register-relative and PC-relative expressions* on page 8-7.

Reference

- *Condition codes* on page 3-163.

3.3.9 LDM and STM

Load and Store Multiple registers. Any combination of registers `R0` to `R15` (PC) can be transferred in ARM state, but there are some restrictions in Thumb state.

Syntax

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

where:

op can be either:

`LDM`	Load Multiple registers
`STM`	Store Multiple registers.

addr_mode is any one of the following:

`IA`	Increment address After each transfer. This is the default, and can be omitted.
`IB`	Increment address Before each transfer (ARM only).
`DA`	Decrement address After each transfer (ARM only).
`DB`	Decrement address Before each transfer.

You can also use the stack oriented addressing mode suffixes, for example, when implementing stacks.

cond is an optional condition code.

Rn is the *base register*, the ARM register holding the initial address for the transfer. *Rn* must not be PC.

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

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

^ is an optional suffix, available in ARM state only. You must not use it in User mode or System mode. It has the following purposes:

- If the instruction is `LDM` (with any addressing mode) and *reglist* contains the PC (`R15`), in addition to the normal multiple register transfer, the `SPSR` is copied into the `CPSR`. This is for returning from exception handlers. Use this only from exception modes.
- Otherwise, data is transferred into or out of the User mode registers instead of the current mode registers.

Restrictions on reglist in 32-bit Thumb instructions

In 32-bit Thumb instructions:

- the SP cannot be in the list
- the PC cannot be in the list in an `STM` instruction
- the PC and LR cannot both be in the list in an `LDM` instruction
- there must be two or more registers in the list.

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

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

Restrictions on reglist in ARM instructions

ARM store instructions can have SP and PC in the *reglist* but these instructions that include SP or PC in the *reglist* are deprecated in ARMv6T2 and above.

ARM load instructions can have SP and PC in the *reglist* but these instructions that include SP in the *reglist* or both PC and LR in the *reglist* are deprecated in ARMv6T2 and above.

16-bit instructions

16-bit versions of a subset of these instructions are available in Thumb code.

The following restrictions apply to the 16-bit instructions:

- all registers in *`reglist`* must be Lo registers
- *`Rn`* must be a Lo register
- *`addr_mode`* must be omitted (or `IA`), meaning increment address after each transfer
- writeback must be specified for `STM` instructions
- writeback must be specified for `LDM` instructions where *`Rn`* is not in the *`reglist`*.

Note

16-bit Thumb `STM` instructions with writeback that specify *`Rn`* as the lowest register in the *`reglist`* are deprecated in ARMv6T2 and above.

In addition, the `PUSH` and `POP` instructions are subsets of the `STM` and `LDM` instructions and can therefore be expressed using the `STM` and `LDM` instructions. Some forms of `PUSH` and `POP` are also 16-bit instructions.

Note

These 16-bit instructions are not available in ThumbEE.

Loading to the PC

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

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

In ARMv5T and above:

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

Loading or storing the base register, with writeback

In ARM or 16-bit Thumb instructions, if *`Rn`* is in *`reglist`*, and writeback is specified with the `!` suffix:

- If the instruction is `STM{`*`addr_mode`*`}{`*`cond`*`}` and *`Rn`* is the lowest-numbered register in *`reglist`*, the initial value of *`Rn`* is stored. These instructions are deprecated in ARMv6T2 and above.

- Otherwise, the loaded or stored value of `Rn` cannot be relied upon, so these instructions are not permitted.

32-bit Thumb instructions are not permitted if `Rn` is in `reglist`, and writeback is specified with the `!` suffix.

Examples

```
    LDM     r8,{r0,r2,r9}      ; LDMIA is a synonym for LDM
    STMDB   r1!,{r3-r6,r11,r12}
```

Incorrect examples

```
    STM     r5!,{r5,r4,r9} ; value stored for R5 UNKNOWN
    LDMDA   r2, {}         ; must be at least one register in list
```

See also

Concepts

Using the Assembler:

- *Stack implementation using LDM and STM* on page 5-22.

Reference

- *Memory access instructions* on page 3-9
- *PUSH and POP* on page 3-33
- *Condition codes* on page 3-163.

3.3.10 PUSH and POP

Push registers onto, and pop registers off a full descending stack.

Syntax

`PUSH{`*`cond`*`}` *`reglist`*

`POP{`*`cond`*`}` *`reglist`*

where:

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

Usage

`PUSH` is a synonym for `STMDB sp!, reglist` and `POP` is a synonym for `LDMIA sp! reglist`. `PUSH` and `POP` are the preferred mnemonics in these cases.

Note

`LDM` and `LDMFD` are synonyms of `LDMIA`. `STMFD` is a synonym of `STMDB`.

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

POP, with reglist including the PC

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

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

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

Thumb instructions

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

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

The following restrictions apply to the 32-bit instructions:
- *`reglist`* must not include the SP
- For `PUSH`, *`reglist`* must not include the PC
- For `POP`, *`reglist`* can include either the LR or the PC, but not both.

Restrictions on reglist in ARM instructions

ARM PUSH instructions can have SP and PC in the *reglist* but these instructions that include SP or PC in the *reglist* are deprecated in ARMv6T2 and above.

ARM POP instructions cannot have SP but can have PC in the *reglist*. These instructions that include both PC and LR in the *reglist* are deprecated in ARMv6T2 and above.

Examples

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

See also

Reference

- *Memory access instructions* on page 3-9
- *LDM and STM* on page 3-30
- *Condition codes* on page 3-163.

3.3.11 RFE

Return From Exception.

Syntax

RFE{*addr_mode*}{*cond*} *Rn*{!}

where:

addr_mode is any one of the following:

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

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

cond is an optional condition code.

Note

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

Rn specifies the base register. *Rn* must not be PC.

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

Usage

You can use RFE to return from an exception if you previously saved the return state using the SRS instruction. *Rn* is usually the SP where the return state information was saved.

Operation

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

Notes

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

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

The results of breaking these rules are UNPREDICTABLE. However, no special precautions are required in software, if the instructions are used to return after a valid exception entry mechanism.

Where addresses are not word-aligned, RFE ignores the least significant two bits of *Rn*.

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

Do not use `RFE` in unprivileged software execution.

Do not use `RFE` in ThumbEE.

Architectures

This ARM instruction is available in ARMv6 and above.

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

There is no 16-bit version of this instruction.

Example

```
    RFE sp!
```

See also

Concepts

Using the Assembler:

- *Processor modes, and privileged and unprivileged software execution* on page 3-5.

Reference

- *SRS* on page 3-38
- *Condition codes* on page 3-163.

3.3.12 ERET

Exception Return.

Syntax

`ERET{cond}`

where:

cond	is an optional condition code.

Usage

In a processor that implements the Virtualization Extensions, you can use `ERET` to perform a return from an exception taken to Hyp mode.

Operation

When executed in Hyp mode, `ERET` loads the PC from ELR_hyp and loads the CPSR from SPSR_hyp. When executed in any other mode, apart from User or System, it behaves as:

- `MOVS PC, LR` in the ARM instruction set
- `SUBS PC, LR, #0` in the Thumb instruction set.

Notes

`ERET` is UNPREDICTABLE if it is executed in ThumbEE state or in User or System mode.

`ERET` is the preferred synonym for `SUBS PC, LR, #0` in the Thumb instruction set.

Architectures

This ARM instruction is available in ARMv7 architectures that include the Virtualization Extensions.

This 32-bit Thumb instruction is available in ARMv7 architectures that include the Virtualization Extensions.

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

See also

Concepts

Using the Assembler:

- *Processor modes, and privileged and unprivileged software execution* on page 3-5.

Reference

- *MOV and MVN* on page 3-62
- *SUBS pc, lr* on page 3-55
- *Condition codes* on page 3-163.

3.3.13 SRS

Store Return State onto a stack.

Syntax

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

SRS{addr_mode}{cond} #modenum{!}        ; This is a pre-UAL syntax
```

where:

addr_mode is any one of the following:

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

If *addr_mode* is omitted, it defaults to Increment After. You can also use stack oriented addressing mode suffixes, for example, when implementing stacks.

cond is an optional condition code.

Note

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

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

modenum specifies the number of the mode whose banked SP is used as the base register. You must use only the defined mode numbers.

Operation

SRS stores the LR and the SPSR of the current mode, at the address contained in SP of the mode specified by *modenum*, and the following word respectively. Optionally updates SP of the mode specified by *modenum*. This is compatible with the normal use of the STM instruction for stack accesses.

Note

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

Usage

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

Notes

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

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

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

Do not use SRS in ThumbEE.

SRS is not permitted in a non-secure state if *modenum* specifies monitor mode.

Architectures

This ARM instruction is available in ARMv6 and above.

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

There is no 16-bit version of this instruction.

Example

```
R13_usr  EQU     16
         SRSFD   sp,#R13_usr
```

See also

Concepts

Using the Assembler:

- *Stack implementation using LDM and STM* on page 5-22
- *Processor modes, and privileged and unprivileged software execution* on page 3-5.

Reference

- *LDM and STM* on page 3-30
- *Condition codes* on page 3-163.

3.3.14 LDREX and STREX

Load and Store Register Exclusive.

Syntax

```
LDREX{cond} Rt, [Rn {, #offset}]

STREX{cond} Rd, Rt, [Rn {, #offset}]

LDREXB{cond} Rt, [Rn]

STREXB{cond} Rd, Rt, [Rn]

LDREXH{cond} Rt, [Rn]

STREXH{cond} Rd, Rt, [Rn]

LDREXD{cond} Rt, Rt2, [Rn]

STREXD{cond} Rd, Rt, Rt2, [Rn]
```

where:

cond	is an optional condition code.
Rd	is the destination register for the returned status.
Rt	is the register to load or store.
Rt2	is the second register for doubleword loads or stores.
Rn	is the register on which the memory address is based.
offset	is an optional offset applied to the value in *Rn*. *offset* is permitted only in 32-bit Thumb instructions. If *offset* is omitted, an offset of 0 is assumed.

LDREX

`LDREX` loads data from memory.

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

STREX

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

- If the physical address does not have the Shared TLB attribute, and the executing processor has an outstanding tagged physical address, the store takes place, the tag is cleared, and the value 0 is returned in *Rd*.
- If the physical address does not have the Shared TLB attribute, and the executing processor does not have an outstanding tagged physical address, the store does not take place, and the value 1 is returned in *Rd*.
- If the physical address has the Shared TLB attribute, and the physical address is tagged as exclusive access for the executing processor, the store takes place, the tag is cleared, and the value 0 is returned in *Rd*.

- If the physical address has the Shared TLB attribute, and the physical address is not tagged as exclusive access for the executing processor, the store does not take place, and the value 1 is returned in *Rd*.

Restrictions

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

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

For ARM instructions:
- SP can be used but use of SP for any of *Rd*, *Rt*, or *Rt2* is deprecated in ARMv6T2 and above
- For LDREXD and STREXD, *Rt* must be an even numbered register, and not LR
- *Rt2* must be *R(t+1)*
- *offset* is not permitted.

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

Usage

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

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

Note

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

Architectures

ARM LDREX and STREX are available in ARMv6 and above.

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

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

There are no 16-bit versions of these instructions.

Examples

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

See also

Reference

- *Memory access instructions* on page 3-9
- *Condition codes* on page 3-163.

3.3.15 CLREX

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

Syntax

`CLREX{cond}`

where:

cond is an optional condition code.

Note

cond is permitted only in Thumb code, using a preceding `IT` instruction. This is an unconditional instruction in ARM.

Usage

Use the `CLREX` instruction to return a closely-coupled exclusive access monitor to its open-access state. This removes the requirement for a dummy store to memory.

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

Architectures

This ARM instruction is available in ARMv6K and above.

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

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

See also

Reference

- *Memory access instructions* on page 3-9
- *Condition codes* on page 3-163
- *ARM Architecture Reference Manual*, `http://infocenter.arm.com/help/topic/com.arm.doc.subset.architecture.reference/index.html`.

3.3.16 SWP and SWPB

Swap data between registers and memory.

Syntax

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

where:

cond	is an optional condition code.
`B`	is an optional suffix. If `B` is present, a byte is swapped. Otherwise, a 32-bit word is swapped.
Rt	is the destination register. *Rt* must not be PC.
Rt2	is the source register. *Rt2* can be the same register as *Rt*. *Rt2* must not be PC.
Rn	contains the address in memory. *Rn* must be a different register from both *Rt* and *Rt2*. *Rn* must not be PC.

Usage

You can use `SWP` and `SWPB` to implement semaphores:

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

Note

The use of `SWP` and `SWPB` is deprecated in ARMv6 and above. You can use `LDREX` and `STREX` instructions to implement more sophisticated semaphores in ARMv6 and above.

Architectures

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

There are no Thumb `SWP` or `SWPB` instructions.

See also

Reference

- *Memory access instructions* on page 3-9
- *LDREX and STREX* on page 3-40
- *Condition codes* on page 3-163.

3.4 General data processing instructions

This section contains the following subsections:

3.4.1 Flexible second operand (Operand2)

Many ARM and Thumb general data processing instructions have a flexible second operand. This is shown as *`Operand2`* in the descriptions of the syntax of each instruction.

`Operand2` can be a:

- constant
- register with optional shift.

3.4.2 Operand2 as a constant

You specify an Operand2 constant in the form:

`#constant`

where *`constant`* is an expression evaluating to a numeric value.

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

In Thumb instructions, *`constant`* can be:

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

——— **Note** ———

In the constants shown above, `X` and `Y` are hexadecimal digits.

———

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

When an Operand2 constant is used with the instructions `MOVS`, `MVNS`, `ANDS`, `ORRS`, `ORNS`, `EORS`, `BICS`, `TEQ` or `TST`, the carry flag is updated to bit[31] of the constant, if the constant is greater than 255 and can be produced by shifting an 8-bit value. These instructions do not affect the carry flag if Operand2 is any other constant.

Instruction substitution

If a value of *`constant`* is not available, but its logical inverse or negation is available, then the assembler produces an equivalent instruction and inverts or negates *`constant`*.

For example, an assembler might assemble the instruction `CMP` *`Rd`*`, #0xFFFFFFFE` as the equivalent instruction `CMN` *`Rd`*`, #0x2`.

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

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

3.4.3 Operand2 as a register with optional shift

You specify an Operand2 register in the form:

`Rm` `{,` *`shift`*`}`

where:

`Rm`	is the register holding the data for the second operand.
`shift`	is an optional constant or register-controlled shift to be applied to *`Rm`*. It can be one of:

`ASR #`*`n`*	arithmetic shift right *n* bits, $1 \leq n \leq 32$.
`LSL #`*`n`*	logical shift left *n* bits, $1 \leq n \leq 31$.
`LSR #`*`n`*	logical shift right *n* bits, $1 \leq n \leq 32$.
`ROR #`*`n`*	rotate right *n* bits, $1 \leq n \leq 31$.
`RRX`	rotate right one bit, with extend.
`type Rs`	register-controlled shift is available in ARM code only, where:
	`type` is one of `ASR`, `LSL`, `LSR`, `ROR`.
	`Rs` is a register supplying the shift amount, and only the least significant byte is used.
-	if omitted, no shift occurs, equivalent to `LSL #0`.

If you omit the shift, or specify `LSL #0`, the instruction uses the value in *`Rm`*.

If you specify a shift, the shift is applied to the value in *`Rm`*, and the resulting 32-bit value is used by the instruction. However, the contents in the register *`Rm`* remains unchanged. Specifying a register with shift also updates the carry flag when used with certain instructions.

3.4.4 Shift operations

Register shift operations move the bits in a register left or right by a specified number of bits, the *shift length*. Register shift can be performed:

- directly by the instructions `ASR`, `LSR`, `LSL`, `ROR`, and `RRX`, and the result is written to a destination register
- during the calculation of *`Operand2`* by the instructions that specify the second operand as a register with shift. The result is used by the instruction.

The permitted shift lengths depend on the shift type and the instruction, see the individual instruction description or the flexible second operand description. If the shift length is 0, no shift occurs. Register shift operations update the carry flag except when the specified shift length is 0. The following sub-sections describe the various shift operations and how they affect the carry flag. In these descriptions, *`Rm`* is the register containing the value to be shifted, and *n* is the shift length.

ASR

Arithmetic shift right by *n* bits moves the left-hand `32-`*`n`* bits of the register *`Rm`*, to the right by *n* places, into the right-hand `32-`*`n`* bits of the result. And it copies the original bit[31] of the register into the left-hand *n* bits of the result. See Figure 3-1 on page 3-48.

You can use the `ASR #`*`n`* operation to divide the value in the register *`Rm`* by 2^n, with the result being rounded towards negative-infinity.

When the instruction is `ASRS` or when `ASR #n` is used in *Operand2* with the instructions `MOVS`, `MVNS`, `ANDS`, `ORRS`, `ORNS`, `EORS`, `BICS`, `TEQ` or `TST`, the carry flag is updated to the last bit shifted out, bit[*n*-1], of the register *Rm*.

Note

- If *n* is 32 or more, then all the bits in the result are set to the value of bit[31] of *Rm*.
- If *n* is 32 or more and the carry flag is updated, it is updated to the value of bit[31] of *Rm*.

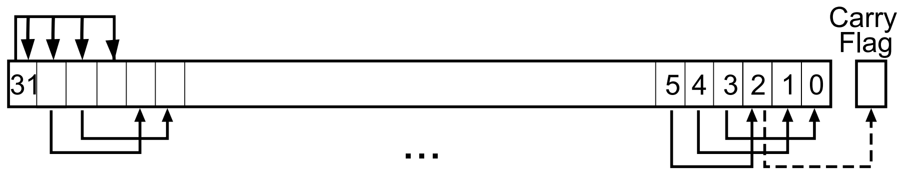

Figure 3-1 ASR #3

LSR

Logical shift right by *n* bits moves the left-hand `32-n` bits of the register *Rm*, to the right by *n* places, into the right-hand `32-n` bits of the result. And it sets the left-hand *n* bits of the result to 0. See Figure 3-2.

You can use the `LSR #n` operation to divide the value in the register *Rm* by 2^n, if the value is regarded as an unsigned integer.

When the instruction is `LSRS` or when `LSR #n` is used in *Operand2* with the instructions `MOVS`, `MVNS`, `ANDS`, `ORRS`, `ORNS`, `EORS`, `BICS`, `TEQ` or `TST`, the carry flag is updated to the last bit shifted out, bit[*n*-1], of the register *Rm*.

Note

- If *n* is 32 or more, then all the bits in the result are cleared to 0.
- If *n* is 33 or more and the carry flag is updated, it is updated to 0.

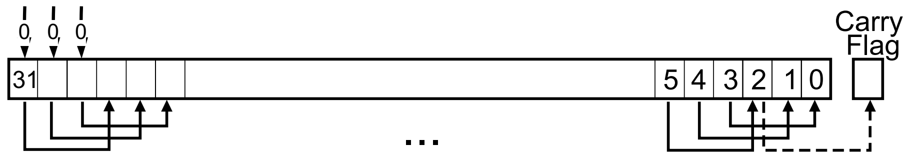

Figure 3-2 LSR #3

LSL

Logical shift left by *n* bits moves the right-hand `32-n` bits of the register *Rm*, to the left by *n* places, into the left-hand `32-n` bits of the result. And it sets the right-hand *n* bits of the result to 0. See Figure 3-3 on page 3-49.

You can use he `LSL #n` operation to multiply the value in the register *Rm* by 2^n, if the value is regarded as an unsigned integer or a two's complement signed integer. Overflow can occur without warning.

When the instruction is `LSLS` or when `LSL #n`, with non-zero *n*, is used in *`Operand2`* with the instructions `MOVS`, `MVNS`, `ANDS`, `ORRS`, `ORNS`, `EORS`, `BICS`, `TEQ` or `TST`, the carry flag is updated to the last bit shifted out, bit[32-*n*], of the register *Rm*. These instructions do not affect the carry flag when used with `LSL #0`.

——— **Note** ———

- If *n* is 32 or more, then all the bits in the result are cleared to 0.
- If *n* is 33 or more and the carry flag is updated, it is updated to 0.

———————————

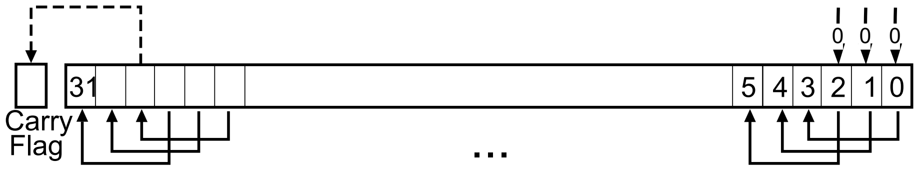

Figure 3-3 LSL #3

ROR

Rotate right by *n* bits moves the left-hand 32-*n* bits of the register *Rm*, to the right by *n* places, into the right-hand 32-*n* bits of the result. And it moves the right-hand *n* bits of the register into the left-hand *n* bits of the result. See Figure 3-4.

When the instruction is `RORS` or when `ROR #n` is used in *`Operand2`* with the instructions `MOVS`, `MVNS`, `ANDS`, `ORRS`, `ORNS`, `EORS`, `BICS`, `TEQ` or `TST`, the carry flag is updated to the last bit rotation, bit[*n*-1], of the register *Rm*.

——— **Note** ———

- If *n* is 32, then the value of the result is same as the value in *Rm*, and if the carry flag is updated, it is updated to bit[31] of *Rm*.
- `ROR` with shift length, *n*, more than 32 is the same as `ROR` with shift length *n*-32.

———————————

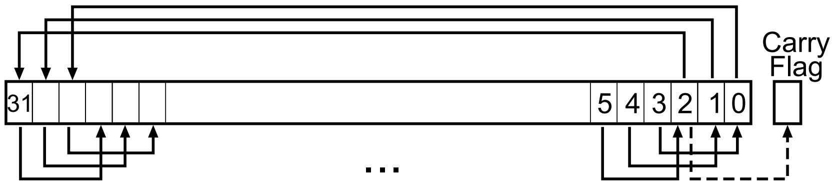

Figure 3-4 ROR #3

RRX

Rotate right with extend moves the bits of the register *Rm* to the right by one bit. And it copies the carry flag into bit[31] of the result. See Figure 3-5 on page 3-50.

When the instruction is `RRXS` or when `RRX` is used in *`Operand2`* with the instructions `MOVS`, `MVNS`, `ANDS`, `ORRS`, `ORNS`, `EORS`, `BICS`, `TEQ` or `TST`, the carry flag is updated to bit[0] of the register *Rm*.

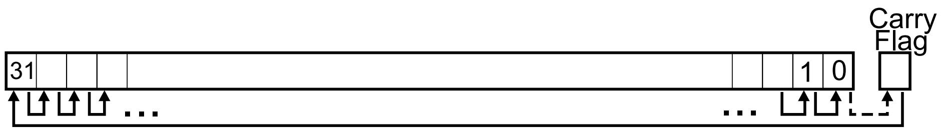

Figure 3-5 RRX

See also

Concepts

- *Flexible second operand (Operand2)* on page 3-46.

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

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

Syntax

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

op{cond} {Rd}, Rn, #imm12            ; 32-bit Thumb ADD and SUB only
```

where:

op	is one of:
	`ADD` Add.
	`ADC` Add with Carry.
	`SUB` Subtract.
	`RSB` Reverse Subtract.
	`SBC` Subtract with Carry.
	`RSC` Reverse Subtract with Carry (ARM only).
`S`	is an optional suffix. If `S` is specified, the condition code flags are updated on the result of the operation.
cond	is an optional condition code.
Rd	is the destination register.
Rn	is the register holding the first operand.
Operand2	is a flexible second operand.
imm12	is any value in the range 0-4095.

Usage

The `ADD` instruction adds the values in *Rn* and *Operand2* or *imm12*.

The `SUB` instruction subtracts the value of *Operand2* or *imm12* from the value in *Rn*.

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

You can use `ADC`, `SBC`, and `RSC` to synthesize multiword arithmetic.

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

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

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

In certain circumstances, the assembler can substitute one instruction for another. Be aware of this when reading disassembly listings.

Use of PC and SP in Thumb instructions

In most of these instructions, you cannot use PC (`R15`) for *Rd*, or any operand.

The exceptions are:

- you can use PC for *Rn* in 32-bit Thumb `ADD` and `SUB` instructions, with a constant *`Operand2`* value in the range 0-4095, and no `S` suffix. These instructions are useful for generating PC-relative addresses. Bit[1] of the PC value reads as 0 in this case, so that the base address for the calculation is always word-aligned.
- you can use PC in 16-bit Thumb `ADD{cond} Rd, Rd, Rm` instructions, where both registers cannot be PC. However, the following 16-bit Thumb instructions are deprecated in ARMv6T2 and above:
  — `ADD{cond} PC, SP, PC`
  — `ADD{cond} SP, SP, PC`.

In most of these instructions, you cannot use SP (`R13`) for *Rd*, or any operand. Except that:

- You can use SP for `Rn` in `ADD` and `SUB` instructions
- `ADD{cond} SP, SP, SP` is permitted but is deprecated in ARMv6T2 and above
- `ADD{S}{cond} SP, SP, Rm{,shift}` and `SUB{S}{cond} SP, SP, Rm{,shift}` are permitted if *`shift`* is omitted or `LSL #1`, `LSL #2`, or `LSL #3`.

Use of PC and SP in ARM instructions

You cannot use PC for *Rd* or any operand in any data processing instruction that has a register-controlled shift.

With the exception of `ADD` and `SUB`, use of PC for any operand, in instructions without register-controlled shift, is deprecated.

In `SUB` instructions without register-controlled shift, use of PC is deprecated except for the following cases:
- Use of PC for `Rd`
- Use of PC for `Rn` in the instruction `SUB{cond} Rd, Rn, #Constant`.

In `ADD` instructions without register-controlled shift, use of PC is deprecated except for the following cases:
- Use of PC for `Rd` in instructions that do not add SP to a register
- Use of PC for `Rn` and use of PC for `Rm` in instructions that add two registers other than SP
- Use of PC for `Rn` in the instruction `ADD{cond} Rd, Rn, #Constant`.

If you use PC (`R15`) as *Rn* or *Rm*, the value used is the address of the instruction plus 8.

If you use PC as *Rd*:
- Execution branches to the address corresponding to the result.
- If you use the `S` suffix, see the `SUBS pc,lr` instruction.

You can use SP for *Rn* in `ADD` and SUB instructions, however, `ADDS PC, SP, #Constant` and `SUBS PC, SP, #Constant` are deprecated.

You can use SP in `ADD` (register) and `SUB` (register) if *Rn* is SP and *`shift`* is omitted or `LSL #1`, `LSL #2`, or `LSL #3`.

Other uses of SP in these ARM instructions are deprecated.

Note

The deprecation of SP and PC in ARM instructions is only in ARMv6T2 and above.

Condition flags

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

16-bit instructions

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

`ADDS Rd, Rn, #imm`	*imm* range 0-7. *Rd* and *Rn* must both be Lo registers.
`ADDS Rd, Rn, Rm`	*Rd*, *Rn* and *Rm* must all be Lo registers.
`ADD Rd, Rd, Rm`	ARMv6 and earlier: either *Rd* or *Rm*, or both, must be a Hi register. ARMv6T2 and above: this restriction does not apply.
`ADDS Rd, Rd, #imm`	*imm* range 0-255. *Rd* must be a Lo register.
`ADCS Rd, Rd, Rm`	*Rd*, *Rn* and *Rm* must all be Lo registers.
`ADD SP, SP, #imm`	*imm* range 0-508, word aligned.
`ADD Rd, SP, #imm`	*imm* range 0-1020, word aligned. *Rd* must be a Lo register.
`ADD Rd, pc, #imm`	*imm* range 0-1020, word aligned. *Rd* must be a Lo register. Bits[1:0] of the PC are read as 0 in this instruction.
`SUBS Rd, Rn, Rm`	*Rd*, *Rn* and *Rm* must all be Lo registers.
`SUBS Rd, Rn, #imm`	*imm* range 0-7. *Rd* and *Rn* both Lo registers.
`SUBS Rd, Rd, #imm`	*imm* range 0-255. *Rd* must be a Lo register.
`SBCS Rd, Rd, Rm`	*Rd*, *Rn* and *Rm* must all be Lo registers.
`SUB SP, SP, #imm`	*imm* range 0-508, word aligned.
`RSBS Rd, Rn, #0`	*Rd* and *Rn* both Lo registers.

Examples

```
    ADD     r2, r1, r3
    SUBS    r8, r6, #240        ; sets the flags on the result
    RSB     r4, r4, #1280       ; subtracts contents of R4 from 1280
    ADCHI   r11, r0, r3         ; only executed if C flag set and Z
                                ; flag clear
    RSCSLE  r0,r5,r0,LSL r4     ; conditional, flags set
```

Incorrect example

```
    RSCSLE  r0,pc,r0,LSL r4    ; PC not permitted with register
                                ; controlled shift
```

Multiword arithmetic examples

These two instructions add a 64-bit integer contained in `R2` and `R3` to another 64-bit integer contained in `R0` and `R1`, and place the result in `R4` and `R5`.

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

These instructions subtract one 96-bit integer from another:

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

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

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

See also

Concepts

- *Flexible second operand (Operand2)* on page 3-46
- *Instruction substitution* on page 3-46.

Developing Software for ARM Processors:

- Chapter 6 *Handling Processor Exceptions*.

Reference

- *Parallel add and subtract* on page 3-103
- *SUBS pc, lr* on page 3-55
- *ADR (PC-relative)* on page 3-24
- *ADR (register-relative)* on page 3-26
- *ADRL pseudo-instruction* on page 3-156
- *Condition codes* on page 3-163.

3.4.6 SUBS pc, lr

Exception return, without popping anything from the stack.

Syntax

```
SUBS{cond} pc, lr, #imm            ; ARM and Thumb code

MOVS{cond} pc, lr                  ; ARM and Thumb code

op1S{cond} pc, Rn, #imm            ; ARM code only and is deprecated

op1S{cond} pc, Rn, Rm {, shift}    ; ARM code only and is deprecated

op2S{cond} pc, #imm                ; ARM code only and is deprecated

op2S{cond} pc, Rm {, shift}        ; ARM code only and is deprecated
```

where:

op1	is one of `ADC`, `ADD`, `AND`, `BIC`, `EOR`, `ORN`, `ORR`, `RSB`, `RSC`, `SBC`, and `SUB`.
op2	is one of `MOV` and `MVN`.
cond	is an optional condition code.
imm	is an immediate value. In Thumb code, it is limited to the range 0-255. In ARM code, it is a flexible second operand.
R*n*	is the first operand register. ARM deprecates the use of any register except `LR`.
R*m*	is the optionally shifted second or only operand register.
shift	is an optional condition code.

Usage

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

You can use `SUBS pc, lr, #imm` to return from an exception if there is no return state on the stack. The value of `#imm` depends on the exception to return from.

Notes

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

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

The results of breaking these rules are UNPREDICTABLE. However, no special precautions are required in software, if the instructions are used to return after a valid exception entry mechanism.

In Thumb, only `SUBS{cond} pc, lr, #imm` is a valid instruction. `MOVS pc, lr` is a synonym of `SUBS pc, lr, #0`. Other instructions are undefined.

In ARM, only `SUBS{cond} pc, lr, #imm` and `MOVS{cond} pc, lr` are valid instructions. Other instructions are deprecated in ARMv6T2 and above.

Caution

Do not use these instructions in User mode or System mode. The effect of such an instruction is UNPREDICTABLE, but the assembler cannot warn you at assembly time.

Architectures

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

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

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

See also

Concepts

- *Flexible second operand (Operand2)* on page 3-46.

Reference

- *ADD, SUB, RSB, ADC, SBC, and RSC* on page 3-51
- *AND, ORR, EOR, BIC, and ORN* on page 3-57
- *MOV and MVN* on page 3-62
- *Condition codes* on page 3-163.

3.4.7 AND, ORR, EOR, BIC, and ORN

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

Syntax

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

where:

op	is one of:
	`AND` logical AND.
	`ORR` logical OR.
	`EOR` logical Exclusive OR.
	`BIC` logical AND NOT.
	`ORN` logical OR NOT (Thumb only).
`S`	is an optional suffix. If `S` is specified, the condition code flags are updated on the result of the operation.
cond	is an optional condition code.
Rd	is the destination register.
Rn	is the register holding the first operand.
Operand2	is a flexible second operand.

Usage

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

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

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

In certain circumstances, the assembler can substitute `BIC` for `AND`, `AND` for `BIC`, `ORN` for `ORR`, or `ORR` for `ORN`. Be aware of this when reading disassembly listings.

Use of PC in 32-bit Thumb instructions

You cannot use PC (`R15`) for *Rd* or any operand in any of these instructions.

Use of PC and SP in ARM instructions

You can use PC and SP in these ARM instructions but they are deprecated in ARMv6T2 and above.

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

If you use PC as *Rd*:
- Execution branches to the address corresponding to the result.
- If you use the `S` suffix, see the `SUBS pc,lr` instruction.

You cannot use PC for any operand in any data processing instruction that has a register-controlled shift.

Condition flags

If `S` is specified, these instructions:

- update the N and Z flags according to the result
- can update the C flag during the calculation of *`Operand2`*
- do not affect the V flag.

16-bit instructions

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

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

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

Examples

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

Incorrect example

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

See also

Concepts

- *Flexible second operand (Operand2)* on page 3-46
- *Instruction substitution* on page 3-46.

Developing Software for ARM Processors:

- Chapter 6 *Handling Processor Exceptions*.

Reference

- *SUBS pc, lr* on page 3-55
- *Condition codes* on page 3-163.

3.4.8 CLZ

Count Leading Zeros.

Syntax

`CLZ{cond} Rd, Rm`

where:

cond	is an optional condition code.
Rd	is the destination register.
Rm	is the operand register.

Usage

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

Register restrictions

You cannot use PC for any operand.

You can use SP in these ARM instructions but this is deprecated in ARMv6T2 and above.

You cannot use SP in Thumb instructions.

Condition flags

This instruction does not change the flags.

Architectures

This ARM instruction is available in ARMv5T and above.

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

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

Examples

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

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

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

See also

Reference

- *Condition codes* on page 3-163.

3.4.9 CMP and CMN

Compare and Compare Negative.

Syntax

`CMP{cond} Rn, Operand2`

`CMN{cond} Rn, Operand2`

where:

cond	is an optional condition code.
Rn	is the ARM register holding the first operand.
Operand2	is a flexible second operand.

Usage

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

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

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

In certain circumstances, the assembler can substitute `CMN` for `CMP`, or `CMP` for `CMN`. Be aware of this when reading disassembly listings.

Use of PC in ARM and Thumb instructions

You cannot use PC for any operand in any data processing instruction that has a register-controlled shift.

You can use PC (`R15`) in these ARM instructions without register controlled shift but this is deprecated in ARMv6T2 and above.

If you use PC as *Rn* in ARM instructions, the value used is the address of the instruction plus 8.

You cannot use PC for any operand in these Thumb instructions.

Use of SP in ARM and Thumb instructions

You can use SP for *Rn* in ARM and Thumb instructions.

You can use SP for *Rm* in ARM instructions but this is deprecated in ARMv6T2 and above.

You can use SP for *Rm* in a 16-bit Thumb `CMP Rn, Rm` instruction but this is deprecated in ARMv6T2 and above. Other use of SP for *Rm* is not permitted in Thumb.

Condition flags

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

16-bit instructions

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

`CMP Rn, Rm`	Lo register restriction does not apply.
`CMN Rn, Rm`	*Rn* and *Rm* must both be Lo registers.
`CMP Rn, #imm`	*Rn* must be a Lo register. *imm* range 0-255.

Examples

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

Incorrect example

```
    CMP     r2, pc, ASR r0 ; PC not permitted with register-controlled shift
```

See also

Concepts

- *Flexible second operand (Operand2)* on page 3-46
- *Instruction substitution* on page 3-46.

Reference

- *Condition codes* on page 3-163.

3.4.10 MOV and MVN

Move and Move Not.

Syntax

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

`MOV{cond} Rd, #imm16`

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

where:

`S`	is an optional suffix. If `S` is specified, the condition code flags are updated on the result of the operation.
cond	is an optional condition code.
Rd	is the destination register.
Operand2	is a flexible second operand.
imm16	is any value in the range 0-65535.

Usage

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

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

In certain circumstances, the assembler can substitute `MVN` for `MOV`, or `MOV` for `MVN`. Be aware of this when reading disassembly listings.

Use of PC and SP in 32-bit Thumb MOV and MVN

You cannot use PC (`R15`) for *Rd*, or in *Operand2*, in 32-bit Thumb `MOV` or `MVN` instructions. With the following exceptions, you cannot use SP (`R13`) for *Rd*, or in *Operand2*:

- `MOV{cond}.W Rd, SP`, where *Rd* is not SP
- `MOV{cond}.W SP, Rm`, where *Rm* is not SP.

Use of PC and SP in 16-bit Thumb

You can use PC or SP in 16-bit Thumb `MOV{cond} Rd, Rm` instructions but these instructions in which both *Rd* and *Rm* are SP or PC are deprecated in ARMv6T2 and above.

You cannot use PC or SP in any other `MOV{S}` or `MVN{S}` 16-bit Thumb instructions.

Use of PC and SP in ARM MOV and MVN

You cannot use PC for *Rd* or any operand in any data processing instruction that has a register-controlled shift.

In instructions without register-controlled shift, use of PC is deprecated except the following cases:

- `MOVS PC, LR`
- `MOV PC, Rm` when `Rm` is not PC or SP
- `MOV Rd, PC` when `Rd` is not PC or SP.

You can use SP for `Rd` or `Rm`. But these are deprecated except the following cases:

- `MOV SP, Rm` when `Rm` is not PC or SP
- `MOV Rd, SP` when `Rd` is not PC or SP.

——— **Note** ———

- You cannot use PC for `Rd` in `MOV Rd, #imm16` if the `#imm16` value is not a permitted `Operand2` value. You can use PC in forms with `Operand2` without register-controlled shift.
- The deprecation of PC and SP in ARM instructions only apply to ARMv6T2 and above.

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

If you use PC as *Rd*:

- Execution branches to the address corresponding to the result.
- If you use the `S` suffix, see the `SUBS pc,lr` instruction.

Condition flags

If `S` is specified, these instructions:

- update the N and Z flags according to the result
- can update the C flag during the calculation of *Operand2*
- do not affect the V flag.

16-bit instructions

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

`MOVS Rd, #imm`	*Rd* must be a Lo register. *imm* range 0-255.
`MOVS Rd, Rm`	*Rd* and *Rm* must both be Lo registers.
`MOV Rd, Rm`	In architectures before ARMv6, either *Rd* or *Rm*, or both, must be a Hi register. In ARMv6 and above, this restriction does not apply.
`MVNS Rd, Rm`	*Rd* and *Rm* must both be Lo registers. This form can only be used outside an IT block.
`MVN{cond} Rd, Rm`	*Rd* and *Rm* must both be Lo registers. This form can only be used inside an IT block.

Architectures

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

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

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

Example

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

Incorrect example

```
    MVN     pc,r3,ASR r0     ; PC not permitted with register-controlled shift
```

See also

Concepts

- *Flexible second operand (Operand2)* on page 3-46
- *Instruction substitution* on page 3-46.

Developing Software for ARM Processors:

- Chapter 6 *Handling Processor Exceptions*.

Reference

- *Condition codes* on page 3-163
- *SUBS pc, lr* on page 3-55.

3.4.11 MOVT

Move Top. Writes a 16-bit immediate value to the top halfword of a register, without affecting the bottom halfword.

Syntax

`MOVT{cond} Rd, #imm16`

where:

cond	is an optional condition code.
Rd	is the destination register.
imm16	is a 16-bit immediate value.

Usage

`MOVT` writes *imm16* to *Rd*[31:16]. The write does not affect *Rd*[15:0].

You can generate any 32-bit immediate with a `MOV`, `MOVT` instruction pair. The assembler implements the `MOV32` pseudo-instruction for convenient generation of this instruction pair.

Register restrictions

You cannot use PC in ARM or Thumb instructions.

You can use SP for *Rd* in ARM instructions but this is deprecated.

You cannot use SP in Thumb instructions.

Condition flags

This instruction does not change the flags.

Architectures

This ARM instruction is available in ARMv6T2 and above.

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

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

See also

Reference

- *MOV32 pseudo--instruction* on page 3-158
- *Condition codes* on page 3-163.

3.4.12 TST and TEQ

Test bits and Test Equivalence.

Syntax

`TST{cond} Rn, Operand2`

`TEQ{cond} Rn, Operand2`

where:

cond	is an optional condition code.
Rn	is the ARM register holding the first operand.
Operand2	is a flexible second operand.

Usage

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

The `TST` instruction performs a bitwise AND operation on the value in *Rn* and the value of *Operand2*. This is the same as an `ANDS` instruction, except that the result is discarded.

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

Use the `TEQ` instruction to test if two values are equal, without affecting the V or C flags (as `CMP` does).

`TEQ` is also useful for testing the sign of a value. After the comparison, the N flag is the logical Exclusive OR of the sign bits of the two operands.

Register restrictions

In these Thumb instructions, you cannot use SP or PC for *Rn* or *Operand2*.

In these ARM instructions, use of SP or PC is deprecated in ARMv6T2 and above.

For ARM instructions:

- if you use PC (`R15`) as *Rn*, the value used is the address of the instruction plus 8
- you cannot use PC for any operand in any data processing instruction that has a register-controlled shift.

Condition flags

These instructions:

- update the N and Z flags according to the result
- can update the C flag during the calculation of *Operand2*
- do not affect the V flag.

16-bit instructions

The following form of the `TST` instruction is available in Thumb code, and is a 16-bit instruction:

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

Architectures

These ARM instructions are available in all architectures with ARM.

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

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

Examples

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

Incorrect example

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

See also

Concepts

- *Flexible second operand (Operand2)* on page 3-46.

Reference

- *Condition codes* on page 3-163.

3.4.13 SEL

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

Syntax

SEL{*cond*} {*Rd*}, *Rn*, *Rm*

where:

cond	is an optional condition code.
Rd	is the destination register.
Rn	is the register holding the first operand.
Rm	is the register holding the second operand.

Operation

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

- if GE[0] is set, `Rd[7:0]` come from `Rn[7:0]`, otherwise from `Rm[7:0]`
- if GE[1] is set, `Rd[15:8]` come from `Rn[15:8]`, otherwise from `Rm[15:8]`
- if GE[2] is set, `Rd[23:16]` come from `Rn[23:16]`, otherwise from `Rm[23:16]`
- if GE[3] is set, `Rd[31:24]` come from `Rn[31:24]`, otherwise from `Rm[31:24]`.

Usage

Use the `SEL` instruction after one of the signed parallel instructions. You can use this to select maximum or minimum values in multiple byte or halfword data.

Register restrictions

You cannot use PC for any register.

You can use SP in ARM instructions but these are deprecated in ARMv6T2 and above. You cannot use SP in Thumb instructions.

Condition flags

This instruction does not change the flags.

Architectures

This ARM instruction is available in ARMv6 and above.

These 32-bit Thumb instructions are available in ARMv6T2 and above. For the ARMv7-M architecture, they are only available in an ARMv7E-M implementation.

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

Examples

```
    SEL     r0, r4, r5
    SELLT   r4, r0, r4
```

The following instruction sequence sets each byte in `R4` equal to the unsigned minimum of the corresponding bytes of `R1` and `R2`:

```
    USUB8    r4, r1, r2
    SEL      r4, r2, r1
```

See also

Reference

- *Parallel add and subtract* on page 3-103
- *Condition codes* on page 3-163.

3.4.14 REV, REV16, REVSH, and RBIT

Reverse bytes or bits within words or halfwords.

Syntax

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

where:

op	is any one of the following:
	`REV` Reverse byte order in a word.
	`REV16` Reverse byte order in each halfword independently.
	`REVSH` Reverse byte order in the bottom halfword, and sign extend to 32 bits.
	`RBIT` Reverse the bit order in a 32-bit word.
cond	is an optional condition code.
Rd	is the destination register.
Rn	is the register holding the operand.

Usage

You can use these instructions to change endianness:

`REV`	converts 32-bit big-endian data into little-endian data or 32-bit little-endian data into big-endian data.
`REV16`	converts 16-bit big-endian data into little-endian data or 16-bit little-endian data into big-endian data.
`REVSH`	converts either: • 16-bit signed big-endian data into 32-bit signed little-endian data • 16-bit signed little-endian data into 32-bit signed big-endian data.

Register restrictions

You cannot use PC for any register.

You can use SP in ARM instructions but these are deprecated in ARMv6T2 and above. You cannot use SP in Thumb instructions.

Condition flags

These instructions do not change the flags.

16-bit instructions

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

`REV` *Rd*, *Rm*	*Rd* and *Rm* must both be Lo registers.
`REV16` *Rd*, *Rm*	*Rd* and *Rm* must both be Lo registers.
`REVSH` *Rd*, *Rm*	*Rd* and *Rm* must both be Lo registers.

Architectures

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

The RBIT ARM instruction is available in ARMv6T2 and above.

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

These 16-bit Thumb instructions are available in ARMv6 and above.

Examples

```
    REV     r3, r7
    REV16   r0, r0
    REVSH   r0, r5      ; Reverse Signed Halfword
    REVHS   r3, r7      ; Reverse with Higher or Same condition
    RBIT    r7, r8
```

See also

Reference

- *Condition codes* on page 3-163.

3.4.15 ASR, LSL, LSR, ROR, and RRX

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

These instructions are the preferred synonyms for `MOV` instructions with shifted register operands.

Syntax

op{S}{*cond*} *Rd*, *Rm*, *Rs*

op{S}{*cond*} *Rd*, *Rm*, *#sh*

RRX{S}{*cond*} *Rd*, *Rm*

where:

op	is one of `ASR`, `LSL`, `LSR`, or `ROR`.
`S`	is an optional suffix. If `S` is specified, the condition code flags are updated on the result of the operation.
Rd	is the destination register.
Rm	is the register holding the first operand. This operand is shifted right.
Rs	is a register holding a shift value to apply to the value in *Rm*. Only the least significant byte is used.
sh	is a constant shift. The range of values permitted depends on the instruction:

`ASR`	permitted shifts 1-32
`LSL`	permitted shifts 0-31
`LSR`	permitted shifts 1-32
`ROR`	permitted shifts 1-31.

Usage

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

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

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

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

Restrictions in Thumb code

Thumb instructions must not use PC or SP.

If you specify zero for the *sh* value in an `LSL` instruction in an IT block, the result is UNPREDICTABLE.

Use of SP and PC in ARM ASR, LSL, LSR, ROR, and RRX instructions

You can use SP in these ARM instructions but these are deprecated in ARMv6T2 and above.

You cannot use PC in instructions with the `op{S}{cond} Rd, Rm, Rs` syntax. You can use PC for `Rd` and `Rm` in the other syntaxes, but these are deprecated in ARMv6T2 and above.

If you use PC as *`Rm`*, the value used is the address of the instruction plus 8.

If you use PC as *`Rd`*:

- Execution branches to the address corresponding to the result.
- If you use the `S` suffix, the SPSR of the current mode is copied to the CPSR. You can use this to return from exceptions.

  ——— Note ———

  The ARM instructions *`op`*`S{`*`cond`*`} pc,Rm,#sh` and `RRXS{cond} pc,Rm` always disassemble to the preferred form `MOVS{`*`cond`*`} pc,Rm{,shift}`.

——— Caution ———

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

You cannot use PC for *`Rd`* or any operand in any of these instructions if they have a register-controlled shift.

Condition flags

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

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

16-bit instructions

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

`ASRS` *`Rd`*, *`Rm`*, *`#sh`*	*`Rd`* and *`Rm`* must both be Lo registers.
`ASRS` *`Rd`*, *`Rd`*, *`Rs`*	*`Rd`* and *`Rs`* must both be Lo registers.
`LSLS` *`Rd`*, *`Rm`*, *`#sh`*	*`Rd`* and *`Rm`* must both be Lo registers.
`LSLS` *`Rd`*, *`Rd`*, *`Rs`*	*`Rd`* and *`Rs`* must both be Lo registers.
`LSRS` *`Rd`*, *`Rm`*, *`#sh`*	*`Rd`* and *`Rm`* must both be Lo registers.
`LSRS` *`Rd`*, *`Rd`*, *`Rs`*	*`Rd`* and *`Rs`* must both be Lo registers.
`RORS` *`Rd`*, *`Rd`*, *`Rs`*	*`Rd`* and *`Rs`* must both be Lo registers.

Architectures

These ARM instructions are available in all architectures.

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

These 16-bit Thumb instructions are available in ARMv4T and above.

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

Examples

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

See also

Reference

- *MOV and MVN* on page 3-62
- *Condition codes* on page 3-163.

3.4.16 SDIV and UDIV

Signed and Unsigned Divide.

Syntax

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

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

where:

cond	is an optional condition code.
Rd	is the destination register.
Rn	is the register holding the value to be divided.
Rm	is a register holding the divisor.

Register restrictions

PC or SP cannot be used for `Rd`, `Rn` or `Rm`.

Architectures

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

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

See also

Reference

- *Condition codes* on page 3-163.

3.5 Multiply instructions

This section contains the following subsections:

- *MUL, MLA, and MLS* on page 3-77

  Multiply, Multiply Accumulate, and Multiply Subtract (32-bit by 32-bit, bottom 32-bit result).

- *UMULL, UMLAL, SMULL, and SMLAL* on page 3-79

  Unsigned and signed Long Multiply and Multiply Accumulate (32-bit by 32-bit, 64-bit result or 64-bit accumulator).

- *SMULxy and SMLAxy* on page 3-81

  Signed Multiply and Signed Multiply Accumulate (16-bit by 16-bit, 32-bit result).

- *SMULWy and SMLAWy* on page 3-83

  Signed Multiply and Signed Multiply Accumulate(32-bit by 16-bit, top 32-bit result).

- *SMLALxy* on page 3-84

  Signed Multiply Accumulate (16-bit by 16-bit, 64-bit accumulate).

- *SMUAD{X} and SMUSD{X}* on page 3-86

  Dual 16-bit Signed Multiply with Addition or Subtraction of products.

- *SMMUL, SMMLA, and SMMLS* on page 3-88

  Multiply, Multiply Accumulate, and Multiply Subtract (32-bit by 32-bit, top 32-bit result).

- *SMLAD and SMLSD* on page 3-90

  Dual 16-bit Signed Multiply, 32-bit Accumulation of Sum or Difference of 32-bit products.

- *SMLALD and SMLSLD* on page 3-92

  Dual 16-bit Signed Multiply, 64-bit Accumulation of Sum or Difference of 32-bit products.

- *UMAAL* on page 3-94

  Unsigned Multiply Accumulate Accumulate Long.

- *MIA, MIAPH, and MIAxy* on page 3-95

  Multiplies with Internal Accumulate (XScale coprocessor 0 instructions).

3.5.1 MUL, MLA, and MLS

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

Syntax

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

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

`MLS{cond} Rd, Rn, Rm, Ra`

where:

cond	is an optional condition code.
`S`	is an optional suffix. If `S` is specified, the condition code flags are updated on the result of the operation.
Rd	is the destination register.
Rn, Rm	are registers holding the values to be multiplied.
Ra	is a register holding the value to be added or subtracted from.

Usage

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

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

The `MLS` instruction multiplies the values from *Rn* and *Rm*, subtracts the result from the value from *Ra*, and places the least significant 32 bits of the final result in *Rd*.

Register restrictions

For the `MUL` and `MLA` instructions, *Rn* must be different from *Rd* in architectures before ARMv6.

You cannot use PC for any register.

You can use SP in ARM instructions but these are deprecated in ARMv6T2 and above. You cannot use SP in Thumb instructions.

Condition flags

If `S` is specified, the `MUL` and `MLA` instructions:

- update the N and Z flags according to the result
- corrupt the C and V flag in ARMv4
- do not affect the C or V flag in ARMv5T and above.

Thumb instructions

The following form of the `MUL` instruction is available in Thumb code, and is a 16-bit instruction:

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

There are no other Thumb multiply instructions that can update the condition code flags.

Architectures

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

The MLS ARM instruction is available in ARMv6T2 and above.

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

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

Examples

```
    MUL     r10, r2, r5
    MLA     r10, r2, r1, r5
    MULS    r0, r2, r2
    MULLT   r2, r3, r2
    MLS     r4, r5, r6, r7
```

See also

Reference

- *Condition codes* on page 3-163.

3.5.2 UMULL, UMLAL, SMULL, and SMLAL

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

Syntax

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

where:

Op	is one of `UMULL`, `UMLAL`, `SMULL`, or `SMLAL`.
S	is an optional suffix available in ARM state only. If S is specified, the condition code flags are updated on the result of the operation.
cond	is an optional condition code.
RdLo, *RdHi*	are the destination registers. For `UMLAL` and `SMLAL` they also hold the accumulating value. *RdLo* and *RdHi* must be different registers
Rn, Rm	are ARM registers holding the operands.

Usage

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

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

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

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

Register restrictions

Rn must be different from *RdLo* and *RdHi* in architectures before ARMv6.

You cannot use PC for any register.

You can use SP in ARM instructions but these are deprecated in ARMv6T2 and above. You cannot use SP in Thumb instructions.

Condition flags

If S is specified, these instructions:
- update the N and Z flags according to the result
- do not affect the C or V flags.

Architectures

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

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

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

Examples

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

See also

Reference

- *Condition codes* on page 3-163.

3.5.3 SMUL*xy* and SMLA*xy*

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

Syntax

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

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

where:

<x>	is either `B` or `T`. `B` means use the bottom half (bits [15:0]) of *Rn*, `T` means use the top half (bits [31:16]) of *Rn*.
<y>	is either `B` or `T`. `B` means use the bottom half (bits [15:0]) of *Rm*, `T` means use the top half (bits [31:16]) of *Rm*.
cond	is an optional condition code.
Rd	is the destination register.
Rn, Rm	are the registers holding the values to be multiplied.
Ra	is the register holding the value to be added.

Usage

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

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

Register restrictions

You cannot use PC for any register.

You can use SP in ARM instructions but these are deprecated in ARMv6T2 and above. You cannot use SP in Thumb instructions.

Condition flags

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

If overflow occurs in the accumulation, `SMLAxy` sets the Q flag. To read the state of the Q flag, use an `MRS` instruction.

Note

`SMLAxy` never clears the Q flag. To clear the Q flag, use an `MSR` instruction.

Architectures

These ARM instructions are available in ARMv6 and above, and E variants of ARMv5T.

These 32-bit Thumb instructions are available in ARMv6T2 and above. For the ARMv7-M architecture, they are only available in an ARMv7E-M implementation.

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

Examples

```
    SMULTBEQ    r8, r7, r9
    SMLABBNE    r0, r2, r1, r10
    SMLABT      r0, r0, r3, r5
```

See also

Reference

- *MRS* on page 3-137
- *MSR* on page 3-139
- *Condition codes* on page 3-163.

3.5.4 SMULW*y* and SMLAW*y*

Signed Multiply Wide and Signed Multiply-Accumulate Wide, with one 32-bit and one 16-bit operand, providing the top 32-bits of the result.

Syntax

`SMULW<y>{cond} {Rd}, Rn, Rm`

`SMLAW<y>{cond} Rd, Rn, Rm, Ra`

where:

<y>	is either `B` or `T`. `B` means use the bottom half (bits [15:0]) of *Rm*, `T` means use the top half (bits [31:16]) of *Rm*.
cond	is an optional condition code.
Rd	is the destination register.
Rn, Rm	are the registers holding the values to be multiplied.
Ra	is the register holding the value to be added.

Usage

`SMULWy` multiplies the signed integer from the selected half of *Rm* by the signed integer from *Rn*, and places the upper 32-bits of the 48-bit result in *Rd*.

`SMLAWy` multiplies the signed integer from the selected half of *Rm* by the signed integer from *Rn*, adds the 32-bit result to the 32-bit value in *Ra*, and places the result in *Rd*.

Register restrictions

You cannot use PC for any register.

You can use SP in ARM instructions but these are deprecated in ARMv6T2 and above. You cannot use SP in Thumb instructions.

Condition flags

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

If overflow occurs in the accumulation, `SMLAWy` sets the Q flag.

Architectures

These ARM instructions are available in ARMv6 and above, and E variants of ARMv5T.

These 32-bit Thumb instructions are available in ARMv6T2 and above. For the ARMv7-M architecture, they are only available in an ARMv7E-M implementation.

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

See also

Reference

- *MRS* on page 3-137
- *Condition codes* on page 3-163.

3.5.5 SMLAL*xy*

Signed Multiply-Accumulate with 16-bit operands and a 64-bit accumulator.

Syntax

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

where:

<x>	is either `B` or `T`. `B` means use the bottom half (bits [15:0]) of *Rn*, `T` means use the top half (bits [31:16]) of *Rn*.
<y>	is either `B` or `T`. `B` means use the bottom half (bits [15:0]) of *Rm*, `T` means use the top half (bits [31:16]) of *Rm*.
cond	is an optional condition code.
RdLo, RdHi	are the destination registers. They also hold the accumulate value. *RdHi* and *RdLo* must be different registers.
Rn, Rm	are the registers holding the values to be multiplied.

Usage

`SMLALxy` multiplies the signed integer from the selected half of *Rm* by the signed integer from the selected half of *Rn*, and adds the 32-bit result to the 64-bit value in *RdHi* and *RdLo*.

Register restrictions

You cannot use PC for any register.

You can use SP in ARM instructions but these are deprecated in ARMv6T2 and above. You cannot use SP in Thumb instructions.

Condition flags

This instruction does not change the flags.

——— **Note** ———

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

————————

Architectures

This ARM instruction is available in ARMv6 and above, and E variants of ARMv5T.

These 32-bit Thumb instructions are available in ARMv6T2 and above. For the ARMv7-M architecture, they are only available in an ARMv7E-M implementation.

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

Examples

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

See also

Reference

- *Condition codes* on page 3-163.

3.5.6 SMUAD{X} and SMUSD{X}

Dual 16-bit Signed Multiply with Addition or Subtraction of products, and optional exchange of operand halves.

Syntax

`op{X}{cond} {Rd}, Rn, Rm`

where:

op	is one of: `SMUAD` Dual multiply, add products. `SMUSD` Dual multiply, subtract products.
`X`	is an optional parameter. If `X` is present, the most and least significant halfwords of the second operand are exchanged before the multiplications occur.
cond	is an optional condition code.
Rd	is the destination register.
Rn, Rm	are the registers holding the operands.

Usage

`SMUAD` multiplies the bottom halfword of *Rn* with the bottom halfword of *Rm*, and the top halfword of *Rn* with the top halfword of *Rm*. It then adds the products and stores the sum to *Rd*.

`SMUSD` multiplies the bottom halfword of *Rn* with the bottom halfword of *Rm*, and the top halfword of *Rn* with the top halfword of *Rm*. It then subtracts the second product from the first, and stores the difference to *Rd*.

Register restrictions

You cannot use PC for any register.

You can use SP in ARM instructions but these are deprecated in ARMv6T2 and above. You cannot use SP in Thumb instructions.

Condition flags

The `SMUAD` instruction sets the Q flag if the addition overflows.

Architectures

These ARM instructions are available in ARMv6 and above.

These 32-bit Thumb instructions are available in ARMv6T2 and above. For the ARMv7-M architecture, they are only available in an ARMv7E-M implementation.

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

Examples

```
    SMUAD       r2, r3, r2
    SMUSDXNE    r0, r1, r2
```

See also

Reference

- *Condition codes* on page 3-163.

3.5.7 SMMUL, SMMLA, and SMMLS

Signed Most significant word Multiply, Signed Most significant word Multiply with Accumulation, and Signed Most significant word Multiply with Subtraction. These instructions have 32-bit operands and produce only the most significant 32-bits of the result.

Syntax

```
SMMUL{R}{cond} {Rd}, Rn, Rm
```

```
SMMLA{R}{cond} Rd, Rn, Rm, Ra
```

```
SMMLS{R}{cond} Rd, Rn, Rm, Ra
```

where:

`R`	is an optional parameter. If `R` is present, the result is rounded, otherwise it is truncated.
`cond`	is an optional condition code.
`Rd`	is the destination register.
`Rn, Rm`	are the registers holding the operands.
`Ra`	is a register holding the value to be added or subtracted from.

Operation

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

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

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

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

Register restrictions

You cannot use PC for any register.

You can use SP in ARM instructions but these are deprecated in ARMv6T2 and above. You cannot use SP in Thumb instructions.

Condition flags

These instructions do not change the flags.

Architectures

These ARM instructions are available in ARMv6 and above.

These 32-bit Thumb instructions are available in ARMv6T2 and above. For the ARMv7-M architecture, they are only available in an ARMv7E-M implementation.

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

Examples

```
    SMMULGE     r6, r4, r3
    SMMULR      r2, r2, r2
```

See also

Reference

- *Condition codes* on page 3-163.

3.5.8 SMLAD and SMLSD

Dual 16-bit Signed Multiply with Addition or Subtraction of products and 32-bit accumulation.

Syntax

op{X}{*cond*} *Rd*, *Rn*, *Rm*, *Ra*

where:

op	is one of: `SMLAD` Dual multiply, accumulate sum of products. `SMLSD` Dual multiply, accumulate difference of products.
cond	is an optional condition code.
`X`	is an optional parameter. If `X` is present, the most and least significant halfwords of the second operand are exchanged before the multiplications occur.
Rd	is the destination register.
Rn, Rm	are the registers holding the operands.
Ra	is the register holding the accumulate operand.

Operation

`SMLAD` multiplies the bottom halfword of *Rn* with the bottom halfword of *Rm*, and the top halfword of *Rn* with the top halfword of *Rm*. It then adds both products to the value in *Ra* and stores the sum to *Rd*.

`SMLSD` multiplies the bottom halfword of *Rn* with the bottom halfword of *Rm*, and the top halfword of *Rn* with the top halfword of *Rm*. It then subtracts the second product from the first, adds the difference to the value in *Ra*, and stores the result to *Rd*.

Register restrictions

You cannot use PC for any register.

You can use SP in ARM instructions but these are deprecated in ARMv6T2 and above. You cannot use SP in Thumb instructions.

Condition flags

These instructions do not change the flags.

Architectures

These ARM instructions are available in ARMv6 and above.

These 32-bit Thumb instructions are available in ARMv6T2 and above. For the ARMv7-M architecture, they are only available in an ARMv7E-M implementation.

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

Examples

```
    SMLSD       r1, r2, r0, r7
    SMLSDX      r11, r10, r2, r3
    SMLADLT     r1, r2, r4, r1
```

See also

Reference

- *Condition codes* on page 3-163.

3.5.9 SMLALD and SMLSLD

Dual 16-bit Signed Multiply with Addition or Subtraction of products and 64-bit Accumulation.

Syntax

`op{X}{cond} RdLo, RdHi, Rn, Rm`

where:

op	is one of: `SMLALD` Dual multiply, accumulate sum of products. `SMLSLD` Dual multiply, accumulate difference of products.
`X`	is an optional parameter. If `X` is present, the most and least significant halfwords of the second operand are exchanged before the multiplications occur.
cond	is an optional condition code.
RdLo, *RdHi*	are the destination registers for the 64-bit result. They also hold the 64-bit accumulate operand. *RdHi* and *RdLo* must be different registers.
Rn, Rm	are the registers holding the operands.

Operation

`SMLALD` multiplies the bottom halfword of *Rn* with the bottom halfword of *Rm*, and the top halfword of *Rn* with the top halfword of *Rm*. It then adds both products to the value in *RdLo*, *RdHi* and stores the sum to *RdLo*, *RdHi*.

`SMLSLD` multiplies the bottom halfword of *Rn* with the bottom halfword of *Rm*, and the top halfword of *Rn* with the top halfword of *Rm*. It then subtracts the second product from the first, adds the difference to the value in *RdLo*, *RdHi*, and stores the result to *RdLo*, *RdHi*.

Register restrictions

You cannot use PC for any register.

You can use SP in ARM instructions but these are deprecated in ARMv6T2 and above. You cannot use SP in Thumb instructions.

Condition flags

These instructions do not change the flags.

Architectures

These ARM instructions are available in ARMv6 and above.

These 32-bit Thumb instructions are available in ARMv6T2 and above. For the ARMv7-M architecture, they are only available in an ARMv7E-M implementation.

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

Examples

```
    SMLALD      r10, r11, r5, r1
    SMLSLD      r3, r0, r5, r1
```

See also

Reference

- *Condition codes* on page 3-163.

3.5.10 UMAAL

Unsigned Multiply Accumulate Accumulate Long.

Syntax

`UMAAL{cond} RdLo, RdHi, Rn, Rm`

where:

cond	is an optional condition code.
RdLo, *RdHi*	are the destination registers for the 64-bit result. They also hold the two 32-bit accumulate operands. *RdLo* and *RdHi* must be different registers.
Rn, Rm	are the registers holding the multiply operands.

Operation

The `UMAAL` instruction multiplies the 32-bit values in *Rn* and *Rm*, adds the two 32-bit values in *RdHi* and *RdLo*, and stores the 64-bit result to *RdLo*, *RdHi*.

Register restrictions

You cannot use PC for any register.

You can use SP in ARM instructions but these are deprecated in ARMv6T2 and above. You cannot use SP in Thumb instructions.

Condition flags

This instruction does not change the flags.

Architectures

This ARM instruction is available in ARMv6 and above.

These 32-bit Thumb instructions are available in ARMv6T2 and above. For the ARMv7-M architecture, they are only available in an ARMv7E-M implementation.

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

Examples

```
    UMAAL       r8, r9, r2, r3
    UMAALGE     r2, r0, r5, r3
```

See also

Reference

- *Condition codes* on page 3-163.

3.5.11 MIA, MIAPH, and MIAxy

Multiply with internal accumulate (32-bit by 32-bit, 40-bit accumulate).

Multiply with internal accumulate, packed halfwords (16-bit by 16-bit twice, 40-bit accumulate).

Multiply with internal accumulate (16-bit by 16-bit, 40-bit accumulate).

Syntax

```
MIA{cond} Acc, Rn, Rm
```

```
MIAPH{cond} Acc, Rn, Rm
```

```
MIA<x><y>{cond} Acc, Rn, Rm
```

where:

cond	is an optional condition code.
Acc	is the internal accumulator. The standard name is `acc`*x*, where *x* is an integer in the range 0 to *n*. The value of *n* depends on the processor. It is 0 in current processors.
Rn, Rm	are the ARM registers holding the values to be multiplied. *Rn* and *Rm* must not be PC.
<*x*><*y*>	is one of: `BB`, `BT`, `TB`, `TT`.

Usage

The `MIA` instruction multiplies the signed integers from *Rn* and *Rm*, and adds the result to the 40-bit value in *Acc*.

The `MIAPH` instruction multiplies the signed integers from the bottom halves of *Rn* and *Rm*, multiplies the signed integers from the upper halves of *Rn* and *Rm*, and adds the two 32-bit results to the 40-bit value in *Acc*.

The `MIAxy` instruction multiplies the signed integer from the selected half of *Rs* by the signed integer from the selected half of *Rm*, and adds the 32-bit result to the 40-bit value in *Acc*. <*x*> == `B` means use the bottom half (bits [15:0]) of *Rn*, <*x*> == `T` means use the top half (bits [31:16]) of *Rn*. <*y*> == `B` means use the bottom half (bits [15:0]) of *Rm*, <*y*> == `T` means use the top half (bits [31:16]) of *Rm*.

Condition flags

These instructions do not change the flags.

Note

These instructions cannot raise an exception. If overflow occurs on these instructions, the result wraps round without any warning.

Architectures

These ARM coprocessor 0 instructions are only available in XScale processors.

There are no Thumb versions of these instructions.

Examples

```
MIA     acc0,r5,r0
MIALE   acc0,r1,r9
MIAPH   acc0,r0,r7
MIAPHNE acc0,r11,r10
MIABB   acc0,r8,r9
MIABT   acc0,r8,r8
MIATB   acc0,r5,r3
MIATT   acc0,r0,r6
MIABTGT acc0,r2,r5
```

See also

Reference

- *Condition codes* on page 3-163.

3.6 Saturating instructions

This section contains:

- *Saturating arithmetic*
- *QADD, QSUB, QDADD, and QDSUB* on page 3-98
- *SSAT and USAT* on page 3-100.

Some of the parallel instructions are also saturating.

3.6.1 Saturating arithmetic

These operations are *saturating* (`SAT`). This means that, for some value of 2^n that depends on the instruction:

- for a signed saturating operation, if the full result would be less than -2^n, the result returned is -2^n
- for an unsigned saturating operation, if the full result would be negative, the result returned is zero
- if the full result would be greater than $2^n - 1$, the result returned is $2^n - 1$.

When any of these things occurs, it is called *saturation*. Some instructions set the Q flag when saturation occurs.

Note

Saturating instructions do not clear the Q flag when saturation does not occur. To clear the Q flag, use an `MSR` instruction.

The Q flag can also be set by two other instructions, but these instructions do not saturate.

See also

Reference

- *MSR* on page 3-139
- *SMULxy and SMLAxy* on page 3-81
- *SMULWy and SMLAWy* on page 3-83
- *Parallel instructions* on page 3-102.

3.6.2 QADD, QSUB, QDADD, and QDSUB

Signed Add, Subtract, Double and Add, Double and Subtract, saturating the result to the signed range $-2^{31} \leq x \leq 2^{31}-1$.

Syntax

`op{cond} {Rd}, Rm, Rn`

where:

op	is one of `QADD`, `QSUB`, `QDADD`, or `QDSUB`.
cond	is an optional condition code.
Rd	is the destination register.
Rm, Rn	are the registers holding the operands.

Usage

The `QADD` instruction adds the values in *Rm* and *Rn*.

The `QSUB` instruction subtracts the value in *Rn* from the value in *Rm*.

The `QDADD` instruction calculates `SAT(Rm + SAT(Rn * 2))`. Saturation can occur on the doubling operation, on the addition, or on both. If saturation occurs on the doubling but not on the addition, the Q flag is set but the final result is unsaturated.

The `QDSUB` instruction calculates `SAT(Rm - SAT(Rn * 2))`. Saturation can occur on the doubling operation, on the subtraction, or on both. If saturation occurs on the doubling but not on the subtraction, the Q flag is set but the final result is unsaturated.

——— **Note** ———
All values are treated as two's complement signed integers by these instructions.

Register restrictions

You cannot use PC for any operand.

You can use SP in ARM instructions but these are deprecated in ARMv6T2 and above. You cannot use SP in Thumb instructions.

Condition flags

If saturation occurs, these instructions set the Q flag. To read the state of the Q flag, use an `MRS` instruction.

Architectures

These ARM instructions are available in ARMv6 and above, and E variants of ARMv5T.

These 32-bit Thumb instructions are available in ARMv6T2 and above. For the ARMv7-M architecture, they are only available in an ARMv7E-M implementation.

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

Examples

```
    QADD    r0, r1, r9
    QDSUBLT r9, r0, r1
```

See also

Reference

- *Parallel add and subtract* on page 3-103
- *MRS* on page 3-137
- *Condition codes* on page 3-163.

3.6.3 SSAT and USAT

Signed Saturate and Unsigned Saturate to any bit position, with optional shift before saturating.

`SSAT` saturates a signed value to a signed range.

`USAT` saturates a signed value to an unsigned range.

Syntax

op{*cond*} *Rd*, #*sat*, *Rm*{, *shift*}

where:

op	is either `SSAT` or `USAT`.
cond	is an optional condition code.
Rd	is the destination register.
sat	specifies the bit position to saturate to, in the range 1 to 32 for `SSAT`, and 0 to 31 for `USAT`.
Rm	is the register containing the operand.
shift	is an optional shift. It must be one of the following:

`ASR #`*n*	where *n* is in the range 1-32 (ARM) or 1-31 (Thumb)
`LSL #`*n*	where *n* is in the range 0-31.

Operation

The `SSAT` instruction applies the specified shift, then saturates to the signed range $-2^{sat-1} \leq x \leq 2^{sat-1} - 1$.

The `USAT` instruction applies the specified shift, then saturates to the unsigned range $0 \leq x \leq 2^{sat} - 1$.

Register restrictions

You cannot use PC for any register.

You can use SP in ARM instructions but these are deprecated in ARMv6T2 and above. You cannot use SP in Thumb instructions.

Condition flags

If saturation occurs, these instructions set the Q flag. To read the state of the Q flag, use an `MRS` instruction.

Architectures

These ARM instructions are available in ARMv6 and above.

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

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

Examples

```
    SSAT    r7, #16, r7, LSL #4
    USATNE  r0, #7, r5
```

See also

Reference

- *SSAT16 and USAT16* on page 3-107
- *MRS* on page 3-137
- *Condition codes* on page 3-163.

3.7 Parallel instructions

This section contains:

- *Parallel add and subtract* on page 3-103

  Various byte-wise and halfword-wise additions and subtractions.

- *USAD8 and USADA8* on page 3-105

  Unsigned sum of absolute differences, and accumulate unsigned sum of absolute differences.

- *SSAT16 and USAT16* on page 3-107

  Parallel halfword saturating instructions.

There are also parallel unpacking instructions such as `SXT`, `SXTA`, `UXT`, and `UXTA`.

See also

Reference

- *SXT, SXTA, UXT, and UXTA* on page 3-112
- *Packing and unpacking instructions* on page 3-109.

3.7.1 Parallel add and subtract

Various byte-wise and halfword-wise additions and subtractions.

Syntax

`<prefix>op{cond} {Rd}, Rn, Rm`

where:

<prefix> is one of:

`S`	Signed arithmetic modulo 2^8 or 2^{16}. Sets APSR GE flags.
`Q`	Signed saturating arithmetic.
`SH`	Signed arithmetic, halving the results.
`U`	Unsigned arithmetic modulo 2^8 or 2^{16}. Sets APSR GE flags.
`UQ`	Unsigned saturating arithmetic.
`UH`	Unsigned arithmetic, halving the results.

op is one of:

`ADD8`	Byte-wise Addition
`ADD16`	Halfword-wise Addition.
`SUB8`	Byte-wise Subtraction.
`SUB16`	Halfword-wise Subtraction.
`ASX`	Exchange halfwords of *Rm*, then Add top halfwords and Subtract bottom halfwords.
`SAX`	Exchange halfwords of *Rm*, then Subtract top halfwords and Add bottom halfwords.

cond is an optional condition code.

Rd is the destination register.

Rm, Rn are the ARM registers holding the operands.

Operation

These instructions perform arithmetic operations separately on the bytes or halfwords of the operands. They perform two or four additions or subtractions, or one addition and one subtraction.

You can choose various kinds of arithmetic:

- Signed or unsigned arithmetic modulo 2^8 or 2^{16}. This sets the APSR GE flags.
- Signed saturating arithmetic to one of the signed ranges $-2^{15} \leq x \leq 2^{15} -1$ or $-2^7 \leq x \leq 2^7 -1$. The Q flag is not affected even if these operations saturate.
- Unsigned saturating arithmetic to one of the unsigned ranges $0 \leq x \leq 2^{16} -1$ or $0 \leq x \leq 2^8 -1$. The Q flag is not affected even if these operations saturate.
- Signed or unsigned arithmetic, halving the results. This cannot cause overflow.

Register restrictions

You cannot use PC for any register.

You can use SP in ARM instructions but these are deprecated in ARMv6T2 and above. You cannot use SP in Thumb instructions.

Condition flags

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

The `Q`, `SH`, `UQ` and `UH` prefix variants of these instructions do not change the flags.

The `S` and `U` prefix variants of these instructions set the GE flags in the APSR as follows:

- For byte-wise operations, the GE flags are used in the same way as the C (Carry) flag for 32-bit `SUB` and `ADD` instructions:

`GE[0]`	for bits[7:0] of the result
`GE[1]`	for bits[15:8] of the result
`GE[2]`	for bits[23:16] of the result
`GE[3]`	for bits[31:24] of the result.

- For halfword-wise operations, the GE flags are used in the same way as the C (Carry) flag for normal word-wise `SUB` and `ADD` instructions:

`GE[1:0]`	for bits[15:0] of the result
`GE[3:2]`	for bits[31:16] of the result.

You can use these flags to control a following `SEL` instruction.

Note

For halfword-wise operations, GE[1:0] are set or cleared together, and GE[3:2] are set or cleared together.

Architectures

These ARM instructions are available in ARMv6 and above.

These 32-bit Thumb instructions are available in ARMv6T2 and above. For the ARMv7-M architecture, they are only available in an ARMv7E-M implementation.

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

Examples

```
    SHADD8      r4, r3, r9
    USAXNE      r0, r0, r2
```

Incorrect examples

```
    QHADD       r2, r9, r3    ; No such instruction, should be QHADD8 or QHADD16
    SAX         r10, r8, r5   ; Must have a prefix.
```

See also

Reference

- *SEL* on page 3-68
- *Condition codes* on page 3-163.

3.7.2 USAD8 and USADA8

Unsigned Sum of Absolute Differences, and Accumulate unsigned sum of absolute differences.

Syntax

`USAD8{cond} {Rd}, Rn, Rm`

`USADA8{cond} Rd, Rn, Rm, Ra`

where:

cond	is an optional condition code.
Rd	is the destination register.
Rn	is the register holding the first operand.
Rm	is the register holding the second operand.
Ra	is the register holding the accumulate operand.

Operation

The `USAD8` instruction finds the four differences between the unsigned values in corresponding bytes of *Rn* and *Rm*. It adds the absolute values of the four differences, and saves the result to *Rd*.

The `USADA8` instruction adds the absolute values of the four differences to the value in *Ra*, and saves the result to *Rd*.

Register restrictions

You cannot use PC for any register.

You can use SP in ARM instructions but these are deprecated in ARMv6T2 and above. You cannot use SP in Thumb instructions.

Condition flags

These instructions do not alter any flags.

Architectures

These ARM instructions are available in ARMv6 and above.

These 32-bit Thumb instructions are available in ARMv6T2 and above. For the ARMv7-M architecture, they are only available in an ARMv7E-M implementation.

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

Examples

```
    USAD8       r2, r4, r6
    USADA8      r0, r3, r5, r2
    USADA8VS    r0, r4, r0, r1
```

Incorrect examples

```
    USADA8      r2, r4, r6      ; USADA8 requires four registers
    USADA16     r0, r4, r0, r1  ; no such instruction
```

See also

Reference

- *Condition codes* on page 3-163.

3.7.3 SSAT16 and USAT16

Parallel halfword Saturating instructions.

`SSAT16` saturates a signed value to a signed range.

`USAT16` saturates a signed value to an unsigned range.

Syntax

op{*cond*} *Rd*, #*sat*, *Rn*

where:

op	is one of: `SSAT16` Signed saturation. `USAT16` Unsigned saturation.
cond	is an optional condition code.
Rd	is the destination register.
sat	specifies the bit position to saturate to, and is in the range 1 to 16 for `SSAT16`, or 0 to 15 for `USAT16`.
Rn	is the register holding the operand.

Operation

Halfword-wise signed and unsigned saturation to any bit position.

The `SSAT16` instruction saturates each signed halfword to the signed range $-2^{sat-1} \leq x \leq 2^{sat-1} -1$.

The `USAT16` instruction saturates each signed halfword to the unsigned range $0 \leq x \leq 2^{sat} -1$.

Register restrictions

You cannot use PC for any register.

You can use SP in ARM instructions but these are deprecated in ARMv6T2 and above. You cannot use SP in Thumb instructions.

Condition flags

If saturation occurs on either halfword, these instructions set the Q flag. To read the state of the Q flag, use an `MRS` instruction.

Architectures

These ARM instructions are available in ARMv6 and above.

These 32-bit Thumb instructions are available in ARMv6T2 and above. For the ARMv7-M architecture, they are only available in an ARMv7E-M implementation.

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

Examples

```
    SSAT16  r7, #12, r7
    USAT16  r0, #7, r5
```

Incorrect examples

```
    SSAT16  r1, #16, r2, LSL #4 ; shifts not permitted with halfword saturations
```

See also

Reference

- *MRS* on page 3-137
- *Condition codes* on page 3-163.

3.8 Packing and unpacking instructions

This section contains the following subsections:

- *BFC and BFI* on page 3-110
  Bit Field Clear and Bit Field Insert.
- *SBFX and UBFX* on page 3-111
  Signed or Unsigned Bit Field extract.
- *SXT, SXTA, UXT, and UXTA* on page 3-112
  Sign Extend or Zero Extend instructions, with optional Add.
- *PKHBT and PKHTB* on page 3-114
  Halfword Packing instructions.

3.8.1 BFC and BFI

Bit Field Clear and Bit Field Insert. Clear adjacent bits in a register, or Insert adjacent bits from one register into another.

Syntax

```
BFC{cond} Rd, #lsb, #width
```

```
BFI{cond} Rd, Rn, #lsb, #width
```

where:

cond	is an optional condition code.
Rd	is the destination register.
Rn	is the source register.
lsb	is the least significant bit that is to be cleared or copied.
width	is the number of bits to be cleared or copied. *width* must not be 0, and (*width*+*lsb*) must be less than 32.

BFC

width bits in *Rd* are cleared, starting at *lsb*. Other bits in *Rd* are unchanged.

BFI

width bits in *Rd*, starting at *lsb*, are replaced by *width* bits from *Rn*, starting at bit[0]. Other bits in *Rd* are unchanged.

Register restrictions

You cannot use PC for any register.

You can use SP in ARM instructions but these are deprecated in ARMv6T2 and above. You cannot use SP in Thumb instructions.

Condition flags

These instructions do not change the flags.

Architectures

These ARM instructions are available in ARMv6T2 and above.

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

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

See also

Reference

- *Condition codes* on page 3-163.

3.8.2 SBFX and UBFX

Signed and Unsigned Bit Field Extract. Copies adjacent bits from one register into the least significant bits of a second register, and sign extends or zero extends to 32 bits.

Syntax

op{*cond*} *Rd*, *Rn*, #*lsb*, #*width*

where:

op	is either `SBFX` or `UBFX`.
cond	is an optional condition code.
Rd	is the destination register.
Rn	is the source register.
lsb	is the bit number of least significant bit in the bitfield, in the range 0 to 31.
width	is the width of the bitfield, in the range 1 to (32–*lsb*).

Register restrictions

You cannot use PC for any register.

You can use SP in ARM instructions but these are deprecated in ARMv6T2 and above. You cannot use SP in Thumb instructions.

Condition flags

These instructions do not alter any flags.

Architectures

These ARM instructions are available in ARMv6T2 and above.

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

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

See also

Reference

- *Condition codes* on page 3-163.

3.8.3 SXT, SXTA, UXT, and UXTA

Sign extend, Sign extend with Add, Zero extend, and Zero extend with Add.

Syntax

```
SXT<extend>{cond}  {Rd}, Rm  {,rotation}

SXTA<extend>{cond} {Rd}, Rn, Rm {,rotation}

UXT<extend>{cond}  {Rd}, Rm  {,rotation}

UXTA<extend>{cond} {Rd}, Rn, Rm {,rotation}
```

where:

<extend>	is one of:
	`B16` Extends two 8-bit values to two 16-bit values.
	`B` Extends an 8-bit value to a 32-bit value.
	`H` Extends a 16-bit value to a 32-bit value.
cond	is an optional condition code.
Rd	is the destination register.
Rn	is the register holding the number to add (`SXTA` and `UXTA` only).
Rm	is the register holding the value to extend.
rotation	is one of:
	`ROR #8` Value from *Rm* is rotated right 8 bits.
	`ROR #16` Value from *Rm* is rotated right 16 bits.
	`ROR #24` Value from *Rm* is rotated right 24 bits.
	If *rotation* is omitted, no rotation is performed.

Operation

These instructions do the following:

1. Rotate the value from *Rm* right by 0, 8, 16 or 24 bits.
2. Do one of the following to the value obtained:
   - Extract bits[7:0], sign or zero extend to 32 bits. If the instruction is extend and add, add the value from *Rn*.
   - Extract bits[15:0], sign or zero extend to 32 bits. If the instruction is extend and add, add the value from *Rn*.
   - Extract bits[23:16] and bits[7:0] and sign or zero extend them to 16 bits. If the instruction is extend and add, add them to bits[31:16] and bits[15:0] respectively of *Rn* to form bits[31:16] and bits[15:0] of the result.

Register restrictions

You cannot use PC for any register.

You can use SP in ARM instructions but these are deprecated in ARMv6T2 and above. You cannot use SP in Thumb instructions.

Condition flags

These instructions do not change the flags.

16-bit instructions

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

`SXTB Rd, Rm`	*Rd* and *Rm* must both be Lo registers.
`SXTH Rd, Rm`	*Rd* and *Rm* must both be Lo registers.
`UXTB Rd, Rm`	*Rd* and *Rm* must both be Lo registers.
`UXTH Rd, Rm`	*Rd* and *Rm* must both be Lo registers.

Architectures

These ARM instructions are available in ARMv6 and above.

These 32-bit Thumb instructions are available in ARMv6T2 and above. For the ARMv7-M architecture, they are only available in an ARMv7E-M implementation.

These 16-bit Thumb instructions are available in ARMv6 and above.

Examples

```
    SXTH        r3, r9, r4
    UXTAB16EQ   r0, r0, r4, ROR #16
```

Incorrect examples

```
    SXTH     r9, r3, r2, ROR #12 ; rotation must be by 0, 8, 16, or 24.
```

See also

Reference

- *Condition codes* on page 3-163.

3.8.4 PKHBT and PKHTB

Halfword Packing instructions.

Combine a halfword from one register with a halfword from another register. One of the operands can be shifted before extraction of the halfword.

Syntax

`PKHBT{cond} {Rd}, Rn, Rm{, LSL #leftshift}`

`PKHTB{cond} {Rd}, Rn, Rm{, ASR #rightshift}`

where:

`PKHBT`	Combines bits[15:0] of *Rn* with bits[31:16] of the shifted value from *Rm*.
`PKHTB`	Combines bits[31:16] of *Rn* with bits[15:0] of the shifted value from *Rm*.
cond	is an optional condition code.
Rd	is the destination register.
Rn	is the register holding the first operand.
Rm	is the register holding the first operand.
leftshift	is in the range 0 to 31.
rightshift	is in the range 1 to 32.

Register restrictions

You cannot use PC for any register.

You can use SP in ARM instructions but these are deprecated in ARMv6T2 and above. You cannot use SP in Thumb instructions.

Condition flags

These instructions do not change the flags.

Architectures

These ARM instructions are available in ARMv6 and above.

These 32-bit Thumb instructions are available in ARMv6T2 and above. For the ARMv7-M architecture, they are only available in an ARMv7E-M implementation.

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

Examples

```
    PKHBT   r0, r3, r5          ; combine the bottom halfword of R3 with
                                ; the top halfword of R5
    PKHBT   r0, r3, r5, LSL #16 ; combine the bottom halfword of R3 with
                                ; the bottom halfword of R5
    PKHTB   r0, r3, r5, ASR #16 ; combine the top halfword of R3 with
                                ; the top halfword of R5
```

You can also scale the second operand by using different values of shift.

Incorrect examples

```
    PKHBTEQ r4, r5, r1, ASR #8  ; ASR not permitted with PKHBT
```

See also

Reference

- *Condition codes* on page 3-163.

3.9 Branch and control instructions

This section contains the following subsections:

- *B, BL, BX, BLX, and BXJ* on page 3-117

  Branch, Branch with Link, Branch and exchange instruction set, Branch with Link and exchange instruction set, Branch and change instruction set to Jazelle.
- *IT* on page 3-120

  If-Then. `IT` makes up to four following instructions conditional, with either the same condition, or some with one condition and others with the inverse condition. `IT` is available only in 32-bit Thumb.
- *CBZ and CBNZ* on page 3-123

  Compare against zero and branch. These instructions are available only in 32-bit Thumb.
- *TBB and TBH* on page 3-124

  Table Branch Byte or Halfword. These instructions are available only in 32-bit Thumb.

3.9.1 B, BL, BX, BLX, and BXJ

Branch, Branch with Link, Branch and exchange instruction set, Branch with Link and exchange instruction set, Branch and change to Jazelle state.

Syntax

`op1{cond}{.W} label`

`op2{cond} Rm`

where:

op1 is one of:

`B`	Branch.
`BL`	Branch with link.
`BLX`	Branch with link, and exchange instruction set.

op2 is one of:

`BX`	Branch and exchange instruction set.
`BLX`	Branch with link, and exchange instruction set.
`BXJ`	Branch, and change to Jazelle execution.

cond is an optional condition code. *cond* is not available on all forms of this instruction.

`.W` is an optional instruction width specifier to force the use of a 32-bit `B` instruction in 32-bit Thumb.

label is a PC-relative expression.

Rm is a register containing an address to branch to.

Operation

All these instructions cause a branch to *label*, or to the address contained in *Rm*. In addition:

- The `BL` and `BLX` instructions copy the address of the next instruction into LR (`R14`, the link register).
- The `BX` and `BLX` instructions can change the processor state from ARM to Thumb, or from Thumb to ARM.

  `BLX` *label* always changes the state.

  `BX` *Rm* and `BLX` *Rm* derive the target state from bit[0] of *Rm*:
  - if bit[0] of *Rm* is 0, the processor changes to, or remains in, ARM state
  - if bit[0] of *Rm* is 1, the processor changes to, or remains in, Thumb state.
- The `BXJ` instruction changes the processor state to Jazelle.

Instruction availability and branch ranges

Table 3-9 shows the instructions that are available in ARM and Thumb state. Instructions that are not shown in this table are not available. Notes in brackets show the first architecture version where the instruction is available.

Table 3-9 Branch instruction availability and range

Instruction	ARM		16-bit Thumb		32-bit Thumb	
`B label`	±32MB	(All)	±2KB	(All T)	±16MB[a]	(All T2)
`B{cond} label`	±32MB	(All)	–252 to +258	(All T)	±1MB[a]	(All T2)
`BL label`	±32MB	(All)	±4MB [b]	(All T)	±16MB	(All T2)
`BL{cond} label`	±32MB	(All)	-		-	-
`BX Rm` [c]	Available	(4T, 5)	Available	(All T)	Use 16-bit	(All T2)
`BX{cond} Rm` [c]	Available	(4T, 5)	-		-	-
`BLX label`	±32MB	(5)	±4MB [b]	(5T)	±16MB	(All T2 except ARMv7-M)
`BLX Rm`	Available	(5)	Available	(5T)	Use 16-bit	(All T2)
`BLX{cond} Rm`	Available	(5)	-		-	-
`BXJ Rm`	Available	(5J, 6)	-		Available	(All T2 except ARMv7-M)
`BXJ{cond} Rm`	Available	(5J, 6)	-		-	-

a. Use `.W` to instruct the assembler to use this 32-bit instruction.

b. This is an instruction pair.

c. The assembler accepts `BX{`*`cond`*`}` *`Rm`* for code assembled for ARMv4 and converts it to `MOV{`*`cond`*`} PC,` *`Rm`* at link time, unless objects targeted for ARMv4T are present.

Extending branch ranges

Machine-level `B` and `BL` instructions have restricted ranges from the address of the current instruction. However, you can use these instructions even if *`label`* is out of range. Often you do not know where the linker places *`label`*. When necessary, the linker adds code to enable longer branches. The added code is called a *veneer*.

B in Thumb

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

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

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

BX, BLX, and BXJ in ThumbEE

These instructions can be used as branches in ThumbEE code, but cannot be used to change state. You cannot use the *`op`*`{`*`cond`*`}` *`label`* form of these instructions in ThumbEE. In the register form, bit[0] of *`Rm`* must be 1, and execution continues at the target address in ThumbEE state.

Note

BXJ behaves like BX in ThumbEE.

Register restrictions

You can use PC for Rm in the ARM BX instruction, but this is deprecated in ARMv6T2 and above. You cannot use PC in other ARM instructions.

You can use PC for Rm in the Thumb BX instruction. You cannot use PC in other Thumb instructions.

You can use SP for Rm in these ARM instructions but these are deprecated in ARMv6T2 and above.

You can use SP for Rm in the Thumb BX and BLX instructions, but these are deprecated. You cannot use SP in the other Thumb instructions.

Condition flags

These instructions do not change the flags.

Architectures

See Table 3-9 on page 3-118 for details of availability of these instructions in each architecture.

Examples

```
B       loopA
BLE     ng+8
BL      subC
BLLT    rtX
BEQ     {PC}+4  ; #0x8004
```

See also

Concepts

Using the Assembler:

- *Register-relative and PC-relative expressions* on page 8-7.

Using the Linker:

- Chapter 4 *Image structure and generation*.

Reference

- *Condition codes* on page 3-163.

3.9.2 IT

The `IT` (If-Then) instruction makes up to four following instructions (the *IT block*) conditional. The conditions can be all the same, or some of them can be the logical inverse of the others.

Syntax

`IT{x{y{z}}} {cond}`

where:

x	specifies the condition switch for the second instruction in the IT block.
y	specifies the condition switch for the third instruction in the IT block.
z	specifies the condition switch for the fourth instruction in the IT block.
cond	specifies the condition for the first instruction in the IT block.

The condition switch for the second, third and fourth instruction in the IT block can be either:

`T`	Then. Applies the condition *cond* to the instruction.
`E`	Else. Applies the inverse condition of *cond* to the instruction.

Usage

The instructions (including branches) in the IT block, except the `BKPT` instruction, must specify the condition in the {*cond*} part of their syntax.

You do not need to write `IT` instructions in your code, because the assembler generates them for you automatically according to the conditions specified on the following instructions. However, if you do write `IT` instructions, the assembler validates the conditions specified in the `IT` instructions against the conditions specified in the following instructions.

Writing the `IT` instructions ensures that you consider the placing of conditional instructions, and the choice of conditions, in the design of your code.

When assembling to ARM code, the assembler performs the same checks, but does not generate any `IT` instructions.

With the exception of `CMP`, `CMN`, and `TST`, the 16-bit instructions that normally affect the condition code flags, do not affect them when used inside an IT block.

A `BKPT` instruction in an IT block is always executed, so it does not need a condition in the {*cond*} part of its syntax. The IT block continues from the next instruction.

Note

You can use an `IT` block for unconditional instructions by using the `AL` condition.

Conditional branches inside an IT block have a longer branch range than those outside the IT block.

Restrictions

The following instructions are not permitted in an IT block:

- `IT`
- `CBZ` and `CBNZ`
- `TBB` and `TBH`
- `CPS`, `CPSID` and `CPSIE`
- `SETEND`.

Other restrictions when using an IT block are:

- A branch or any instruction that modifies the PC is only permitted in an IT block if it is the last instruction in the block.
- You cannot branch to any instruction in an IT block, unless when returning from an exception handler.
- You cannot use any assembler directives in an IT block.

——— **Note** ———

The assembler shows a diagnostic message when any of these instructions are used in an `IT` block.

———————————

Condition flags

This instruction does not change the flags.

Exceptions

Exceptions can occur between an `IT` instruction and the corresponding IT block, or within an IT block. This exception results in entry to the appropriate exception handler, with suitable return information in LR and SPSR.

Instructions designed for use as exception returns can be used as normal to return from the exception, and execution of the IT block resumes correctly. This is the only way that a PC-modifying instruction can branch to an instruction in an IT block.

Architectures

This 16-bit Thumb instruction is available in ARMv6T2 and above.

In ARM code, `IT` is a pseudo-instruction that does not generate any code.

There is no 32-bit version of this instruction.

Example

```
    ITTE   NE        ; IT can be omitted
    ANDNE  r0,r0,r1  ; 16-bit AND, not ANDS
    ADDSNE r2,r2,#1  ; 32-bit ADDS (16-bit ADDS does not set flags in IT block)
    MOVEQ  r2,r3     ; 16-bit MOV

    ITT    AL        ; emit 2 non-flag setting 16-bit instructions
    ADDAL  r0,r0,r1  ; 16-bit ADD, not ADDS
    SUBAL  r2,r2,#1  ; 16-bit SUB, not SUB
    ADD    r0,r0,r1  ; expands into 32-bit ADD, and is not in IT block

    ITT    EQ
    MOVEQ  r0,r1
    BEQ    dloop     ; branch at end of IT block is permitted

    ITT    EQ
    MOVEQ  r0,r1
    BKPT   #1        ; BKPT always executes
    ADDEQ  r0,r0,#1
```

Incorrect example

```
    IT      NE
    ADD     r0,r0,r1  ; syntax error: no condition code used in IT block
```

3.9.3 CBZ and CBNZ

Compare and Branch on Zero, Compare and Branch on Non-Zero.

Syntax

`CBZ` *`Rn`*`,` *`label`*

`CBNZ` *`Rn`*`,` *`label`*

where:

`Rn`	is the register holding the operand.
`label`	is the branch destination.

Usage

You can use the `CBZ` or `CBNZ` instructions to avoid changing the condition code flags and to reduce the number of instructions.

Except that it does not change the condition code flags, `CBZ Rn, label` is equivalent to:

```
    CMP     Rn, #0
    BEQ     label
```

Except that it does not change the condition code flags, `CBNZ Rn, label` is equivalent to:

```
    CMP     Rn, #0
    BNE     label
```

Restrictions

The branch destination must be within 4 to 130 bytes after the instruction and in the same execution state.

These instructions must not be used inside an IT block.

Condition flags

These instructions do not change the flags.

Architectures

These 16-bit Thumb instructions are available in ARMv6T2 and above.

There are no ARM or 32-bit Thumb versions of these instructions.

3.9.4 TBB and TBH

Table Branch Byte and Table Branch Halfword.

Syntax

`TBB [Rn, Rm]`

`TBH [Rn, Rm, LSL #1]`

where:

Rn	is the base register. This contains the address of the table of branch lengths. *Rn* must not be SP. If PC is specified for *Rn*, the value used is the address of the instruction plus 4.
Rm	is the index register. This contains an index into the table. *Rm* must not be PC or SP.

Operation

These instructions cause a PC-relative forward branch using a table of single byte offsets (`TBB`) or halfword offsets (`TBH`). *Rn* provides a pointer to the table, and *Rm* supplies an index into the table. The branch length is twice the value of the byte (`TBB`) or the halfword (`TBH`) returned from the table. The target of the branch table must be in the same execution state.

Notes

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

Architectures

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

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

3.10 Coprocessor instructions

This section contains the following subsections:

- *CDP and CDP2* on page 3-126

  Coprocessor Data oPerations.

- *MCR, MCR2, MCRR, and MCRR2* on page 3-127

  Move to Coprocessor from ARM Register or Registers, possibly with coprocessor operations.

- *MRC, MRC2, MRRC and MRRC2* on page 3-128

  Move to ARM Register or Registers from Coprocessor, possibly with coprocessor operations.

- *MSR* on page 3-129

  Move to system coprocessor from ARM register.

- *MRS* on page 3-130

  Move to ARM register from system coprocessor.

- *SYS* on page 3-131

  Execute system coprocessor instruction.

- *LDC, LDC2, STC, and STC2* on page 3-132

  Transfer data between memory and Coprocessor.

Note

A coprocessor instruction causes an Undefined Instruction exception if the specified coprocessor is not present, or if it is not enabled.

This section does not describe VFP or Wireless MMX Technology instructions. XScale-specific instructions are described later in this document.

See also

Reference

- Chapter 4 *NEON and VFP Programming*
- Chapter 5 *Wireless MMX Technology Instructions*
- *Miscellaneous instructions* on page 3-134.

3.10.1 CDP and CDP2

Coprocessor data operations.

Syntax

op{*cond*} *coproc*, #*opcode1*, *CRd*, *CRn*, *CRm*{, #*opcode2*}

where:

op	is either `CDP` or `CDP2`.
cond	is an optional condition code. In ARM code, *cond* is not permitted for `CDP2`.
coproc	is the name of the coprocessor the instruction is for. The standard name is `p`*n*, where *n* is an integer in the range 0 to 15.
opcode1	is a 4-bit coprocessor-specific opcode.
opcode2	is an optional 3-bit coprocessor-specific opcode.
CRd, *CRn*, *CRm*	are coprocessor registers.

Usage

The use of these instructions depends on the coprocessor. See the coprocessor documentation for details.

Architectures

The `CDP` ARM instruction is available in all versions of the ARM architecture.

The `CDP2` ARM instruction is available in ARMv5T and above.

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

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

See also

Reference

- *Condition codes* on page 3-163.

3.10.2 MCR, MCR2, MCRR, and MCRR2

Move to Coprocessor from ARM Register or Registers. Depending on the coprocessor, you might be able to specify various operations in addition.

Syntax

op1{*cond*} *coproc*, #*opcode1*, *Rt*, *CRn*, *CRm*{, #*opcode2*}

op2{*cond*} *coproc*, #*opcode3*, *Rt*, *Rt2*, *CRm*

where:

op1	is either `MCR` or `MCR2`.
op2	is either `MCRR` or `MCRR2`.
cond	is an optional condition code. In ARM code, *cond* is not permitted for `MCR2` or `MCRR2`.
coproc	is the name of the coprocessor the instruction is for. The standard name is `p`*n*, where *n* is an integer in the range 0 to 15.
opcode1	is a 3-bit coprocessor-specific opcode.
opcode2	is an optional 3-bit coprocessor-specific opcode.
opcode3	is a 4-bit coprocessor-specific opcode.
Rt, *Rt2*	are ARM source registers. `R`*t* and `R`*t2* must not be PC.
CRn, *CRm*	are coprocessor registers.

Usage

The use of these instructions depends on the coprocessor. See the coprocessor documentation for details.

Architectures

The `MCR` ARM instruction is available in all versions of the ARM architecture.

The `MCR2` ARM instruction is available in ARMv5T and above.

The `MCRR` ARM instruction is available in ARMv6 and above, and E variants of ARMv5T.

The `MCRR2` ARM instruction is available in ARMv6 and above.

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

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

See also

Reference

- *Condition codes* on page 3-163.

3.10.3 MRC, MRC2, MRRC and MRRC2

Move to ARM Register or Registers from Coprocessor.

Depending on the coprocessor, you might be able to specify various operations in addition.

Syntax

op1{*cond*} *coproc*, #*opcode1*, *Rt*, *CRn*, *CRm*{, #*opcode2*}

op2{*cond*} *coproc*, #*opcode3*, *Rt*, *Rt2*, *CRm*

where:

op1	is either `MRC` or `MRC2`.
op2	is either `MRRC` or `MRRC2`.
cond	is an optional condition code. In ARM code, *cond* is not permitted for `MRC2` or `MRRC2`.
coproc	is the name of the coprocessor the instruction is for. The standard name is `p`*n*, where *n* is an integer in the range 0 to 15.
opcode1	is a 3-bit coprocessor-specific opcode.
opcode2	is an optional 3-bit coprocessor-specific opcode.
opcode3	is a 4-bit coprocessor-specific opcode.
Rt, *Rt2*	are ARM destination registers. *Rt* and *Rt2* must not be PC. In `MRC` and `MRC2`, *Rt* can be `APSR_nzcv`. This means that the coprocessor executes an instruction that changes the value of the condition code flags in the APSR.
CRn, *CRm*	are coprocessor registers.

Usage

The use of these instructions depends on the coprocessor. See the coprocessor documentation for details.

Architectures

The `MRC` ARM instruction is available in all versions of the ARM architecture.

The `MRC2` ARM instruction is available in ARMv5T and above.

The `MRRC` ARM instruction is available in ARMv6 and above, and E variants of ARMv5T.

The `MRRC2` ARM instruction is available in ARMv6 and above.

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

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

See also

Reference

- *Condition codes* on page 3-163.

3.10.4 MSR

Move to system coprocessor register from ARM register.

Syntax

`MSR{cond} coproc_register, Rn`

where:

cond	is an optional condition code.
coproc_register	is the name of the coprocessor register.
Rn	is the ARM source register. *Rn* must not be PC.

Usage

You can use this instruction to write to any CP14 or CP15 coprocessor writable register. A complete list of the applicable coprocessor register names is in the *ARMv7-AR Architecture Reference Manual*. For example:

```
MSR SCTLR, R1 ; writes the contents of R1 into the CP15 coprocessor register
              ; SCTLR
```

Architectures

This `MSR` ARM instruction is available in ARMv7-A and ARMv7-R.

This `MSR` 32-bit Thumb instruction is available in ARMv7-A and ARMv7-R.

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

See also

Reference

- *SYS* on page 3-131
- *MRS* on page 3-130
- *MRS* on page 3-137
- *MSR* on page 3-139
- *Condition codes* on page 3-163
- *ARM Architecture Reference Manual*, `http://infocenter.arm.com/help/topic/com.arm.doc.subset.architecture.reference/index.html`.

3.10.5 MRS

Move to ARM register from system coprocessor register.

Syntax

`MRS{cond} Rn, coproc_register`

`MRS{cond} APSR_nzcv, special_register`

where:

cond	is an optional condition code.
coproc_register	is the name of the coprocessor register.
special_register	is the name of the coprocessor register that can be written to APSR_nzcv. This is only possible for the coprocessor register DBGDSCRint.
Rn	is the ARM destination register. `Rn` must not be PC.

Usage

You can use this instruction to read CP14 or CP15 coprocessor registers, with the exception of write-only registers. A complete list of the applicable coprocessor register names is in the *ARMv7-AR Architecture Reference Manual*. For example:

```
MRS R1, SCTLR ; writes the contents of the CP15 coprocessor register SCTLR
              ; into R1
```

Architectures

This `MRS` ARM instruction is available in ARMv7-A and ARMv7-R.

This `MRS` 32-bit Thumb instruction is available in ARMv7-A and ARMv7-R.

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

See also

Reference

- *Condition codes* on page 3-163
- *MSR* on page 3-129
- *MSR* on page 3-139
- *MRS* on page 3-137
- *ARM Architecture Reference Manual*, `http://infocenter.arm.com/help/topic/com.arm.doc.subset.architecture.reference/index.html`.

3.10.6 SYS

Execute system coprocessor instruction.

Syntax

`SYS{cond} instruction{, Rn}`

where:

cond is an optional condition code.

instruction

is the coprocessor instruction to execute.

Rn is an operand to the instruction. For instructions that take an argument, *Rn* is compulsory. For instructions that do not take an argument, *Rn* is optional and if it is not specified, `R0` is used. R*n* must not be PC.

Usage

You can use this instruction to execute special coprocessor instructions such as cache, branch predictor, and TLB operations. The instructions operate by writing to special write-only coprocessor registers. The instruction names are the same as the write-only coprocessor register names and are listed in the *ARMv7-AR Architecture Reference Manual*. For example:

```
    SYS ICIALLUIS ; invalidates all instruction caches Inner Shareable to Point
                  ; of Unification and also flushes branch target cache.
```

Architectures

The `SYS` ARM instruction is available in ARMv7-A and ARMv7-R.

The `SYS` 32-bit Thumb instruction is available in ARMv7-A and ARMv7-R.

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

See also

Reference

- *Condition codes* on page 3-163.

3.10.7 LDC, LDC2, STC, and STC2

Transfer Data between memory and Coprocessor.

Syntax

```
op{L}{cond} coproc, CRd, [Rn]

op{L}{cond} coproc, CRd, [Rn, #{-}offset]   ; offset addressing

op{L}{cond} coproc, CRd, [Rn, #{-}offset]!  ; pre-index addressing

op{L}{cond} coproc, CRd, [Rn], #{-}offset    ; post-index addressing

op{L}{cond} coproc, CRd, label
```

where:

op	is one of LDC, LDC2, STC, or STC2.
cond	is an optional condition code. In ARM code, *cond* is not permitted for LDC2 or STC2.
L	is an optional suffix specifying a long transfer.
coproc	is the name of the coprocessor the instruction is for. The standard name is p*n*, where *n* is an integer in the range 0 to 15.
CRd	is the coprocessor register to load or store.
Rn	is the register on which the memory address is based. If PC is specified, the value used is the address of the current instruction plus eight.
-	is an optional minus sign. If - is present, the offset is subtracted from *Rn*. Otherwise, the offset is added to *Rn*.
offset	is an expression evaluating to a multiple of 4, in the range 0 to 1020.
!	is an optional suffix. If ! is present, the address including the offset is written back into *Rn*.
label	is a word-aligned PC-relative expression. *label* must be within 1020 bytes of the current instruction.

Usage

The use of these instructions depends on the coprocessor. See the coprocessor documentation for details.

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

Architectures

LDC and STC are available in all versions of the ARM architecture.

LDC2 and STC2 are available in ARMv5T and above.

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

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

Register restrictions

You cannot use PC for `Rn` in the pre-index and post-index instructions. These are the forms that write back to `Rn`.

You cannot use PC for `Rn` in Thumb `STC` and `STC2` instructions.

ARM `STC` and `STC2` instructions that use the label syntax, or where `Rn` is PC, are deprecated in ARMv6T2 and above.

See also

Concepts

Using the Assembler:

- *Register-relative and PC-relative expressions* on page 8-7.

Reference

- *Condition codes* on page 3-163.

3.11 Miscellaneous instructions

This section contains the following subsections:

- *BKPT* on page 3-135
  Breakpoint.
- *SVC* on page 3-136
  Supervisor Call (formerly `SWI`).
- *MRS* on page 3-137
  Move the contents of the CPSR or SPSR to a general-purpose register.
- *MSR* on page 3-139
  Load specified fields of the CPSR or SPSR with an immediate value, or from the contents of a general-purpose register.
- *CPS* on page 3-141
  Change Processor State.
- *SMC* on page 3-142
  Secure Monitor Call (formerly `SMI`).
- *SETEND* on page 3-143
  Set the Endianness bit in the CPSR.
- *NOP* on page 3-144
  No Operation.
- *SEV, WFE, WFI, and YIELD* on page 3-145
  Set Event, Wait For Event, Wait for Interrupt, and Yield hint instructions.
- *DBG* on page 3-147
  Debug.
- *DMB, DSB, and ISB* on page 3-148
  Data Memory Barrier, Data Synchronization Barrier, and Instruction Synchronization Barrier hint instructions.
- *MAR and MRA* on page 3-150
  Transfer between two general-purpose registers and a 40-bit internal accumulator (XScale coprocessor 0 instructions).

3.11.1 BKPT

Breakpoint.

Syntax

```
BKPT #imm
```

where:

imm	is an expression evaluating to an integer in the range: • 0-65535 (a 16-bit value) in an ARM instruction • 0-255 (an 8-bit value) in a 16-bit Thumb instruction.

Usage

The `BKPT` instruction causes the processor to enter Debug state. Debug tools can use this to investigate system state when the instruction at a particular address is reached.

In both ARM state and Thumb state, *imm* is ignored by the ARM hardware. However, a debugger can use it to store additional information about the breakpoint.

`BKPT` is an unconditional instruction. It must not have a condition code in ARM code. In Thumb code, the `BKPT` instruction does not need a condition code suffix because `BKPT` always executes irrespective of its condition code suffix.

Architectures

This ARM instruction is available in ARMv5T and above.

This 16-bit Thumb instruction is available in ARMv5T and above.

There is no 32-bit Thumb version of this instruction.

3.11.2 SVC

SuperVisor Call.

Syntax

`SVC{cond} #imm`

where:

cond	is an optional condition code.
imm	is an expression evaluating to an integer in the range: • 0 to 2^{24}–1 (a 24-bit value) in an ARM instruction • 0-255 (an 8-bit value) in a 16-bit Thumb instruction.

Usage

The `SVC` instruction causes an exception. This means that the processor mode changes to Supervisor, the CPSR is saved to the Supervisor mode SPSR, and execution branches to the SVC vector.

imm is ignored by the processor. However, it can be retrieved by the exception handler to determine what service is being requested.

Note

`SVC` was called `SWI` in earlier versions of the ARM assembly language. `SWI` instructions disassemble to `SVC`, with a comment to say that this was formerly `SWI`.

Condition flags

This instruction does not change the flags.

Architectures

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

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

There is no 32-bit Thumb version of this instruction.

See also

Concepts

Developing Software for ARM Processors:

- Chapter 6 *Handling Processor Exceptions*.

Reference

- *Condition codes* on page 3-163.

3.11.3 MRS

Move the contents of a PSR to a general-purpose register.

Syntax

MRS{*cond*} *Rd*, *psr*

where:

cond	is an optional condition code.
Rd	is the destination register.
psr	is one of:

APSR	on any processor, in any mode.
CPSR	deprecated synonym for APSR and for use in Debug state, on any processor except ARMv7-M and ARMv6-M.
SPSR	on any processor except ARMv7-M and ARMv6-M, in privileged software execution only.
Mpsr	on ARMv7-M and ARMv6-M processors only.

Mpsr	can be any of: IPSR, EPSR, IEPSR, IAPSR, EAPSR, MSP, PSP, XPSR, PRIMASK, BASEPRI, BASEPRI_MAX, FAULTMASK, or CONTROL.

Usage

Use MRS in combination with MSR as part of a read-modify-write sequence for updating a PSR, for example to change processor mode, or to clear the Q flag.

In process swap code, the programmers' model state of the process being swapped out must be saved, including relevant PSR contents. Similarly, the state of the process being swapped in must also be restored. These operations make use of MRS/store and load/MSR instruction sequences.

SPSR

You must not attempt to access the SPSR when the processor is in User or System mode. This is your responsibility. The assembler cannot warn you about this, because it has no information about the processor mode at execution time.

If you attempt to access the SPSR when the processor is in User or System mode, the result is UNPREDICTABLE.

CPSR

The CPSR endianness bit (E) can be read in any privileged software execution.

The CPSR execution state bits, other than the E bit, can only be read when the processor is in Debug state, halting debug-mode. Otherwise, the execution state bits in the CPSR read as zero.

The condition flags can be read in any mode on any processor. Use APSR if you are only interested in accessing the condition code flags in User mode.

Register restrictions

You cannot use PC in ARM instructions. You can use SP for `Rd` in ARM instructions but this is deprecated in ARMv6T2 and above.

You cannot use PC or SP in Thumb instructions.

Condition flags

This instruction does not change the flags.

Architectures

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

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

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

See also

Concepts

Using the Assembler:

- *Current Program Status Register* on page 3-20.

Reference

- *MSR* on page 3-139
- *MSR* on page 3-129
- *MRS* on page 3-130
- *Condition codes* on page 3-163.

3.11.4 MSR

Load an immediate value, or the contents of a general-purpose register, into specified fields of a *Program Status Register* (PSR).

Syntax

`MSR{cond} APSR_flags, Rm`

where:

cond	is an optional condition code.
flags	specifies the APSR flags to be moved. *flags* can be one or more of: `nzcvq` ALU flags field mask, PSR[31:27] (User mode) `g` SIMD GE flags field mask, PSR[19:16] (User mode).
Rm	is the source register. R*m* must not be PC.

Syntax (except ARMv7-M and ARMv6-M)

You can also use the following syntax on architectures other than ARMv7 and ARMv6M.

`MSR{cond} APSR_flags, #constant`

`MSR{cond} psr_fields, #constant`

`MSR{cond} psr_fields, Rm`

where:

cond	is an optional condition code.
flags	specifies the APSR flags to be moved. *flags* can be one or more of: `nzcvq` ALU flags field mask, PSR[31:27] (User mode) `g` SIMD GE flags field mask, PSR[19:16] (User mode).
constant	is an expression evaluating to a numeric value. The value must correspond to an 8-bit pattern rotated by an even number of bits within a 32-bit word. Not available in Thumb.
Rm	is the source register. R*m* must not be PC.
psr	is one of: `CPSR` for use in Debug state, also deprecated synonym for APSR `SPSR` on any processor, in privileged software execution only.
fields	specifies the SPSR or CPSR fields to be moved. *fields* can be one or more of: `c` control field mask byte, PSR[7:0] (privileged software execution) `x` extension field mask byte, PSR[15:8] (privileged software execution) `s` status field mask byte, PSR[23:16] (privileged software execution) `f` flags field mask byte, PSR[31:24] (privileged software execution).

Syntax (ARMv7-M and ARMv6-M only)

You can also use the following syntax on ARMv7 and ARMv6M.

`MSR{cond} psr, Rm`

where:

cond	is an optional condition code.
Rm	is the source register. *Rm* must not be PC.
psr	can be any of: `APSR`, `IPSR`, `EPSR`, `IEPSR`, `IAPSR`, `EAPSR`, `XPSR`, `MSP`, `PSP`, `PRIMASK`, `BASEPRI`, `BASEPRI_MAX`, `FAULTMASK`, or `CONTROL`.

Usage

In User mode:

- Use APSR to access condition flags, Q, or GE bits.
- Writes to unallocated, privileged or execution state bits in the CPSR are ignored. This ensures that User mode programs cannot change to privileged software execution.

If you access the SPSR when in User or System mode, the result is UNPREDICTABLE.

Register restrictions

You cannot use PC in ARM instructions. You can use SP for *Rm* in ARM instructions but these are deprecated in ARMv6T2 and above.

You cannot use PC or SP in Thumb instructions.

Condition flags

This instruction updates the flags explicitly if the `APSR_nzcvq` or `CPSR_f` field is specified.

Architectures

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

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

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

See also

Reference

- *MRS* on page 3-137
- *MRS* on page 3-130
- *MSR* on page 3-129
- *Condition codes* on page 3-163.

3.11.5 CPS

`CPS` (Change Processor State) changes one or more of the mode, A, I, and F bits in the CPSR, without changing the other CPSR bits.

`CPS` is only permitted in privileged software execution, and has no effect in User mode.

`CPS` cannot be conditional, and is not permitted in an IT block.

Syntax

`CPS`*`effect iflags`*`{, #`*`mode`*`}`

`CPS #`*`mode`*

where:

effect	is one of:	
	`IE`	Interrupt or abort enable.
	`ID`	Interrupt or abort disable.
iflags	is a sequence of one or more of:	
	`a`	Enables or disables imprecise aborts.
	`i`	Enables or disables IRQ interrupts.
	`f`	Enables or disables FIQ interrupts.
mode	specifies the number of the mode to change to.	

Condition flags

This instruction does not change the condition flags.

16-bit instructions

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

- `CPSIE` *iflags*
- `CPSID` *iflags*

You cannot specify a mode change in a 16-bit Thumb instruction.

Architectures

This ARM instruction is available in ARMv6 and above.

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

This 16-bit Thumb instruction is available in T variants of ARMv6 and above.

Examples

```
    CPSIE if      ; enable interrupts and fast interrupts
    CPSID A       ; disable imprecise aborts
    CPSID ai, #17 ; disable imprecise aborts and interrupts, and enter FIQ mode
    CPS #16       ; enter User mode
```

3.11.6 SMC

Secure Monitor Call.

Syntax

`SMC{cond} #imm4`

where:

cond	is an optional condition code.
imm4	is a 4-bit immediate value. This is ignored by the ARM processor, but can be used by the SMC exception handler to determine what service is being requested.

Note

`SMC` was called `SMI` in earlier versions of the ARM assembly language. `SMI` instructions disassemble to `SMC`, with a comment to say that this was formerly `SMI`.

Architectures

This ARM instruction is available in implementations of ARMv6 and above, if they have the Security Extensions.

This 32-bit Thumb instruction is available in implementations of ARMv6T2 and above, if they have the Security Extensions.

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

See also

Reference

- *Condition codes* on page 3-163
- *ARM Architecture Reference Manual*, `http://infocenter.arm.com/help/topic/com.arm.doc.subset.architecture.reference/index.html`.

3.11.7 SETEND

Set the endianness bit in the CPSR, without affecting any other bits in the CPSR.

`SETEND` cannot be conditional, and is not permitted in an IT block.

Syntax

`SETEND` *`specifier`*

where:

`specifier`	is one of:	
	`BE`	Big-endian.
	`LE`	Little-endian.

Usage

Use `SETEND` to access data of different endianness, for example, to access several big-endian DMA-formatted data fields from an otherwise little-endian application.

Architectures

This ARM instruction is available in ARMv6 and above.

This 16-bit Thumb instruction is available in T variants of ARMv6 and above, except the ARMv6-M and ARMv7-M architectures.

There is no 32-bit Thumb version of this instruction.

Example

```
    SETEND  BE          ; Set the CPSR E bit for big-endian accesses
    LDR     r0, [r2, #header]
    LDR     r1, [r2, #CRC32]
    SETEND  le          ; Set the CPSR E bit for little-endian accesses for the
                        ; rest of the application
```

3.11.8 NOP

No Operation.

Syntax

`NOP{cond}`

where:

cond	is an optional condition code.

Usage

`NOP` does nothing. If `NOP` is not implemented as a specific instruction on your target architecture, the assembler treats it as a pseudo-instruction and generates an alternative instruction that does nothing, such as `MOV r0, r0` (ARM) or `MOV r8, r8` (Thumb).

`NOP` is not necessarily a time-consuming `NOP`. The processor might remove it from the pipeline before it reaches the execution stage.

You can use `NOP` for padding, for example to place the following instruction on a 64-bit boundary in ARM, or a 32-bit boundary in Thumb.

Architectures

This ARM instructions are available in ARMv6K and above.

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

This 16-bit Thumb instructions are available in ARMv6T2 and above.

`NOP` is available on all other ARM and Thumb architectures as a pseudo-instruction.

See also

Reference

- *Condition codes* on page 3-163.

3.11.9 SEV, WFE, WFI, and YIELD

Set Event, Wait For Event, Wait for Interrupt, and Yield.

Syntax

SEV{*cond*}

WFE{*cond*}

WFI{*cond*}

YIELD{*cond*}

where:

cond	is an optional condition code.

Usage

These are hint instructions. It is optional whether they are implemented or not. If any one of them is not implemented, it executes as a `NOP`. The assembler produces a diagnostic message if the instruction executes as a `NOP` on the target.

`SEV`, `WFE`, `WFI`, and `YIELD` execute as `NOP` instructions in ARMv6T2.

SEV

`SEV` causes an event to be signaled to all cores within a multiprocessor system. If `SEV` is implemented, `WFE` must also be implemented.

WFE

If the Event Register is not set, `WFE` suspends execution until one of the following events occurs:

- an IRQ interrupt, unless masked by the CPSR I-bit
- an FIQ interrupt, unless masked by the CPSR F-bit
- an Imprecise Data abort, unless masked by the CPSR A-bit
- a Debug Entry request, if Debug is enabled
- an Event signaled by another processor using the `SEV` instruction.

If the Event Register is set, `WFE` clears it and returns immediately.

If `WFE` is implemented, `SEV` must also be implemented.

WFI

`WFI` suspends execution until one of the following events occurs:

- an IRQ interrupt, regardless of the CPSR I-bit
- an FIQ interrupt, regardless of the CPSR F-bit
- an Imprecise Data abort, unless masked by the CPSR A-bit
- a Debug Entry request, regardless of whether Debug is enabled.

YIELD

`YIELD` indicates to the hardware that the current thread is performing a task, for example a spinlock, that can be swapped out. Hardware can use this hint to suspend and resume threads in a multithreading system.

Architectures

These ARM instructions are available in ARMv6K and above.

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

These 16-bit Thumb instructions are available in ARMv6T2 and above.

See also

Reference

- *NOP* on page 3-144
- *Condition codes* on page 3-163.

3.11.10 DBG

Debug.

Syntax

`DBG{cond} {option}`

where:

cond	is an optional condition code.
option	is an optional limitation on the operation of the hint. The range is 0-15.

Usage

`DBG` is a hint instruction. It is optional whether they are implemented or not. If it is not implemented, it behaves as a `NOP`. The assembler produces a diagnostic message if the instruction executes as `NOP` on the target.

`DBG` executes as a `NOP` instruction in ARMv6K and ARMv6T2.

Debug hint provides a hint to debug and related systems. See their documentation for what use (if any) they make of this instruction.

Architectures

These ARM instructions are available in ARMv6K and above.

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

There are no 16-bit Thumb versions of this instruction.

See also

Reference

- *NOP* on page 3-144
- *Condition codes* on page 3-163.

3.11.11 DMB, DSB, and ISB

Data Memory Barrier, Data Synchronization Barrier, and Instruction Synchronization Barrier.

Syntax

`DMB{cond} {option}`

`DSB{cond} {option}`

`ISB{cond} {option}`

where:

cond	is an optional condition code.
option	is an optional limitation on the operation of the hint.

DMB

Data Memory Barrier acts as a memory barrier. It ensures that all explicit memory accesses that appear in program order before the `DMB` instruction are observed before any explicit memory accesses that appear in program order after the `DMB` instruction. It does not affect the ordering of any other instructions executing on the processor.

Permitted values of *option* are:

`SY`	Full system DMB operation. This is the default and can be omitted.
`ST`	DMB operation that waits only for stores to complete.
`ISH`	DMB operation only to the inner shareable domain.
`ISHST`	DMB operation that waits only for stores to complete, and only to the inner shareable domain.
`NSH`	DMB operation only out to the point of unification.
`NSHST`	DMB operation that waits only for stores to complete and only out to the point of unification.
`OSH`	DMB operation only to the outer shareable domain.
`OSHST`	DMB operation that waits only for stores to complete, and only to the outer shareable domain.

DSB

Data Synchronization Barrier acts as a special kind of memory barrier. No instruction in program order after this instruction executes until this instruction completes. This instruction completes when:

- All explicit memory accesses before this instruction complete.
- All Cache, Branch predictor and TLB maintenance operations before this instruction complete.

Permitted values of *option* are:

`SY`	Full system DSB operation. This is the default and can be omitted.
`ST`	DSB operation that waits only for stores to complete.
`ISH`	DSB operation only to the inner shareable domain.

`ISHST`	DSB operation that waits only for stores to complete, and only to the inner shareable domain.
`NSH`	DSB operation only out to the point of unification.
`NSHST`	DSB operation that waits only for stores to complete and only out to the point of unification.
`OSH`	DSB operation only to the outer shareable domain.
`OSHST`	DSB operation that waits only for stores to complete, and only to the outer shareable domain.

ISB

Instruction Synchronization Barrier flushes the pipeline in the processor, so that all instructions following the `ISB` are fetched from cache or memory, after the instruction has been completed. It ensures that the effects of context altering operations, such as changing the ASID, or completed TLB maintenance operations, or branch predictor maintenance operations, in addition to all changes to the CP15 registers, executed before the `ISB` instruction are visible to the instructions fetched after the `ISB`.

In addition, the `ISB` instruction ensures that any branches that appear in program order after it are always written into the branch prediction logic with the context that is visible after the `ISB` instruction. This is required to ensure correct execution of the instruction stream.

Permitted values of *`option`* are:

`SY`	Full system ISB operation. This is the default, and can be omitted.

Alias

The following alternative values of *`option`* are supported for `DMB` and `DSB`, but ARM recommends that you do not use them:

- `SH` is an alias for `ISH`
- `SHST` is an alias for `ISHST`
- `UN` is an alias for `NSH`
- `UNST` is an alias for `NSHST`

Architectures

These ARM and 32-bit Thumb instructions are available in ARMv7.

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

See also

Reference

- *Condition codes* on page 3-163.

3.11.12 MAR and MRA

Transfer between two general-purpose registers and a 40-bit internal accumulator.

Syntax

MAR{*cond*} *Acc*, *RdLo*, *RdHi*

MRA{*cond*} *RdLo*, *RdHi*, *Acc*

where:

cond	is an optional condition code.
Acc	is the internal accumulator. The standard name is acc*x*,where *x* is an integer in the range 0 to *n*. The value of *n* depends on the processor. It is 0 for current processors.
RdLo, *RdHi*	are general-purpose registers. *RdLo* and *RdHi* must not be the PC, and for MRA they must be different registers.

Usage

The MAR instruction copies the contents of *RdLo* to bits[31:0] of *Acc*, and the least significant byte of *RdHi* to bits[39:32] of *Acc*.

The MRA instruction:
- copies bits[31:0] of *Acc* to *RdLo*
- copies bits[39:32] of *Acc* to *RdHi* bits[7:0]
- sign extends the value by copying bit[39] of *Acc* to bits[31:8] of *RdHi*.

Architectures

These ARM coprocessor 0 instructions are only available in XScale processors.

There are no Thumb versions of these instructions.

Examples

```
    MAR     acc0, r0, r1
    MRA     r4, r5, acc0
    MARNE   acc0, r9, r2
    MRAGT   r4, r8, acc0
```

See also

Reference

- *Condition codes* on page 3-163.

3.12 ThumbEE instructions

Apart from `ENTERX` and `LEAVEX`, these ThumbEE instructions are only accepted when the assembler has been switched into the ThumbEE state using the `--thumbx` command line option or the `THUMBX` directive.

This section contains the following subsections:

- *ENTERX and LEAVEX* on page 3-152

  Switch between Thumb state and ThumbEE state.

- *CHKA* on page 3-153

  Check array.

- *HB, HBL, HBLP, and HBP* on page 3-154

  Handler Branch, branches to a specified handler.

3.12.1 ENTERX and LEAVEX

Switch between Thumb state and ThumbEE state.

Syntax

```
ENTERX
```

```
LEAVEX
```

Usage

`ENTERX` causes a change from Thumb state to ThumbEE state, or has no effect in ThumbEE state.

`LEAVEX` causes a change from ThumbEE state to Thumb state, or has no effect in Thumb state.

Do not use `ENTERX` or `LEAVEX` in an IT block.

Architectures

These instructions are not available in the ARM instruction set.

These 32-bit Thumb and ThumbEE instructions are available in ARMv7, with ThumbEE support.

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

See also

Reference

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

3.12.2 CHKA

`CHKA` (Check Array) compares the unsigned values in two registers.

If the value in the first register is lower than, or the same as, the second, it copies the PC to the LR, and causes a branch to the IndexCheck handler.

Syntax

```
CHKA Rn, Rm
```

where:

Rn	contains the array size. R*n* must not be PC.
Rm	contains the array index. R*n* must not be PC or SP.

Architectures

This instruction is not available in ARM state.

This 16-bit ThumbEE instruction is only available in ARMv7, with ThumbEE support.

3.12.3 HB, HBL, HBLP, and HBP

Handler Branch, branches to a specified handler.

This instruction can optionally store a return address to the LR, pass a parameter to the handler, or both.

Syntax

`HB{L} #HandlerID`

`HB{L}P #imm, #HandlerID`

where:

`L`	is an optional suffix. If `L` is present, the instruction saves a return address in the LR.
`P`	is an optional suffix. If `P` is present, the instruction passes the value of *`imm`* to the handler in `R8`.
`imm`	is an immediate value. If `L` is present, *`imm`* must be in the range 0-31, otherwise *`imm`* must be in the range 0-7.
`HandlerID`	is the index number of the handler to be called. If *`P`* is present, *`HandlerID`* must be in the range 0-31, otherwise *`HandlerID`* must be in the range 0-255.

Architectures

These instructions are not available in ARM state.

These 16-bit ThumbEE instructions are only available in ThumbEE state, in ARMv7 with ThumbEE support.

3.13 Pseudo-instructions

The ARM assembler supports a number of pseudo-instructions that are translated into the appropriate combination of ARM, or Thumb instructions at assembly time.

The pseudo-instructions are described in the following sections:

- *ADRL pseudo-instruction* on page 3-156

  Load a PC-relative or register-relative address into a register (medium range, position independent)

- *MOV32 pseudo--instruction* on page 3-158

  Load a register with a 32-bit immediate value or an address (unlimited range, but not position independent). Available for ARMv6T2 and above only.

- *LDR pseudo-instruction* on page 3-159

  Load a register with a 32-bit immediate value or an address (unlimited range, but not position independent). Available for all ARM architectures.

- *UND pseudo-instruction* on page 3-162

  Generate an architecturally undefined instruction. Available for all ARM architectures.

3.13.1 ADRL pseudo-instruction

Load a PC-relative or register-relative address into a register. It is similar to the `ADR` instruction. `ADRL` can load a wider range of addresses than `ADR` because it generates two data processing instructions.

——— **Note** ———

When assembling Thumb instructions, `ADRL` is only available in ARMv6T2 and later.

———————

Syntax

`ADRL{`*`cond`*`} `*`Rd`*`,`*`label`*

where:

cond	is an optional condition code.
Rd	is the register to load.
label	is a PC-relative or register-relative expression.

Usage

`ADRL` always assembles to two 32-bit instructions. Even if the address can be reached in a single instruction, a second, redundant instruction is produced.

If the assembler cannot construct the address in two instructions, it generates an error message and the assembly fails. You can use the `LDR` pseudo-instruction for loading a wider range of addresses.

`ADRL` produces position-independent code, because the address is PC-relative or register-relative.

If *label* is PC-relative, it must evaluate to an address in the same assembler area as the `ADRL` pseudo-instruction.

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

Architectures and range

The available range depends on the instruction set in use:

ARM	The range of the instruction is 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. See *Operand2 as a constant* on page 3-46 for more information.
32-bit Thumb	±1MB bytes to a byte, halfword, or word-aligned address.
16-bit Thumb	`ADRL` is not available.

The given range is relative to a point four bytes (in Thumb code) or two words (in ARM code) after the address of the current instruction.

See also

Concepts

Using the Assembler:

- *Register-relative and PC-relative expressions* on page 8-7
- *Load immediates into registers* on page 5-5.

Reference

- *LDR pseudo-instruction* on page 3-159
- *AREA* on page 6-61
- *ADD, SUB, RSB, ADC, SBC, and RSC* on page 3-51
- *Condition codes* on page 3-163
- *ARM Architecture Reference Manual*, `http://infocenter.arm.com/help/topic/com.arm.doc.subset.architecture.reference/index.html`.

3.13.2 MOV32 pseudo--instruction

Load a register with either:

- a 32-bit immediate value
- any address.

`MOV32` always generates two 32-bit instructions, a `MOV`, `MOVT` pair. This enables you to load any 32-bit immediate, or to access the whole 32-bit address space.

Syntax

`MOV32{cond} Rd, expr`

where:

cond	is an optional condition code.
Rd	is the register to be loaded. *Rd* must not be SP or PC.
expr	can be any one of the following:

symbol	A label in this or another program area.
#constant	Any 32-bit immediate value.
symbol + constant	A label plus a 32-bit immediate value.

Usage

The main purposes of the `MOV32` pseudo-instruction are:

- To generate literal constants when an immediate value cannot be generated in a single instruction.
- To load a PC-relative or external address into a register. The address remains valid regardless of where the linker places the ELF section containing the `MOV32`.

  Note

  An address loaded in this way is fixed at link time, so the code is *not* position-independent.

`MOV32` sets the Thumb bit (bit 0) of the address if the label referenced is in Thumb code.

Architectures

This pseudo-instruction is available in ARMv6T2 and above in both ARM and Thumb.

Examples

```
    MOV32 r3, #0xABCDEF12   ; loads 0xABCDEF12 into R3
    MOV32 r1, Trigger+12    ; loads the address that is 12 bytes higher than
                            ; the address Trigger into R1
```

See also

Reference

- *Condition codes* on page 3-163.

3.13.3 LDR pseudo-instruction

Load a register with either:

- a 32-bit immediate value
- an address.

Note

This section describes the `LDR` *pseudo*-instruction only, and not the `LDR` instruction.

Syntax

`LDR{cond}{.W} Rt, =expr`

`LDR{cond}{.W} Rt, =label_expr`

where:

cond	is an optional condition code.
`.W`	is an optional instruction width specifier.
Rt	is the register to be loaded.
expr	evaluates to a numeric value.
label_expr	is a PC-relative or external expression of an address in the form of a label plus or minus a numeric value.

Usage

When using the `LDR` pseudo-instruction:

- If the value of *expr* can be loaded with a valid `MOV` or `MVN` instruction, the assembler uses that instruction.
- If a valid `MOV` or `MVN` instruction cannot be used, or if the `label_expr` syntax is used, the assembler places the constant in a literal pool and generates a PC-relative `LDR` instruction that reads the constant from the literal pool.

  Note

  — An address loaded in this way is fixed at link time, so the code is *not* position-independent.

  — The address holding the constant remains valid regardless of where the linker places the ELF section containing the `LDR` instruction.

The assembler places the value of *label_expr* in a literal pool and generates a PC-relative `LDR` instruction that loads the value from the literal pool.

If *label_expr* is an external expression, or is not contained in the current section, the assembler places a linker relocation directive in the object file. The linker generates the address at link time.

If *label_expr* is a local label, the assembler places a linker relocation directive in the object file and generates a symbol for that local label. The address is generated at link time. If the local label references Thumb code, the Thumb bit (bit 0) of the address is set.

The offset from the PC to the value in the literal pool must be less than ±4KB (ARM, 32-bit Thumb) or in the range 0 to +1KB (16-bit Thumb). You are responsible for ensuring that there is a literal pool within range.

If the label referenced is in Thumb code, the `LDR` pseudo-instruction sets the Thumb bit (bit 0) of *`label_expr`*.

Note

In *RealView® Compilation Tools* (RVCT) v2.2, the Thumb bit of the address was not set. If you have code that relies on this behavior, use the command line option `--untyped_local_labels` to force the assembler not to set the Thumb bit when referencing labels in Thumb code.

LDR in Thumb code

You can use the `.W` width specifier to force `LDR` to generate a 32-bit instruction in Thumb code on ARMv6T2 and above processors. `LDR.W` always generates a 32-bit instruction, even if the immediate value could be loaded in a 16-bit `MOV`, or there is a literal pool within reach of a 16-bit PC-relative load.

If the value to be loaded is not known in the first pass of the assembler, `LDR` without `.W` generates a 16-bit instruction in Thumb code, even if that results in a 16-bit PC-relative load for a value that could be generated in a 32-bit `MOV` or `MVN` instruction. However, if the value is known in the first pass, and it can be generated using a 32-bit `MOV` or `MVN` instruction, the `MOV` or `MVN` instruction is used.

The `LDR` pseudo-instruction never generates a 16-bit flag-setting `MOV` instruction. Use the `--diag_warning 1727` assembler command line option to check when a 16-bit instruction could have been used.

You can use the `MOV32` pseudo-instruction for generating immediate values or addresses without loading from a literal pool.

Examples

```
        LDR     r3,=0xff0    ; loads 0xff0 into R3
                             ; =>  MOV.W r3,#0xff0
        LDR     r1,=0xfff    ; loads 0xfff into R1
                             ; =>  LDR r1,[pc,offset_to_litpool]
                             ;     ...
                             ;     litpool DCD 0xfff
        LDR     r2,=place    ; loads the address of
                             ; place into R2
                             ; =>  LDR r2,[pc,offset_to_litpool]
                             ;     ...
                             ;     litpool DCD place
```

See also

Concepts

Using the Assembler:

- *Numeric constants* on page 8-5
- *Register-relative and PC-relative expressions* on page 8-7
- *Local labels* on page 8-12
- *Load immediates into registers* on page 5-5
- *Load immediate 32-bit values to a register using LDR Rd, =const* on page 5-10.

Reference

- *Memory access instructions* on page 3-9
- *LTORG* on page 6-16
- *MOV32 pseudo--instruction* on page 3-158
- *Condition codes* on page 3-163.

3.13.4 UND pseudo-instruction

Generate an architecturally undefined instruction. An attempt to execute an undefined instruction causes the Undefined instruction exception. Architecturally undefined instructions are expected to remain undefined.

Syntax

`UND{cond}{.W} {#expr}`

where:

cond	is an optional condition code.
`.W`	is an optional instruction width specifier.
expr	evaluates to a numeric value. Table 3-10 shows the range and encoding of *expr* in the instruction, where Y shows the locations of the bits that encode for *expr* and V is the 4 bits that encode for the condition code. If *expr* is omitted, the value 0 is used.

Table 3-10 Range and encoding of *expr*

Instruction	Encoding	Number of bits for *expr*	Range
ARM	0xV7FYYYFY	16	0-65535
32-bit Thumb	0xF7FYAYFY	12	0-4095
16-bit Thumb	0xDEYY	8	0-255

UND in Thumb code

You can use the `.W` width specifier to force `UND` to generate a 32-bit instruction in Thumb code on ARMv6T2 and above processors. `UND.W` always generates a 32-bit instruction, even if *expr* is in the range 0-255.

Disassembly

The encodings that this pseudo-instruction produces disassemble to `DCI`.

See also

Reference

- *Condition codes* on page 3-163.

3.14 Condition codes

The instructions that can be conditional have an optional condition code, shown in syntax descriptions as `{cond}`. Table 3-11 shows the condition codes that you can use.

Table 3-11 Condition code suffixes

Suffix	Meaning
`EQ`	Equal
`NE`	Not equal
`CS`	Carry set (identical to HS)
`HS`	Unsigned higher or same (identical to CS)
`CC`	Carry clear (identical to LO)
`LO`	Unsigned lower (identical to CC)
`MI`	Minus or negative result
`PL`	Positive or zero result
`VS`	Overflow
`VC`	No overflow
`HI`	Unsigned higher
`LS`	Unsigned lower or same
`GE`	Signed greater than or equal
`LT`	Signed less than
`GT`	Signed greater than
`LE`	Signed less than or equal
`AL`	Always (this is the default)

Note

The precise meanings of the condition codes depend on whether the condition code flags were set by a VFP instruction or by an ARM data processing instruction.

See also

Concepts

Using the Assembler:
- *Condition code meanings* on page 6-8
- *Conditional execution of NEON and VFP instructions* on page 9-11.

Reference
- *IT* on page 3-120
- *VMRS and VMSR* on page 4-12.

Chapter 4
NEON and VFP Programming

The following topics describe the assembly programming of NEON™ and the VFP coprocessor:

4.1 Instruction summary

This section provides a summary of the NEON and VFP instructions. Use it to locate individual instructions and pseudo-instructions. It contains:

- *NEON instructions*
- *Shared NEON and VFP instructions* on page 4-4
- *VFP instructions* on page 4-5.

4.1.1 NEON instructions

Table 4-1 shows a summary of NEON instructions. These instructions are not available in VFP.

Table 4-1 Location of NEON instructions

Mnemonic	Brief description	See
`VABA`, `VABD`	Absolute difference, Absolute difference and Accumulate	page 4-43
`VABS`	Absolute value	page 4-44
`VACGE`, `VACGT`	Absolute Compare Greater than or Equal, Greater Than	page 4-18
`VACLE`, `VACLT`	Absolute Compare Less than or Equal, Less Than (pseudo-instructions)	page 4-74
`VADD`	Add	page 4-45
`VADDHN`	Add, select High half	page 4-46
`VAND`	Bitwise AND	page 4-14
`VAND`	Bitwise AND (pseudo-instruction)	page 4-73
`VBIC`	Bitwise Bit Clear (register)	page 4-14
`VBIC`	Bitwise Bit Clear (immediate)	page 4-15
`VBIF`, `VBIT`, `VBSL`	Bitwise Insert if False, Insert if True, Select	page 4-16
`VCEQ`, `VCLE`, `VCLT`	Compare Equal, Less than or Equal, Compare Less Than	page 4-19
`VCGE`, `VCGT`	Compare Greater than or Equal, Greater Than	page 4-19
`VCLE`, `VCLT`	Compare Less than or Equal, Compare Less Than (pseudo-instruction)	page 4-75
`VCLS`, `VCLZ`, `VCNT`	Count Leading Sign bits, Count Leading Zeros, and Count set bits	page 4-51
`VCVT`	Convert fixed-point or integer to floating point, floating-point to integer or fixed-point	page 4-22
`VCVT`	Convert between half-precision and single-precision floating-point numbers	page 4-23
`VDUP`	Duplicate scalar to all lanes of vector	page 4-24
`VEOR`	Bitwise Exclusive OR	page 4-14
`VEXT`	Extract	page 4-25
`VFMA`, `VFMS`	Fused Multiply Accumulate, Fused Multiply Subtract (vector)	page 4-58
`VHADD`, `VHSUB`	Halving Add, Halving Subtract	page 4-47
`VLD`	Vector Load	page 4-61
`VMAX`, `VMIN`	Maximum, Minimum	page 4-50
`VMLA`, `VMLS`	Multiply Accumulate, Multiply Subtract (vector)	page 4-56

Table 4-1 Location of NEON instructions (continued)

Mnemonic	Brief description	See
VMLA, VMLS	Multiply Accumulate, Multiply Subtract (by scalar)	page 4-57
VMOV	Move (immediate)	page 4-26
VMOV	Move (register)	page 4-17
VMOVL, VMOV{U}N	Move Long, Move Narrow (register)	page 4-27
VMUL	Multiply (vector)	page 4-56
VMUL	Multiply (by scalar)	page 4-57
VMVN	Move Negative (immediate)	page 4-26
VNEG	Negate	page 4-44
VORN	Bitwise OR NOT	page 4-14
VORN	Bitwise OR NOT (pseudo-instruction)	page 4-73
VORR	Bitwise OR (register)	page 4-14
VORR	Bitwise OR (immediate)	page 4-15
VPADD, VPADAL	Pairwise Add, Pairwise Add and Accumulate	page 4-48
VPMAX, VPMIN	Pairwise Maximum, Pairwise Minimum	page 4-50
VQABS	Absolute value, saturate	page 4-44
VQADD	Add, saturate	page 4-45
VQDMLAL, VQDMLSL	Saturating Doubling Multiply Accumulate, and Multiply Subtract	page 4-59
VQDMUL	Saturating Doubling Multiply	page 4-59
VQDMULH	Saturating Doubling Multiply returning High half	page 4-60
VQMOV{U}N	Saturating Move (register)	page 4-27
VQNEG	Negate, saturate	page 4-44
VQRDMULH	Saturating Doubling Multiply returning High half	page 4-60
VQRSHL	Shift Left, Round, saturate (by signed variable)	page 4-37
VQRSHR{U}N	Shift Right, Round, saturate (by immediate)	page 4-39
VQSHL	Shift Left, saturate (by immediate)	page 4-35
VQSHL	Shift Left, saturate (by signed variable)	page 4-37
VQSHR{U}N	Shift Right, saturate (by immediate)	page 4-39
VQSUB	Subtract, saturate	page 4-45
VRADDHN	Add, select High half, Round	page 4-46
VRECPE	Reciprocal Estimate	page 4-52
VRECPS	Reciprocal Step	page 4-53
VREV	Reverse elements	page 4-28
VRHADD	Halving Add, Round	page 4-47

Table 4-1 Location of NEON instructions (continued)

Mnemonic	Brief description	See
VRSHR, VRSRA	Shift Right and Round, Shift Right, Round, and Accumulate (by immediate)	page 4-38
VRSQRTE	Reciprocal Square Root Estimate	page 4-52
VRSQRTS	Reciprocal Square Root Step	page 4-53
VRSUBHN	Subtract, select High half, Round	page 4-46
VSHL	Shift Left (by immediate)	page 4-35
VSHR	Shift Right (by immediate)	page 4-38
VSLI	Shift Left and Insert	page 4-40
VSRA	Shift Right, Accumulate (by immediate)	page 4-38
VSRI	Shift Right and Insert	page 4-40
VST	Vector Store	page 4-61
VSUB	Subtract	page 4-45
VSUBHN	Subtract, select High half	page 4-46
VSWP	Swap vectors	page 4-29
VTBL, VTBX	Vector table look-up	page 4-30
VTRN	Vector transpose	page 4-31
VTST	Test bits	page 4-20
VUZP, VZIP	Vector interleave and de-interleave	page 4-32

4.1.2 Shared NEON and VFP instructions

Table 4-2 shows a summary of instructions that are common to NEON and VFP.

Table 4-2 Location of shared NEON and VFP instructions

Mnemonic	Brief description	See	Op.	Arch.
VLDM	Load multiple	page 4-8	-	All
VLDR	Load (see also *VLDR pseudo-instruction* on page 4-70)	page 4-7	Scalar	All
	Load (post-increment and pre-decrement)	page 4-71	Scalar	All
VMOV	Transfer from one ARM register to half of a doubleword register	page 4-10	Scalar	All
	Transfer from two ARM registers to a doubleword register	page 4-9	Scalar	VFPv2
	Transfer from half of a doubleword register to ARM register	page 4-10	Scalar	All
	Transfer from a doubleword register to two ARM registers	page 4-9	Scalar	VFPv2
	Transfer from single-precision to ARM register	page 4-11	Scalar	All
	Transfer from ARM register to single-precision	page 4-11	Scalar	All
VMRS	Transfer from NEON and VFP system register to ARM register	page 4-12	-	All
VMSR	Transfer from ARM register to NEON and VFP system register	page 4-12	-	All

Table 4-2 Location of shared NEON and VFP instructions (continued)

Mnemonic	Brief description	See	Op.	Arch.
VPOP	Pop VFP or NEON registers from full-descending stack	page 4-8	-	All
VPUSH	Push VFP or NEON registers to full-descending stack	page 4-8	-	All
VSTM	Store multiple	page 4-8	-	All
VSTR	Store	page 4-7	Scalar	All
	Store (post-increment and pre-decrement)	page 4-71	Scalar	All

4.1.3 VFP instructions

Table 4-3 shows a summary of VFP instructions that are not available in NEON.

Table 4-3 Location of VFP instructions

Mnemonic	Brief description	See	Op.	Arch.
VABS	Absolute value	page 4-77	Vector	All
VADD	Add	page 4-78	Vector	All
VCMP	Compare	page 4-81	Scalar	All
VCVT	Convert between single-precision and double-precision	page 4-82	Scalar	All
	Convert between floating-point and integer	page 4-83	Scalar	All
	Convert between floating-point and fixed-point	page 4-84	Scalar	VFPv3
VCVTB, VCVTT	Convert between half-precision and single-precision floating-point	page 4-85	Scalar	Half-precision
VDIV	Divide	page 4-78	Vector	All
VFMA, VFMS	Fused multiply accumulate, Fused multiply subtract	page 4-80	Scalar	VFPv4
VFNMA, VFNMS	Fused multiply accumulate with negation, Fused multiply subtract with negation	page 4-80	Scalar	VFPv4
VMLA	Multiply accumulate	page 4-79	Vector	All
VMLS	Multiply subtract	page 4-79	Vector	All
VMOV	Insert floating-point immediate in single-precision or double-precision register (see also Table 4-2 on page 4-4)	page 4-86	Scalar	VFPv3
VMUL	Multiply	page 4-79	Vector	All
VNEG	Negate	page 4-77	Vector	All
VNMLA	Negated multiply accumulate	page 4-79	Vector	All
VNMLS	Negated multiply subtract	page 4-79	Vector	All
VNMUL	Negated multiply	page 4-79	Vector	All
VSQRT	Square Root	page 4-77	Vector	All
VSUB	Subtract	page 4-78	Vector	All

4.2 Instructions shared by NEON and VFP

This section contains the following subsections:

- *VLDR and VSTR* on page 4-7

  Extension register load and store.
- *VLDM, VSTM, VPOP, and VPUSH* on page 4-8

  Extension register load and store multiple.
- *VMOV (between two ARM registers and an extension register)* on page 4-9

  Transfer contents between two ARM registers and a 64-bit extension register.
- *VMOV (between an ARM register and a NEON scalar)* on page 4-10

  Transfer contents between an ARM register and a half of a 64-bit extension register.
- *VMOV (between one ARM register and single precision VFP)* on page 4-11

  Transfer contents between a 32-bit extension register and an ARM register.
- *VMRS and VMSR* on page 4-12

  Transfer contents between an ARM register and a NEON and VFP system register.

4.2.1 VLDR and VSTR

Extension register load and store.

Syntax

```
VLDR{cond}{.size} Fd, [Rn{, #offset}]
VSTR{cond}{.size} Fd, [Rn{, #offset}]
VLDR{cond}{.size} Fd, label
VSTR{cond}{.size} Fd, label
```

where:

cond	is an optional condition code.
size	is an optional data size specifier. Must be 32 if *Fd* is an S register, or 64 otherwise.
Fd	is the extension register to be loaded or saved. For a NEON instruction, it must be a D register. For a VFP instruction, it can be either a D or S register.
Rn	is the ARM register holding the base address for the transfer.
offset	is an optional numeric expression. It must evaluate to a numeric value at assembly time. The value must be a multiple of 4, and lie in the range –1020 to +1020. The value is added to the base address to form the address used for the transfer.
label	is a PC-relative expression. *label* must be aligned on a word boundary within ±1KB of the current instruction.

Usage

The VLDR instruction loads an extension register from memory. The VSTR instruction saves the contents of an extension register to memory.

One word is transferred if *Fd* is an S register (VFP only). Two words are transferred otherwise.

There is also an VLDR pseudo-instruction.

See also

Concepts

Using the Assembler:

- *Register-relative and PC-relative expressions* on page 8-7.

Reference

- *Condition codes* on page 3-163
- *VLDR pseudo-instruction* on page 4-70.

4.2.2 VLDM, VSTM, VPOP, and VPUSH

Extension register load multiple, store multiple, pop from stack, push onto stack.

Syntax

VLDM*mode*{*cond*} *Rn*{!}, *Registers*

VSTM*mode*{*cond*} *Rn*{!}, *Registers*

VPOP{*cond*} *Registers*

VPUSH{*cond*} *Registers*

where:

mode must be one of:

IA	meaning Increment address After each transfer. IA is the default, and can be omitted.
DB	meaning Decrement address Before each transfer.
EA	meaning Empty Ascending stack operation. This is the same as DB for loads, and the same as IA for saves.
FD	meaning Full Descending stack operation. This is the same as IA for loads, and the same as DB for saves.

cond is an optional condition code.

Rn is the ARM register holding the base address for the transfer.

! is optional. ! specifies that the updated base address must be written back to *Rn*. If ! is not specified, *mode* must be IA.

Registers is a list of consecutive extension registers enclosed in braces, { and }. The list can be comma-separated, or in range format. There must be at least one register in the list.

You can specify S, D, or Q registers, but they must not be mixed. The number of registers must not exceed 16 D registers, or 8 Q registers. If Q registers are specified, on disassembly they are shown as D registers.

Note

VPOP *Registers* is equivalent to VLDM sp!, *Registers*.

VPUSH *Registers* is equivalent to VSTMDB sp!, *Registers*.

You can use either form of these instructions. They disassemble to VPOP and VPUSH.

See also

Concepts

Using the Assembler:

- *Stack implementation using LDM and STM* on page 5-22.

Reference

- *Condition codes* on page 3-163.

4.2.3 VMOV (between two ARM registers and an extension register)

Transfer contents between two ARM registers and a 64-bit extension register, or two consecutive 32-bit VFP registers.

Syntax

`VMOV{cond} Dm, Rd, Rn`

`VMOV{cond} Rd, Rn, Dm`

`VMOV{cond} Sm, Sm1, Rd, Rn`

`VMOV{cond} Rd, Rn, Sm, Sm1`

where:

cond	is an optional condition code.
Dm	is a 64-bit extension register.
Sm	is a VFP 32-bit register.
Sm1	is the next consecutive VFP 32-bit register after *Sm*.
Rd, *Rn*	are the ARM registers. R*d* and R*n* must not be PC.

Usage

`VMOV` *Dm*, *Rd*, *Rn* transfers the contents of *Rd* into the low half of *Dm*, and the contents of *Rn* into the high half of *Dm*.

`VMOV` *Rd*, *Rn*, *Dm* transfers the contents of the low half of *Dm* into *Rd*, and the contents of the high half of *Dm* into *Rn*.

`VMOV` *Rd*, *Rn*, *Sm*, *Sm1* transfers the contents of *Sm* into *Rd*, and the contents of *Sm1* into *Rn*.

`VMOV` *Sm*, *Sm1*, *Rd*, *Rn* transfers the contents of *Rd* into *Sm*, and the contents of *Rn* into *Sm1*.

Architectures

The 64-bit instructions are available in:
- NEON
- VFPv2 and above.

The 2 x 32-bit instructions are available in VFPv2 and above.

See also

Reference

- *Condition codes* on page 3-163.

4.2.4 VMOV (between an ARM register and a NEON scalar)

Transfer contents between an ARM register and a NEON scalar.

Syntax

`VMOV{cond}{.size} Dn[x], Rd`

`VMOV{cond}{.datatype} Rd, Dn[x]`

where:

cond	is an optional condition code.
size	the data size. Can be `8`, `16`, or `32`. If omitted, *size* is `32`. For VFP instructions, *size* must be `32` or omitted.
datatype	the data type. Can be `U8`, `S8`, `U16`, `S16`, or `32`. If omitted, *datatype* is `32`. For VFP instructions, *datatype* must be `32` or omitted.
Dn[*x*]	is the NEON scalar.
Rd	is the ARM register. *Rd* must not be PC.

Usage

`VMOV Rd, Dn[x]` transfers the contents of `Dn[x]` into the least significant byte, halfword, or word of *Rd*. The remaining bits of *Rd* are either zero or sign extended.

`VMOV Dn[x], Rd` transfers the contents of the least significant byte, halfword, or word of *Rd* into `Dn[x]`.

See also

Concepts

Using the Assembler:

- *NEON scalars* on page 9-20
- *NEON and VFP data types* on page 9-13.

Reference

- *Condition codes* on page 3-163.

4.2.5 VMOV (between one ARM register and single precision VFP)

Transfer contents between a single-precision floating-point register and an ARM register.

Syntax

`VMOV{cond} Rd, Sn`

`VMOV{cond} Sn, Rd`

where:

cond	is an optional condition code.
Sn	is the VFP single-precision register.
Rd	is the ARM register. *Rd* must not be PC.

Usage

`VMOV Rd, Sn` transfers the contents of *Sn* into *Rd*.

`VMOV Sn, Rd` transfers the contents of *Rd* into *Sn*.

See also

Reference

- *Condition codes* on page 3-163.

4.2.6 VMRS and VMSR

Transfer contents between an ARM register and a NEON and VFP system register.

Syntax

`VMRS{`*`cond`*`}` *`Rd`*`,` *`extsysreg`*

`VMSR{`*`cond`*`}` *`extsysreg`*`,` *`Rd`*

where:

`cond`	is an optional condition code.
`extsysreg`	is the NEON and VFP system register, usually `FPSCR`, `FPSID`, or `FPEXC`.
`Rd`	is the ARM register. *`Rd`* must not be PC. It can be `APSR_nzcv`, if *`extsysreg`* is `FPSCR`. In this case, the floating-point status flags are transferred into the corresponding flags in the ARM APSR.

Usage

The `VMRS` instruction transfers the contents of *`extsysreg`* into *`Rd`*.

The `VMSR` instruction transfers the contents of *`Rd`* into *`extsysreg`*.

Note

These instructions stall the ARM until all current NEON or VFP operations complete.

Examples

```
    VMRS    r2,FPCID
    VMRS    APSR_nzcv, FPSCR     ; transfer FP status register to ARM APSR
    VMSR    FPSCR, r4
```

See also

Concepts

Using the Assembler:

- *NEON and VFP system registers* on page 9-23.

Reference

- *Condition codes* on page 3-163.

4.3 NEON logical and compare operations

This section contains the following subsections:

4.3.1 VAND, VBIC, VEOR, VORN, and VORR (register)

`VAND` (Bitwise AND), `VBIC` (Bit Clear), `VEOR` (Bitwise Exclusive OR), `VORN` (Bitwise OR NOT), and `VORR` (Bitwise OR) instructions perform bitwise logical operations between two registers, and place the results in the destination register.

Syntax

`V`*`op`*`{`*`cond`*`}{.`*`datatype`*`} {`*`Qd`*`},` *`Qn`*`,` *`Qm`*

`V`*`op`*`{`*`cond`*`}{.`*`datatype`*`} {`*`Dd`*`},` *`Dn`*`,` *`Dm`*

where:

`op` must be one of:

`AND`	Logical AND
`ORR`	Logical OR
`EOR`	Logical exclusive OR
`BIC`	Logical AND complement
`ORN`	Logical OR complement.

`cond` is an optional condition code.

`datatype` is an optional data type. The assembler ignores *`datatype`*.

`Qd, Qn, Qm` specifies the destination register, the first operand register, and the second operand register, for a quadword operation.

`Dd, Dn, Dm` specifies the destination register, the first operand register, and the second operand register, for a doubleword operation.

Note

`VORR` with the same register for both operands is a `VMOV` instruction. You can use `VORR` in this way, but disassembly of the resulting code produces the `VMOV` syntax.

See also

Reference

- *VMOV, VMVN (register)* on page 4-17
- *Condition codes* on page 3-163.

4.3.2 VBIC and VORR (immediate)

`VBIC` (Bit Clear immediate) takes each element of the destination vector, performs a bitwise AND Complement with an immediate value, and returns the result into the destination vector.

`VORR` (Bitwise OR immediate) takes each element of the destination vector, performs a bitwise OR with an immediate value, and returns the result into the destination vector.

Syntax

`V`*`op`*`{`*`cond`*`}.`*`datatype Qd`*`, #`*`imm`*

`V`*`op`*`{`*`cond`*`}.`*`datatype Dd`*`, #`*`imm`*

where:

op	must be either `BIC` or `ORR`.
cond	is an optional condition code.
datatype	must be either `I8`, `I16`, `I32`, or `I64`.
Qd or *Dd*	is the NEON register for the source and result.
imm	is the immediate value.

Immediate values

You can either specify *`imm`* as a pattern which the assembler repeats to fill the destination register, or you can directly specify the immediate value (that conforms to the pattern) in full. The pattern for *`imm`* depends on *`datatype`* as shown in Table 4-4:

Table 4-4 Patterns for immediate value

I16	I32
`0x00XY`	`0x000000XY`
`0xXY00`	`0x0000XY00`
	`0x00XY0000`
	`0xXY000000`

If you use the `I8` or `I64` datatypes, the assembler will convert it to either the `I16` or `I32` instruction to match the pattern of *`imm`*. If the immediate value does not match any of the patterns in Table 4-4, the assembler generates an error.

See also

Concepts

Using the Assembler:

- *NEON and VFP data types* on page 9-13.

Reference

- *VAND and VORN (immediate)* on page 4-73
- *Condition codes* on page 3-163.

4.3.3 VBIF, VBIT, and VBSL

`VBIT` (Bitwise Insert if True) inserts each bit from the first operand into the destination if the corresponding bit of the second operand is 1, otherwise leaves the destination bit unchanged.

`VBIF` (Bitwise Insert if False) inserts each bit from the first operand into the destination if the corresponding bit of the second operand is 0, otherwise leaves the destination bit unchanged.

`VBSL` (Bitwise Select) selects each bit for the destination from the first operand if the corresponding bit of the destination is 1, or from the second operand if the corresponding bit of the destination is 0.

Syntax

`V`*`op`*`{`*`cond`*`}{.`*`datatype`*`} {`*`Qd`*`}, `*`Qn`*`, `*`Qm`*

`V`*`op`*`{`*`cond`*`}{.`*`datatype`*`} {`*`Dd`*`}, `*`Dn`*`, `*`Dm`*

where:

op	must be one of `BIT`, `BIF`, or `BSL`.
cond	is an optional condition code.
datatype	is an optional datatype. The assembler ignores *datatype*.
Qd, Qn, Qm	specifies the destination register, the first operand register, and the second operand register, for a quadword operation.
Dd, Dn, Dm	specifies the destination register, the first operand register, and the second operand register, for a doubleword operation.

See also

Reference

- *Condition codes* on page 3-163.

4.3.4 VMOV, VMVN (register)

Vector Move (register) copies a value from the source register into the destination register.

Vector Move Not (register) inverts the value of each bit from the source register and places the results into the destination register.

Syntax

```
VMOV{cond}{.datatype} Qd, Qm
```

```
VMOV{cond}{.datatype} Dd, Dm
```

```
VMVN{cond}{.datatype} Qd, Qm
```

```
VMVN{cond}{.datatype} Dd, Dm
```

where:

cond	is an optional condition code.
datatype	is an optional datatype. The assembler ignores *datatype*.
Qd, Qm	specifies the destination vector and the source vector, for a quadword operation.
Dd, Dm	specifies the destination vector and the source vector, for a doubleword operation.

See also

Reference

- *Condition codes* on page 3-163.

4.3.5 VACGE and VACGT

Vector Absolute Compare takes the absolute value of each element in a vector, and compares it with the absolute value of the corresponding element of a second vector. If the condition is true, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

Syntax

`VACop{cond}.F32 {Qd}, Qn, Qm`

`VACop{cond}.F32 {Dd}, Dn, Dm`

where:

op	must be one of: `GE` Absolute Greater than or Equal `GT` Absolute Greater Than.
cond	is an optional condition code.
Qd, Qn, Qm	specifies the destination register, the first operand register, and the second operand register, for a quadword operation.
Dd, Dn, Dm	specifies the destination register, the first operand register, and the second operand register, for a doubleword operation.

The result datatype is `I32`.

See also

Reference

- *VACLE and VACLT* on page 4-74
- *Condition codes* on page 3-163.

4.3.6 VCEQ, VCGE, VCGT, VCLE, and VCLT

Vector Compare takes the value of each element in a vector, and compares it with the value of the corresponding element of a second vector, or zero. If the condition is true, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

Syntax

`VCop{cond}.datatype {Qd}, Qn, Qm`

`VCop{cond}.datatype {Dd}, Dn, Dm`

`VCop{cond}.datatype {Qd}, Qn, #0`

`VCop{cond}.datatype {Dd}, Dn, #0`

where:

op must be one of:

`EQ`	Equal
`GE`	Greater than or Equal
`GT`	Greater Than
`LE`	Less than or Equal (only if the second operand is `#0`)
`LT`	Less Than (only if the second operand is `#0`).

cond is an optional condition code.

datatype must be one of:

- `I8`, `I16`, `I32`, or `F32` for `EQ`
- `S8`, `S16`, `S32`, `U8`, `U16`, `U32`, or `F32` for `GE`, `GT`, `LE`, or `LT` (except `#0` form)
- `S8`, `S16`, `S32`, or `F32` for `GE`, `GT`, `LE`, or `LT` (`#0` form).

The result datatype is:

- `I32` for operand datatypes `I32`, `S32`, `U32`, or `F32`
- `I16` for operand datatypes `I16`, `S16`, or `U16`
- `I8` for operand datatypes `I8`, `S8`, or `U8`.

Qd, Qn, Qm specifies the destination register, the first operand register, and the second operand register, for a quadword operation.

Dd, Dn, Dm specifies the destination register, the first operand register, and the second operand register, for a doubleword operation.

#0 replaces *Qm* or *Dm* for comparisons with zero.

See also

Concepts

Using the Assembler:

- *NEON and VFP data types* on page 9-13.

Reference

- *VCLE and VCLT* on page 4-75
- *Condition codes* on page 3-163.

4.3.7 VTST

`VTST` (Vector Test Bits) takes each element in a vector, and bitwise logical ANDs them with the corresponding element of a second vector. If the result is not zero, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

Syntax

```
VTST{cond}.size {Qd}, Qn, Qm
```

```
VTST{cond}.size {Dd}, Dn, Dm
```

where:

cond	is an optional condition code.
size	must be one of `8`, `16`, or `32`.
Qd, Qn, Qm	specifies the destination register, the first operand register, and the second operand register, for a quadword operation.
Dd, Dn, Dm	specifies the destination register, the first operand register, and the second operand register, for a doubleword operation.

See also

Reference

- *Condition codes* on page 3-163.

4.4 NEON general data processing instructions

This section contains the following subsections:

- *VCVT (between fixed-point or integer, and floating-point)* on page 4-22
  Vector convert between fixed-point or integer and floating-point.
- *VCVT (between half-precision and single-precision floating-point)* on page 4-23
  Vector convert between half-precision and single-precision floating-point.
- *VDUP* on page 4-24
  Duplicate scalar to all lanes of vector.
- *VEXT* on page 4-25
  Extract.
- *VMOV, VMVN (immediate)* on page 4-26
  Move and Move Negative (immediate).
- *VMOVL, V{Q}MOVN, VQMOVUN* on page 4-27
  Move (register).
- *VREV* on page 4-28
  Reverse elements within a vector.
- *VSWP* on page 4-29
  Swap vectors.
- *VTBL, VTBX* on page 4-30
  Vector table look-up.
- *VTRN* on page 4-31
  Vector transpose.
- *VUZP, VZIP* on page 4-32
  Vector interleave and de-interleave.

4.4.1 VCVT (between fixed-point or integer, and floating-point)

`VCVT` (Vector Convert) converts each element in a vector in one of the following ways, and places the results in the destination vector:

- from floating-point to integer
- from integer to floating-point
- from floating-point to fixed-point
- from fixed-point to floating-point.

Syntax

`VCVT{cond}.type Qd, Qm {, #fbits}`

`VCVT{cond}.type Dd, Dm {, #fbits}`

where:

cond	is an optional condition code.
type	specifies the data types for the elements of the vectors. It must be one of:

`S32.F32`	floating-point to signed integer or fixed-point
`U32.F32`	floating-point to unsigned integer or fixed-point
`F32.S32`	signed integer or fixed-point to floating-point
`F32.U32`	unsigned integer or fixed-point to floating-point

Qd, Qm	specifies the destination vector and the operand vector, for a quadword operation.
Dd, Dm	specifies the destination vector and the operand vector, for a doubleword operation.
fbits	if present, specifies the number of fraction bits in the fixed point number. Otherwise, the conversion is between floating-point and integer. *fbits* must lie in the range 0-32. If *fbits* is omitted, the number of fraction bits is 0.

Rounding

Integer or fixed-point to floating-point conversions use round to nearest.

Floating-point to integer or fixed-point conversions use round towards zero.

See also

Reference

- *Condition codes* on page 3-163.

4.4.2 VCVT (between half-precision and single-precision floating-point)

`VCVT` (Vector Convert), with half-precision extension, converts each element in a vector in one of the following ways, and places the results in the destination vector:

- from half-precision floating-point to single-precision floating-point (F32.F16)
- from single-precision floating-point to half-precision floating-point (F16.F32).

Syntax

`VCVT{`*`cond`*`}.F32.F16` *`Qd`*`,` *`Dm`*

`VCVT{`*`cond`*`}.F16.F32` *`Dd`*`,` *`Qm`*

where:

cond	is an optional condition code.
Qd, Dm	specifies the destination vector for the single-precision results and the half-precision operand vector.
Dd, Qm	specifies the destination vector for half-precision results and the single-precision operand vector.

Architectures

This instruction is only available in NEON systems with the half-precision extension.

See also

Reference

- *Condition codes* on page 3-163.

4.4.3 VDUP

`VDUP` (Vector Duplicate) duplicates a scalar into every element of the destination vector. The source can be a NEON scalar or an ARM register.

Syntax

```
VDUP{cond}.size Qd, Dm[x]

VDUP{cond}.size Dd, Dm[x]

VDUP{cond}.size Qd, Rm

VDUP{cond}.size Dd, Rm
```

where:

cond	is an optional condition code.
size	must be `8`, `16`, or `32`.
Qd	specifies the destination register for a quadword operation.
Dd	specifies the destination register for a doubleword operation.
Dm`[`*x*`]`	specifies the NEON scalar.
Rm	specifies the ARM register. *Rm* must not be the R15.

See also

Reference

- *Condition codes* on page 3-163.

4.4.4 VEXT

`VEXT` (Vector Extract) extracts 8-bit elements from the bottom end of the second operand vector and the top end of the first, concatenates them, and places the result in the destination vector. See Figure 4-1 for an example.

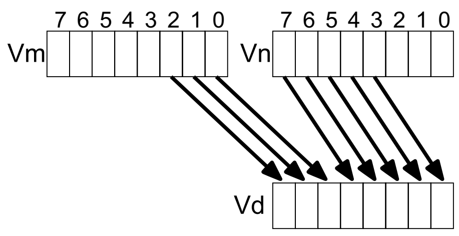

Figure 4-1 Operation of doubleword VEXT for imm = 3

Syntax

VEXT{*cond*}.8 {*Qd*}, *Qn*, *Qm*, #*imm*

VEXT{*cond*}.8 {*Dd*}, *Dn*, *Dm*, #*imm*

where:

cond	is an optional condition code.
Qd, Qn, Qm	specifies the destination register, the first operand register, and the second operand register, for a quadword operation.
Dd, Dn, Dm	specifies the destination register, the first operand register, and the second operand register, for a doubleword operation.
imm	is the number of 8-bit elements to extract from the bottom of the second operand vector, in the range 0-7 for doubleword operations, or 0-15 for quadword operations.

VEXT pseudo-instruction

You can specify a datatype of 16, 32, or 64 instead of 8. In this case, `#imm` refers to halfwords, words, or doublewords instead of referring to bytes, and the permitted ranges are correspondingly reduced.

See also

Reference

- *Condition codes* on page 3-163.

4.4.5 VMOV, VMVN (immediate)

`VMOV` (Vector Move) and `VMVN` (Vector Move Negative) immediate generate an immediate value into the destination register.

Syntax

```
Vop{cond}.datatype Qd, #imm
Vop{cond}.datatype Dd, #imm
```

where:

op	must be either `MOV` or `MVN`.
cond	is an optional condition code.
datatype	must be one of `I8`, `I16`, `I32`, `I64`, or `F32`.
Qd or *Dd*	is the NEON register for the result.
imm	is an immediate value of the type specified by *datatype*. This is replicated to fill the destination register.

Table 4-5 Available immediate values

datatype	VMOV	VMVN
`I8`	`0xXY`	-
`I16`	`0x00XY`, `0xXY00`	`0xFFXY`, `0xXYFF`
`I32`	`0x000000XY`, `0x0000XY00`, `0x00XY0000`, `0xXY000000`	`0xFFFFFFXY`, `0xFFFFXYFF`, `0xFFXYFFFF`, `0xXYFFFFFF`
	`0x0000XYFF`, `0x00XYFFFF`	`0xFFFFXY00`, `0xFFXY0000`
`I64`	byte masks, `0xGGHHJJKKLLMMNNPP` [a]	-
`F32`	floating-point numbers [b]	-

a. Each of `0xGG`, `0xHH`, `0xJJ`, `0xKK`, `0xLL`, `0xMM`, `0xNN`, and `0xPP` must be either `0x00` or `0xFF`.

b. Any number that can be expressed as $+/-n * 2^{-r}$, where n and r are integers, $16 <= n <= 31$, $0 <= r <= 7$.

See also

Concepts

Using the Assembler:

- *NEON and VFP data types* on page 9-13.

Reference

- *Condition codes* on page 3-163.

4.4.6 VMOVL, V{Q}MOVN, VQMOVUN

`VMOVL` (Vector Move Long) takes each element in a doubleword vector, sign or zero extends them to twice their original length, and places the results in a quadword vector.

`VMOVN` (Vector Move and Narrow) copies the least significant half of each element of a quadword vector into the corresponding elements of a doubleword vector.

`VQMOVN` (Vector Saturating Move and Narrow) copies each element of the operand vector to the corresponding element of the destination vector. The result element is half the width of the operand element, and values are saturated to the result width.

`VQMOVUN` (Vector Saturating Move and Narrow, signed operand with Unsigned result) copies each element of the operand vector to the corresponding element of the destination vector. The result element is half the width of the operand element, and values are saturated to the result width.

Syntax

`VMOVL{cond}.datatype Qd, Dm`

`V{Q}MOVN{cond}.datatype Dd, Qm`

`VQMOVUN{cond}.datatype Dd, Qm`

where:

`Q`	if present, specifies that the results are saturated.
`cond`	is an optional condition code.
`datatype`	must be one of:

`S8`, `S16`, `S32`	for `VMOVL`
`U8`, `U16`, `U62`	for `VMOVL`
`I16`, `I32`, `I64`	for `VMOVN`
`S16`, `S32`, `S64`	for `VQMOVN` or `VQMOVUN`
`U16`, `U32`, `U64`	for `VQMOVN`.

`Qd, Dm`	specifies the destination vector and the operand vector for `VMOVL`.
`Dd, Qm`	specifies the destination vector and the operand vector for `V{Q}MOV{U}N`.

See also

Concepts

Using the Assembler:

- *NEON and VFP data types* on page 9-13.

Reference

- *Condition codes* on page 3-163.

4.4.7 VREV

`VREV16` (Vector Reverse within halfwords) reverses the order of 8-bit elements within each halfword of the vector, and places the result in the corresponding destination vector.

`VREV32` (Vector Reverse within words) reverses the order of 8-bit or 16-bit elements within each word of the vector, and places the result in the corresponding destination vector.

`VREV64` (Vector Reverse within doublewords) reverses the order of 8-bit, 16-bit, or 32-bit elements within each doubleword of the vector, and places the result in the corresponding destination vector.

Syntax

```
VREVn{cond}.size Qd, Qm
```

```
VREVn{cond}.size Dd, Dm
```

where:

n	must be one of `16`, `32`, or `64`.
cond	is an optional condition code.
size	must be one of `8`, `16`, or `32`, and must be less than *n*.
Qd, Qm	specifies the destination vector and the operand vector, for a quadword operation.
Dd, Dm	specifies the destination vector and the operand vector, for a doubleword operation.

See also

Reference

- *Condition codes* on page 3-163.

4.4.8 VSWP

`VSWP` (Vector Swap) exchanges the contents of two vectors. The vectors can be either doubleword or quadword. There is no distinction between data types.

Syntax

```
VSWP{cond}{.datatype} Qd, Qm
```

```
VSWP{cond}{.datatype} Dd, Dm
```

where:

`cond`	is an optional condition code.
`datatype`	is an optional datatype. The assembler ignores *`datatype`*.
`Qd, Qm`	specifies the vectors for a quadword operation.
`Dd, Dm`	specifies the vectors for a doubleword operation.

See also

Reference

- *Condition codes* on page 3-163.

4.4.9 VTBL, VTBX

`VTBL` (Vector Table Lookup) uses byte indexes in a control vector to look up byte values in a table and generate a new vector. Indexes out of range return 0.

`VTBX` (Vector Table Extension) works in the same way, except that indexes out of range leave the destination element unchanged.

Syntax

```
Vop{cond}.8 Dd, list, Dm
```

where:

op	must be either `TBL` or `TBX`.
cond	is an optional condition code.
Dd	specifies the destination vector.
list	Specifies the vectors containing the table. It must be one of: • {*Dn*} • {*Dn*,*D(n+1)*} • {*Dn*,*D(n+1)*,*D(n+2)*} • {*Dn*,*D(n+1)*,*D(n+2)*,*D(n+3)*} • {*Qn*,*Q(n+1)*}. All the registers in *list* must be in the range `D0`-`D31` or `Q0`-`Q15` and must not wraparound the end of the register bank. For example {`D31`,`D0`,`D1`} is not permitted. If *list* contains `Q` registers, they disassemble to the equivalent `D` registers.
Dm	specifies the index vector.

See also

Reference

- *Condition codes* on page 3-163.

4.4.10 VTRN

`VTRN` (Vector Transpose) treats the elements of its operand vectors as elements of 2 x 2 matrices, and transposes the matrices. Figure 4-2 and Figure 4-3 show examples of the operation of `VTRN`.

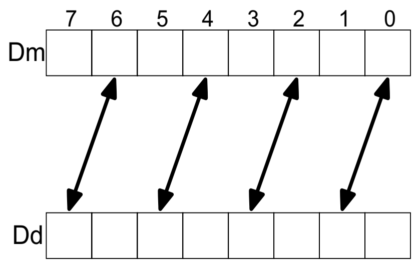

Figure 4-2 Operation of doubleword VTRN.8

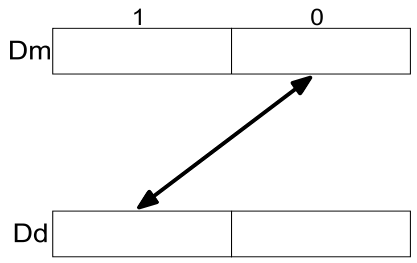

Figure 4-3 Operation of doubleword VTRN.32

Syntax

`VTRN{`*`cond`*`}.`*`size Qd`*`, `*`Qm`*

`VTRN{`*`cond`*`}.`*`size Dd`*`, `*`Dm`*

where:

cond	is an optional condition code.
size	must be one of `8`, `16`, or `32`.
Qd, Qm	specifies the vectors, for a quadword operation.
Dd, Dm	specifies the vectors, for a doubleword operation.

See also

Reference

- *Condition codes* on page 3-163.

4.4.11 VUZP, VZIP

`VZIP` (Vector Zip) interleaves the elements of two vectors.

`VUZP` (Vector Unzip) de-interleaves the elements of two vectors.

De-interleaving is the inverse process of interleaving.

Table 4-6 Operation of doubleword VZIP.8

	Register state before operation								Register state after operation							
Dd	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	B_3	A_3	B_2	A_2	B_1	A_1	B_0	A_0
Dm	B_7	B_6	B_5	B_4	B_3	B_2	B_1	B_0	B_7	A_7	B_6	A_6	B_5	A_5	B_4	A_4

Table 4-7 Operation of quadword VZIP.32

	Register state before operation				Register state after operation			
Qd	A_3	A_2	A_1	A_0	B_1	A_1	B_0	A_0
Qm	B_3	B_2	B_1	B_0	B_3	A_3	B_2	A_2

Table 4-8 Operation of doubleword VUZP.8

	Register state before operation								Register state after operation							
Dd	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	B_6	B_4	B_2	B_0	A_6	A_4	A_2	A_0
Dm	B_7	B_6	B_5	B_4	B_3	B_2	B_1	B_0	B_7	B_5	B_3	B_1	A_7	A_5	A_3	A_1

Table 4-9 Operation of quadword VUZP.32

	Register state before operation				Register state after operation			
Qd	A_3	A_2	A_1	A_0	B_2	B_0	A_2	A_0
Qm	B_3	B_2	B_1	B_0	B_3	B_1	A_3	A_1

Syntax

`V`*`op{cond}.size Qd, Qm`*

`V`*`op{cond}.size Dd, Dm`*

where:

`op`	must be either `UZP` or `ZIP`.
`cond`	is an optional condition code.
`size`	must be one of `8`, `16`, or `32`.
`Qd, Qm`	specifies the vectors, for a quadword operation.
`Dd, Dm`	specifies the vectors, for a doubleword operation.

——— **Note** ———

The following are all the same instruction:

- `VZIP.32` *`Dd, Dm`*

- `VUZP.32 Dd, Dm`
- `VTRN.32 Dd, Dm`

The instruction is disassembled as `VTRN.32 Dd, Dm`.

See also

Reference

- *De-interleaving an array of 3-element structures* on page 4-61
- *VTRN* on page 4-31
- *Condition codes* on page 3-163.

4.5 NEON shift instructions

This section contains the following subsections:

4.5.1 VSHL, VQSHL, VQSHLU, and VSHLL (by immediate)

Vector Shift Left (by immediate) instructions take each element in a vector of integers, left shift them by an immediate value, and place the results in the destination vector.

For `VSHL` (Vector Shift Left), bits shifted out of the left of each element are lost.

For `VQSHL` (Vector Saturating Shift Left) and `VQSHLU` (Vector Saturating Shift Left Unsigned), the sticky QC flag (FPSCR bit[27]) is set if saturation occurs.

For `VSHLL` (Vector Shift Left Long), values are sign or zero extended.

Figure 4-4 shows the operation of `VSHL` with two elements and a shift value of one. The least significant bit in each element in the destination vector is set to zero.

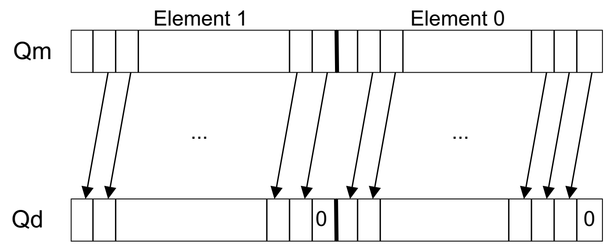

Figure 4-4 Operation of quadword VSHL.64 Qd, Qm, #1

Syntax

`V{Q}SHL{U}{`*`cond`*`}.`*`datatype`*` {`*`Qd`*`}, `*`Qm`*`, #`*`imm`*

`V{Q}SHL{U}{`*`cond`*`}.`*`datatype`*` {`*`Dd`*`}, `*`Dm`*`, #`*`imm`*

`VSHLL{`*`cond`*`}.`*`datatype`*` `*`Qd`*`, `*`Dm`*`, #`*`imm`*

where:

`Q`	if present, indicates that if any of the results overflow, they are saturated.
`U`	only permitted if Q is also present. Indicates that the results are unsigned even though the operands are signed.
`cond`	is an optional condition code.
`datatype`	must be one of:

`I8`, `I16`, `I32`, `I64`	for `VSHL`
`S8`, `S16`, `S32`	for `VSHLL`, `VQSHL`, or `VQSHLU`
`U8`, `U16`, `U32`	for `VSHLL` or `VQSHL`
`S64`	for `VQSHL` or `VQSHLU`
`U64`	for `VQSHL`.

`Qd, Qm`	are the destination and operand vectors, for a quadword operation.
`Dd, Dm`	are the destination and operand vectors, for a doubleword operation.
`Qd, Dm`	are the destination and operand vectors, for a long operation.
`imm`	is the immediate value specifying the size of the shift, in the range:

- 1 to size(*`datatype`*) for `VSHLL`

- 1 to (size(*datatype*) – 1) for VSHL, VQSHL, or VQSHLU.

0 is permitted, but the resulting code disassembles to VMOV or VMOVL.

See also

Concepts

Using the Assembler:

- *NEON and VFP data types* on page 9-13.

Reference

- *Condition codes* on page 3-163.

4.5.2 V{Q}{R}SHL (by signed variable)

VSHL (Vector Shift Left by signed variable) takes each element in a vector, shifts them by a value from the least significant byte of the corresponding element of a second vector, and places the results in the destination vector. If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift.

The results can be optionally saturated, rounded, or both. The sticky QC flag (FPSCR bit[27]) is set if saturation occurs.

Syntax

`V{Q}{R}SHL{cond}.datatype {Qd}, Qm, Qn`

`V{Q}{R}SHL{cond}.datatype {Dd}, Dm, Dn`

where:

`Q`	if present, indicates that If any of the results overflow, they are saturated.
`R`	if present, indicates that each result is rounded. Otherwise, each result is truncated.
cond	is an optional condition code.
datatype	must be one of `S8`, `S16`, `S32`, `S64`, `U8`, `U16`, `U32`, or `U64`.
Qd, *Qm*, *Qn*	are the destination vector, the first operand vector, and the second operand vector, for a quadword operation.
Dd, *Dm*, *Dn*	are the destination vector, the first operand vector, and the second operand vector, for a doubleword operation.

See also

Concepts

Using the Assembler:

- *NEON and VFP data types* on page 9-13.

Reference

- *Condition codes* on page 3-163.

4.5.3 V{R}SHR{N}, V{R}SRA (by immediate)

`V{R}SHR{N}` (Vector Shift Right by immediate value) takes each element in a vector, right shifts them by an immediate value, and places the results in the destination vector. The results can be optionally rounded, or narrowed, or both.

`V{R}SRA` (Vector Shift Right by immediate value and Accumulate) takes each element in a vector, right shifts them by an immediate value, and accumulates the results into the destination vector. The results can be optionally rounded.

Syntax

```
V{R}SHR{cond}.datatype {Qd}, Qm, #imm

V{R}SHR{cond}.datatype {Dd}, Dm, #imm

V{R}SRA{cond}.datatype {Qd}, Qm, #imm

V{R}SRA{cond}.datatype {Dd}, Dm, #imm

V{R}SHRN{cond}.datatype Dd, Qm, #imm
```

where:

`R`	if present, indicates that the results are rounded. Otherwise, the results are truncated.
cond	is an optional condition code.
datatype	must be one of:

`S8, S16, S32, S64`	for `V{R}SHR` or `V{R}SRA`
`U8, U16, U32, U64`	for `V{R}SHR` or `V{R}SRA`
`I16, I32, I64`	for `V{R}SHRN`.

Qd, *Qm*	are the destination vector and the operand vector, for a quadword operation.
Dd, *Dm*	are the destination vector and the operand vector, for a doubleword operation.
Dd, *Qm*	are the destination vector and the operand vector, for a narrow operation.
imm	is the immediate value specifying the size of the shift, in the range 0 to (size(*datatype*) – 1).

See also

Concepts

Using the Assembler:

- *NEON and VFP data types* on page 9-13.

Reference

- *Condition codes* on page 3-163.

4.5.4 VQ{R}SHR{U}N (by immediate)

`VQ{R}SHR{U}N` (Vector Saturating Shift Right, Narrow, by immediate value, with optional Rounding) takes each element in a quadword vector of integers, right shifts them by an immediate value, and places the results in a doubleword vector.

The sticky QC flag (FPSCR bit[27]) is set if saturation occurs.

Syntax

`VQ{R}SHR{U}N{cond}.datatype Dd, Qm, #imm`

where:

`R`	if present, indicates that the results are rounded. Otherwise, the results are truncated.
`U`	if present, indicates that the results are unsigned, although the operands are signed. Otherwise, the results are the same type as the operands.
`cond`	is an optional condition code.
`datatype`	must be one of:
	`S16, S32, S64` for `VQ{R}SHRN` or `VQ{R}SHRUN`
	`U16, U32, U64` for `VQ{R}SHRN` only.
`Dd, Qm`	are the destination vector and the operand vector.
`imm`	is the immediate value specifying the size of the shift, in the range 0 to (size(*`datatype`*) – 1).

See also

Concepts

Using the Assembler:

- *NEON and VFP data types* on page 9-13.

Reference

- *Condition codes* on page 3-163.

4.5.5 VSLI and VSRI

`VSLI` (Vector Shift Left and Insert) takes each element in a vector, left shifts them by an immediate value, and inserts the results in the destination vector. Bits shifted out of the left of each element are lost. Figure 4-5 shows the operation of `VSLI` with two elements and a shift value of one. The least significant bit in each element in the destination vector is unchanged.

`VSRI` (Vector Shift Right and Insert) takes each element in a vector, right shifts them by an immediate value, and inserts the results in the destination vector. Bits shifted out of the right of each element are lost. Figure 4-6 shows the operation of `VSRI` with a single element and a shift value of two. The two most significant bits in the destination vector are unchanged.

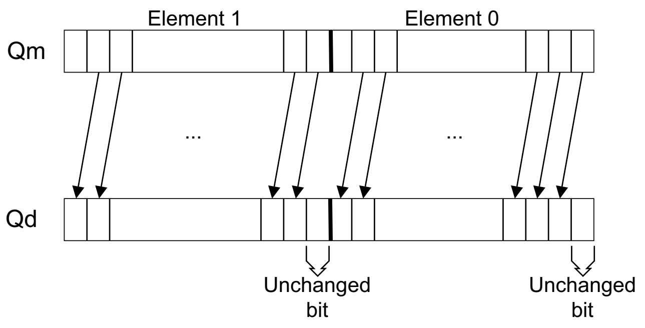

Figure 4-5 Operation of quadword VSLI.64 Qd, Qm, #1

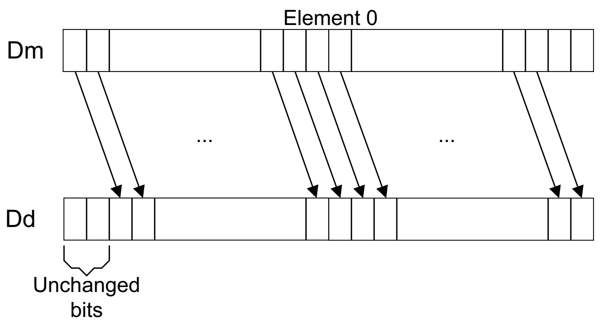

Figure 4-6 Operation of doubleword VSRI.64 Dd, Dm, #2

Syntax

`Vop{cond}.size {Qd}, Qm, #imm`

`Vop{cond}.size {Dd}, Dm, #imm`

where:

op	must be either `SLI` or `SRI`.
cond	is an optional condition code.
size	must be one of `8`, `16`, `32`, or `64`.
Qd, *Qm*	are the destination vector and the operand vector, for a quadword operation.
Dd, *Dm*	are the destination vector and the operand vector, for a doubleword operation.

imm	is the immediate value specifying the size of the shift, in the range: • 0 to (*size* – 1) for VSLI • 1 to *size* for VSRI.

See also

Reference

- *Condition codes* on page 3-163.

4.6 NEON general arithmetic instructions

This section contains the following subsections:

4.6.1 VABA{L} and VABD{L}

`VABA` (Vector Absolute Difference and Accumulate) subtracts the elements of one vector from the corresponding elements of another vector, and accumulates the absolute values of the results into the elements of the destination vector.

`VABD` (Vector Absolute Difference) subtracts the elements of one vector from the corresponding elements of another vector, and places the absolute values of the results into the elements of the destination vector.

Long versions of both instructions are available.

Syntax

`V`*`op`*`{`*`cond`*`}.`*`datatype`* `{`*`Qd`*`}, `*`Qn`*`, `*`Qm`*

`V`*`op`*`{`*`cond`*`}.`*`datatype`* `{`*`Dd`*`}, `*`Dn`*`, `*`Dm`*

`V`*`op`*`L{`*`cond`*`}.`*`datatype Qd`*`, `*`Dn`*`, `*`Dm`*

where:

op	must be either `ABA` or `ABD`.
cond	is an optional condition code.
datatype	must be one of: • `S8`, `S16`, `S32`, `U8`, `U16`, or `U32` for `VABA`, `VABAL`, or `VABDL` • `S8`, `S16`, `S32`, `U8`, `U16`, `U32` or `F32` for `VABD`.
Qd, *Qn*, *Qm*	are the destination vector, the first operand vector, and the second operand vector, for a quadword operation.
Dd, *Dn*, *Dm*	are the destination vector, the first operand vector, and the second operand vector, for a doubleword operation.
Qd, *Dn*, *Dm*	are the destination vector, the first operand vector, and the second operand vector, for a long operation.

See also

Concepts

Using the Assembler:

- *NEON and VFP data types* on page 9-13.

Reference

- *Condition codes* on page 3-163.

4.6.2 V{Q}ABS and V{Q}NEG

`VABS` (Vector Absolute) takes the absolute value of each element in a vector, and places the results in a second vector. (The floating-point version only clears the sign bit.)

`VNEG` (Vector Negate) negates each element in a vector, and places the results in a second vector. (The floating-point version only inverts the sign bit.)

Saturating versions of both instructions are available. The sticky QC flag (FPSCR bit[27]) is set if saturation occurs.

Syntax

```
V{Q}op{cond}.datatype Qd, Qm
```

```
V{Q}op{cond}.datatype Dd, Dm
```

where:

`Q`	if present, indicates that If any of the results overflow, they are saturated.
`op`	must be either `ABS` or `NEG`.
`cond`	is an optional condition code.
`datatype`	must be one of:
	`S8`, `S16`, `S32` — for `VABS`, `VNEG`, `VQABS`, or `VQNEG`
	`F32` — for `VABS` and `VNEG` only.
`Qd`, *`Qm`*	are the destination vector and the operand vector, for a quadword operation.
`Dd`, *`Dm`*	are the destination vector and the operand vector, for a doubleword operation.

See also

Concepts

Using the Assembler:

- *NEON and VFP data types* on page 9-13.

Reference

- *Condition codes* on page 3-163.

4.6.3 V{Q}ADD, VADDL, VADDW, V{Q}SUB, VSUBL, and VSUBW

VADD (Vector Add) adds corresponding elements in two vectors, and places the results in the destination vector.

VSUB (Vector Subtract) subtracts the elements of one vector from the corresponding elements of another vector, and places the results in the destination vector.

VADD and VSUB have these forms:

- Saturating. The sticky QC flag (FPSCR bit[27]) is set if saturation occurs.
- Long.
- Wide.

Syntax

```
V{Q}op{cond}.datatype {Qd}, Qn, Qm    ; Saturating instruction
V{Q}op{cond}.datatype {Dd}, Dn, Dm    ; Saturating instruction
VopL{cond}.datatype Qd, Dn, Dm        ; Long instruction
VopW{cond}.datatype {Qd}, Qn, Dm      ; Wide instruction
```

where:

`Q`	if present, indicates that if any of the results overflow, they are saturated.
op	must be either ADD or SUB.
cond	is an optional condition code.
datatype	must be one of:
	I8, I16, I32, I64, F32 for VADD or VSUB
	S8, S16, S32 for VQADD, VQSUB, VADDL, VADDW, VSUBL, or VSUBW
	U8, U16, U32 for VQADD, VQSUB, VADDL, VADDW, VSUBL, or VSUBW
	S64, U64 for VQADD or VQSUB.
Qd, *Qn*, *Qm*	are the destination vector, the first operand vector, and the second operand vector, for a quadword operation.
Dd, *Dn*, *Dm*	are the destination vector, the first operand vector, and the second operand vector, for a doubleword operation.
Qd, *Dn*, *Dm*	are the destination vector, the first operand vector, and the second operand vector, for a long operation.
Qd, *Qn*, *Dm*	are the destination vector, the first operand vector, and the second operand vector, for a wide operation.

See also

Concepts

Using the Assembler:

- *NEON and VFP data types* on page 9-13.

Reference

- *Condition codes* on page 3-163.

4.6.4 V{R}ADDHN and V{R}SUBHN

`V{R}ADDH` (Vector Add and Narrow, selecting High half) adds corresponding elements in two vectors, selects the most significant halves of the results, and places the final results in the destination vector. Results can be either rounded or truncated.

`V{R}SUBH` (Vector Subtract and Narrow, selecting High half) subtracts the elements of one vector from the corresponding elements of another vector, selects the most significant halves of the results, and places the final results in the destination vector. Results can be either rounded or truncated.

Syntax

```
V{R}opHN{cond}.datatype Dd, Qn, Qm
```

where:

`R`	if present, indicates that each result is rounded. Otherwise, each result is truncated.
`op`	must be either `ADD` or `SUB`.
`cond`	is an optional condition code.
`datatype`	must be one of `I16`, `I32`, or `I64`.
`Dd`, *`Qn`*, *`Qm`*	are the destination vector, the first operand vector, and the second operand vector.

See also

Concepts

Using the Assembler:

- *NEON and VFP data types* on page 9-13.

Reference

- *Condition codes* on page 3-163.

4.6.5 V{R}HADD and VHSUB

`VHADD` (Vector Halving Add) adds corresponding elements in two vectors, shifts each result right one bit, and places the results in the destination vector. Results can be either rounded or truncated.

`VHSUB` (Vector Halving Subtract) subtracts the elements of one vector from the corresponding elements of another vector, shifts each result right one bit, and places the results in the destination vector. Results are always truncated.

Syntax

```
V{R}HADD{cond}.datatype {Qd}, Qn, Qm
```

```
V{R}HADD{cond}.datatype {Dd}, Dn, Dm
```

```
VHSUB{cond}.datatype {Qd}, Qn, Qm
```

```
VHSUB{cond}.datatype {Dd}, Dn, Dm
```

where:

`R`	if present, indicates that each result is rounded. Otherwise, each result is truncated.
`cond`	is an optional condition code.
`datatype`	must be one of `S8`, `S16`, `S32`, `U8`, `U16`, or `U32`.
`Qd`, *`Qn`*, *`Qm`*	are the destination vector, the first operand vector, and the second operand vector, for a quadword operation.
`Dd`, *`Dn`*, *`Dm`*	are the destination vector, the first operand vector, and the second operand vector, for a doubleword operation.

See also

Concepts

Using the Assembler:

- *NEON and VFP data types* on page 9-13.

Reference

- *Condition codes* on page 3-163.

4.6.6 VPADD{L}, VPADAL

`VPADD` (Vector Pairwise Add) adds adjacent pairs of elements of two vectors, and places the results in the destination vector.

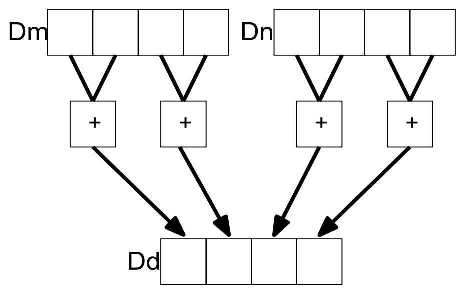

Figure 4-7 Example of operation of VPADD (in this case, for data type `I16`)

`VPADDL` (Vector Pairwise Add Long) adds adjacent pairs of elements of a vector, sign or zero extends the results to twice their original width, and places the final results in the destination vector.

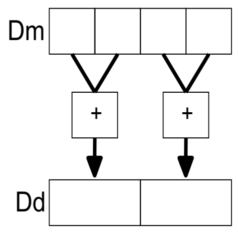

Figure 4-8 Example of operation of doubleword VPADDL (in this case, for data type `S16`)

`VPADAL` (Vector Pairwise Add and Accumulate Long) adds adjacent pairs of elements of a vector, and accumulates the absolute values of the results into the elements of the destination vector.

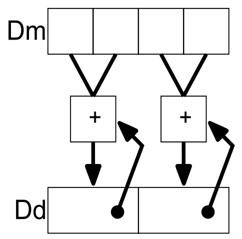

Figure 4-9 Example of operation of VPADAL (in this case for data type `S16`)

Syntax

`VPADD{`*`cond`*`}.`*`datatype`*` {`*`Dd`*`}, `*`Dn`*`, `*`Dm`*

`VP`*`op`*`L{`*`cond`*`}.`*`datatype Qd`*`, `*`Qm`*

`VP`*`op`*`L{`*`cond`*`}.`*`datatype Dd`*`, `*`Dm`*

where:

op	must be either `ADD` or `ADA`.
cond	is an optional condition code.

datatype	must be one of:	
	`I8`, `I16`, `I32`, `F32`	for `VPADD`
	`S8`, `S16`, `S32`	for `VPADDL` or `VPADAL`
	`U8`, `U16`, `U32`	for `VPADDL` or `VPADAL`.
Dd, *Dn*, *Dm*	are the destination vector, the first operand vector, and the second operand vector, for a `VPADD` instruction.	
Qd, *Qm*	are the destination vector and the operand vector, for a quadword `VPADDL` or `VPADAL`.	
Dd, *Dm*	are the destination vector and the operand vector, for a doubleword `VPADDL` or `VPADAL`.	

See also

Concepts

Using the Assembler:

- *NEON and VFP data types* on page 9-13.

Reference

- *Condition codes* on page 3-163.

4.6.7 VMAX, VMIN, VPMAX, and VPMIN

`VMAX` (Vector Maximum) compares corresponding elements in two vectors, and copies the larger of each pair into the corresponding element in the destination vector.

`VMIN` (Vector Minimum) compares corresponding elements in two vectors, and copies the smaller of each pair into the corresponding element in the destination vector.

`VPMAX` (Vector Pairwise Maximum) compares adjacent pairs of elements in two vectors, and copies the larger of each pair into the corresponding element in the destination vector. Operands and results must be doubleword vectors.

`VPMIN` (Vector Pairwise Minimum) compares adjacent pairs of elements in two vectors, and copies the smaller of each pair into the corresponding element in the destination vector. Operands and results must be doubleword vectors.

Syntax

```
Vop{cond}.datatype Qd, Qn, Qm
```

```
Vop{cond}.datatype Dd, Dn, Dm
```

```
VPop{cond}.datatype Dd, Dn, Dm
```

where:

op	must be either `MAX` or `MIN`.
cond	is an optional condition code.
datatype	must be one of `S8`, `S16`, `S32`, `U8`, `U16`, `U32`, or `F32`.
Qd, *Qn*, *Qm*	are the destination vector, the first operand vector, and the second operand vector, for a quadword operation.
Dd, *Dn*, *Dm*	are the destination vector, the first operand vector, and the second operand vector, for a doubleword operation.

Floating-point maximum and minimum

max(+0.0, –0.0) = +0.0.

min(+0.0, –0.0) = –0.0

If any input is a NaN, the corresponding result element is the default NaN.

See also

Concepts

Using the Assembler:

- *NEON and VFP data types* on page 9-13.

Reference

- *Condition codes* on page 3-163
- *VPADD{L}, VPADAL* on page 4-48.

4.6.8 VCLS, VCLZ, and VCNT

`VCLS` (Vector Count Leading Sign bits) counts the number of consecutive bits following the topmost bit, that are the same as the topmost bit, in each element in a vector, and places the results in a second vector.

`VCLZ` (Vector Count Leading Zeros) counts the number of consecutive zeros, starting from the top bit, in each element in a vector, and places the results in a second vector.

`VCNT` (Vector Count set bits) counts the number of bits that are one in each element in a vector, and places the results in a second vector.

Syntax

```
Vop{cond}.datatype Qd, Qm
```

```
Vop{cond}.datatype Dd, Dm
```

where:

op	must be one of `CLS`, `CLZ`, or `CNT`.
cond	is an optional condition code.
datatype	must be one of: • `S8`, `S16`, or `S32` for `CLS`. • `I8`, `I16`, or `I32` for `CLZ`. • `I8` for `CNT`.
Qd, *Qm*	are the destination vector and the operand vector, for a quadword operation.
Dd, *Dm*	are the destination vector and the operand vector, for a doubleword operation.

See also

Concepts

Using the Assembler:

- *NEON and VFP data types* on page 9-13.

Reference

- *Condition codes* on page 3-163.

4.6.9 VRECPE and VRSQRTE

`VRECPE` (Vector Reciprocal Estimate) finds an approximate reciprocal of each element in a vector, and places the results in a second vector.

`VRSQRTE` (Vector Reciprocal Square Root Estimate) finds an approximate reciprocal square root of each element in a vector, and places the results in a second vector.

Syntax

```
Vop{cond}.datatype Qd, Qm
```

```
Vop{cond}.datatype Dd, Dm
```

where:

op	must be either `RECPE` or `RSQRTE`.
cond	is an optional condition code.
datatype	must be either `U32` or `F32`.
Qd, *Qm*	are the destination vector and the operand vector, for a quadword operation.
Dd, *Dm*	are the destination vector and the operand vector, for a doubleword operation.

Results for out-of-range inputs

Table 4-10 shows the results where input values are out of range.

Table 4-10 Results for out-of-range inputs

	Operand element (`VRECPE`)	**Operand element (`VRSQRTE`)**	**Result element**
Integer	<= `0x7FFFFFFF`	<= `0x3FFFFFFF`	`0xFFFFFFFF`
Floating-point	NaN	NaN, Negative Normal, Negative Infinity	Default NaN
	Negative 0, Negative Denormal	Negative 0, Negative Denormal	Negative Infinity [a]
	Positive 0, Positive Denormal	Positive 0, Positive Denormal	Positive Infinity [a]
	Positive infinity	Positive infinity	Positive 0
	Negative infinity		Negative 0

a. The Division by Zero exception bit in the FPSCR (FPSCR[1]) is set

See also

Concepts

Using the Assembler:

- *NEON and VFP data types* on page 9-13.

Reference

- *Condition codes* on page 3-163.

4.6.10 VRECPS and VRSQRTS

`VRECPS` (Vector Reciprocal Step) multiplies the elements of one vector by the corresponding elements of another vector, subtracts each of the results from 2, and places the final results into the elements of the destination vector.

`VRSQRTS` (Vector Reciprocal Square Root Step) multiplies the elements of one vector by the corresponding elements of another vector, subtracts each of the results from 3, divides these results by two, and places the final results into the elements of the destination vector.

Syntax

```
Vop{cond}.F32 {Qd}, Qn, Qm
```

```
Vop{cond}.F32 {Dd}, Dn, Dm
```

where:

op	must be either `RECPS` or `RSQRTS`.
cond	is an optional condition code.
Qd, *Qn*, *Qm*	are the destination vector, the first operand vector, and the second operand vector, for a quadword operation.
Dd, *Dn*, *Dm*	are the destination vector, the first operand vector, and the second operand vector, for a doubleword operation.

Results for out-of-range inputs

Table 4-11 shows the results where input values are out of range.

Table 4-11 Results for out-of-range inputs

1st operand element	2nd operand element	Result element (`VRECPS`)	Result element (`VRSQRTS`)
NaN	-	Default NaN	Default NaN
-	NaN	Default NaN	Default NaN
+/– 0.0 or denormal	+/– infinity	2.0	1.5
+/– infinity	+/– 0.0 or denormal	2.0	1.5

Usage

The Newton-Raphson iteration:

$x_{n+1} = x_n(2-dx_n)$

converges to $(1/d)$ if x_0 is the result of `VRECPE` applied to d.

The Newton-Raphson iteration:

$x_{n+1} = x_n(3-dx_n^2)/2$

converges to $(1/\sqrt{d})$if x_0 is the result of `VRSQRTE` applied to d.

See also

Reference

- *Condition codes* on page 3-163.

4.7 NEON multiply instructions

This section contains the following subsections:

- *VMUL{L}, VMLA{L}, and VMLS{L}* on page 4-56.

  Vector Multiply, Multiply Accumulate, and Multiply Subtract.

- *VMUL{L}, VMLA{L}, and VMLS{L} (by scalar)* on page 4-57.

  Vector Multiply, Multiply Accumulate, and Multiply Subtract (by scalar).

- *VFMA, VFMS* on page 4-58.

  Vector Fused Multiply Accumulate and Vector Fused Multiply Subtract.

- *VQDMULL, VQDMLAL, and VQDMLSL (by vector or by scalar)* on page 4-59

  Vector Saturating Doubling Multiply, Multiply Accumulate, and Multiply Subtract (by vector or scalar).

- *VQ{R}DMULH (by vector or by scalar)* on page 4-60

  Vector Saturating Doubling Multiply returning High half (by vector or scalar).

4.7.1 VMUL{L}, VMLA{L}, and VMLS{L}

`VMUL` (Vector Multiply) multiplies corresponding elements in two vectors, and places the results in the destination vector.

`VMLA` (Vector Multiply Accumulate) multiplies corresponding elements in two vectors, and accumulates the results into the elements of the destination vector.

`VMLS` (Vector Multiply Subtract) multiplies corresponding elements in two vectors, subtracts the results from corresponding elements of the destination vector, and places the final results in the destination vector.

Syntax

```
Vop{cond}.datatype {Qd}, Qn, Qm
```

```
Vop{cond}.datatype {Dd}, Dn, Dm
```

```
VopL{cond}.datatype Qd, Dn, Dm
```

where:

op must be one of:

`MUL`	Multiply
`MLA`	Multiply Accumulate
`MLS`	Multiply Subtract.

cond is an optional condition code.

datatype must be one of:

`I8`, `I16`, `I32`, `F32`	for `VMUL`, `VMLA`, or `VMLS`
`S8`, `S16`, `S32`	for `VMULL`, `VMLAL`, or `VMLSL`
`U8`, `U16`, `U32`	for `VMULL`, `VMLAL`, or `VMLSL`
`P8`	for `VMUL` or `VMULL`.

Qd, *Qn*, *Qm* are the destination vector, the first operand vector, and the second operand vector, for a quadword operation.

Dd, *Dn*, *Dm* are the destination vector, the first operand vector, and the second operand vector, for a doubleword operation.

Qd, *Dn*, *Dm* are the destination vector, the first operand vector, and the second operand vector, for a long operation.

See also

Concepts

Using the Assembler:

- *Polynomial arithmetic over {0,1}* on page 9-22
- *NEON and VFP data types* on page 9-13.

Reference

- *Condition codes* on page 3-163.

4.7.2 VMUL{L}, VMLA{L}, and VMLS{L} (by scalar)

`VMUL` (Vector Multiply by scalar) multiplies each element in a vector by a scalar, and places the results in the destination vector.

`VMLA` (Vector Multiply Accumulate) multiplies each element in a vector by a scalar, and accumulates the results into the corresponding elements of the destination vector.

`VMLS` (Vector Multiply Subtract) multiplies each element in a vector by a scalar, and subtracts the results from the corresponding elements of the destination vector, and places the final results in the destination vector.

Syntax

```
Vop{cond}.datatype {Qd}, Qn, Dm[x]

Vop{cond}.datatype {Dd}, Dn, Dm[x]

VopL{cond}.datatype Qd, Dn, Dm[x]
```

where:

op must be one of:

`MUL`	Multiply
`MLA`	Multiply Accumulate
`MLS`	Multiply Subtract.

cond is an optional condition code.

datatype must be one of:

`I16`, `I32`, `F32`	for `VMUL`, `VMLA`, or `VMLS`
`S16`, `S32`	for `VMULL`, `VMLAL`, or `VMLSL`
`U16`, `U32`	for `VMULL`, `VMLAL`, or `VMLSL`.

Qd, *Qn*	are the destination vector and the first operand vector, for a quadword operation.
Dd, *Dn*	are the destination vector and the first operand vector, for a doubleword operation.
Qd, *Dn*	are the destination vector and the first operand vector, for a long operation.
Dm[*x*]	is the scalar holding the second operand.

See also

Concepts

Using the Assembler:

- *NEON and VFP data types* on page 9-13.

Reference

- *Condition codes* on page 3-163.

4.7.3 VFMA, VFMS

`VFMA` (Vector Fused Multiply Accumulate) multiplies corresponding elements in the two operand vectors, and accumulates the results into the elements of the destination vector. The result of the multiply is not rounded before the accumulation.

`VFMS` (Vector Fused Multiply Subtract) multiplies corresponding elements in the two operand vectors, then subtracts the products from the corresponding elements of the destination vector, and places the final results in the destination vector. The result of the multiply is not rounded before the subtraction.

Syntax

```
Vop{cond}.F32 {Qd}, Qn, Qm

Vop{cond}.F32 {Dd}, Dn, Dm

Vop{cond}.F64 {Dd}, Dn, Dm

Vop{cond}.F32 {Sd}, Sn, Sm
```

where:

op	is one of `FMA` or `FMS`.
cond	is an optional condition code.
Sd, *Sn*, *Sm*	are the destination and operand vectors for word operation.
Dd, *Dn*, *Dm*	are the destination and operand vectors for doubleword operation.
Qd, *Qn*, *Qm*	are the destination and operand vectors for quadword operation.

See also

Reference

- *Condition codes* on page 3-163
- *VMUL{L}, VMLA{L}, and VMLS{L}* on page 4-56.

4.7.4 VQDMULL, VQDMLAL, and VQDMLSL (by vector or by scalar)

Vector Saturating Doubling Multiply instructions multiply their operands and double the results. `VQDMULL` places the results in the destination register. `VQDMLAL` adds the results to the values in the destination register. `VQDMLSL` subtracts the results from the values in the destination register.

If any of the results overflow, they are saturated. The sticky QC flag (FPSCR bit[27]) is set if saturation occurs.

Syntax

`VQDopL{cond}.datatype Qd, Dn, Dm`

`VQDopL{cond}.datatype Qd, Dn, Dm[x]`

where:

op	must be one of:
	`MUL` Multiply
	`MLA` Multiply Accumulate
	`MLS` Multiply Subtract.
cond	is an optional condition code.
datatype	must be either `S16` or `S32`.
Qd, *Dn*	are the destination vector and the first operand vector.
Dm	is the vector holding the second operand, for a *by vector* operation.
Dm[*x*]	is the scalar holding the second operand, for a *by scalar* operation.

See also

Concepts

Using the Assembler:

- *NEON and VFP data types* on page 9-13.

Reference

- *Condition codes* on page 3-163.

4.7.5 VQ{R}DMULH (by vector or by scalar)

Vector Saturating Doubling Multiply instructions multiply their operands and double the results. They return only the high half of the results.

If any of the results overflow, they are saturated. The sticky QC flag (FPSCR bit[27]) is set if saturation occurs.

Syntax

```
VQ{R}DMULH{cond}.datatype {Qd}, Qn, Qm
```

```
VQ{R}DMULH{cond}.datatype {Dd}, Dn, Dm
```

```
VQ{R}DMULH{cond}.datatype {Qd}, Qn, Dm[x]
```

```
VQ{R}DMULH{cond}.datatype {Dd}, Dn, Dm[x]
```

where:

`R`	if present, indicates that each result is rounded. Otherwise, each result is truncated.
cond	is an optional condition code.
datatype	must be either `S16` or `S32`.
Qd, *Qn*	are the destination vector and the first operand vector, for a quadword operation.
Dd, *Dn*	are the destination vector and the first operand vector, for a doubleword operation.
Qm or *Dm*	is the vector holding the second operand, for a *by vector* operation.
Dm[*x*]	is the scalar holding the second operand, for a *by scalar* operation.

See also

Concepts

Using the Assembler:

- *NEON and VFP data types* on page 9-13.

Reference

- *Condition codes* on page 3-163.

4.8 NEON load and store element and structure instructions

This section contains the following subsections:

- *Interleaving*.
- *Alignment restrictions in load and store, element and structure instructions* on page 4-62.
- *VLDn and VSTn (single n-element structure to one lane)* on page 4-63.

  This is used for almost all data accesses. A normal vector can be loaded (*n* = 1).
- *VLDn (single n-element structure to all lanes)* on page 4-65.
- *VLDn and VSTn (multiple n-element structures)* on page 4-67.

4.8.1 Interleaving

Many instructions in this group provide interleaving when structures are stored to memory, and de-interleaving when structures are loaded from memory. Figure 4-10 shows an example of de-interleaving. Interleaving is the inverse process.

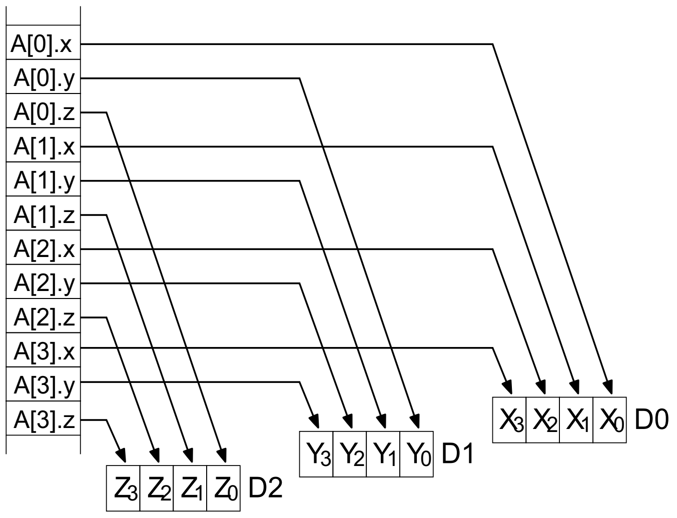

Figure 4-10 De-interleaving an array of 3-element structures

4.8.2 Alignment restrictions in load and store, element and structure instructions

Many of these instructions permit memory alignment restrictions to be specified. When the alignment is not specified in the instruction, the alignment restriction is controlled by the A bit (SCTLR bit[1]):

- if the A bit is 0, there are no alignment restrictions (except for strongly ordered or device memory, where accesses must be element aligned or the result is UNPREDICTABLE)
- if the A bit is 1, accesses must be element aligned.

If an address is not correctly aligned, an alignment fault occurs.

See also

Reference

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

4.8.3 VLD*n* and VST*n* (single *n*-element structure to one lane)

Vector Load single *n*-element structure to one lane. It loads one *n*-element structure from memory into one or more NEON registers. Elements of the register that are not loaded are unaltered.

Vector Store single *n*-element structure to one lane. It stores one *n*-element structure into memory from one or more NEON registers.

Syntax

```
Vopn{cond}.datatype list, [Rn{@align}]{!}
```

```
Vopn{cond}.datatype list, [Rn{@align}], Rm
```

where:

op	must be either `LD` or `ST`.
n	must be one of 1, 2, 3, or 4.
cond	is an optional condition code.
datatype	see Table 4-12.
list	specifies the NEON register list. See Table 4-12 for options.
Rn	is the ARM register containing the base address. *Rn* cannot be PC.
align	specifies an optional alignment. See Table 4-12 for options.
!	if ! is present, *Rn* is updated to (*Rn* + the number of bytes transferred by the instruction). The update occurs after all the loads or stores have taken place.
Rm	is an ARM register containing an offset from the base address. If *Rm* is present, *Rn* is updated to (*Rn* + *Rm*) *after* the address is used to access memory. *Rm* cannot be SP or PC.

Table 4-12 Permitted combinations of parameters

n	*datatype*	list [a]	*align* [b]	alignment
1	8	`{Dd[x]}`	-	Standard only
	16	`{Dd[x]}`	`@16`	2-byte
	32	`{Dd[x]}`	`@32`	4-byte
2	8	`{Dd[x], D(d+1)[x]}`	`@16`	2-byte
	16	`{Dd[x], D(d+1)[x]}`	`@32`	4-byte
		`{Dd[x], D(d+2)[x]}`	`@32`	4-byte
	32	`{Dd[x], D(d+1)[x]}`	`@64`	8-byte
		`{Dd[x], D(d+2)[x]}`	`@64`	8-byte
3	8	`{Dd[x], D(d+1)[x], D(d+2)[x]}`	-	Standard only
	16 or 32	`{Dd[x], D(d+1)[x], D(d+2)[x]}`	-	Standard only
		`{Dd[x], D(d+2)[x], D(d+4)[x]}`	-	Standard only
4	8	`{Dd[x], D(d+1)[x], D(d+2)[x], D(d+3)[x]}`	`@32`	4-byte

Table 4-12 Permitted combinations of parameters (continued)

n	*datatype*	list [a]	*align* [b]	alignment
	16	`{Dd[x], D(d+1)[x], D(d+2)[x], D(d+3)[x]}`	`@64`	8-byte
		`{Dd[x], D(d+2)[x], D(d+4)[x], D(d+6)[x]}`	`@64`	8-byte
	32	`{Dd[x], D(d+1)[x], D(d+2)[x], D(d+3)[x]}`	`@64` or `@128`	8-byte or 16-byte
		`{Dd[x], D(d+2)[x], D(d+4)[x], D(d+6)[x]}`	`@64` or `@128`	8-byte or 16-byte

a. Every register in the list must be in the range D0-D31.

b. `align` can be omitted. In this case, standard alignment rules apply, see *Alignment restrictions in load and store, element and structure instructions* on page 4-62.

See also

Reference

- *Condition codes* on page 3-163.

4.8.4 VLD*n* (single *n*-element structure to all lanes)

Vector Load single *n*-element structure to all lanes. It loads multiple copies of one *n*-element structure from memory into one or more NEON registers.

Syntax

`VLDn{cond}.datatype list, [Rn{@align}]{!}`

`VLDn{cond}.datatype list, [Rn{@align}], Rm`

where:

n	must be one of 1, 2, 3, or 4.
cond	is an optional condition code.
datatype	see Table 4-13.
list	specifies the NEON register list. See Table 4-13 for options.
Rn	is the ARM register containing the base address. *Rn* cannot be PC.
align	specifies an optional alignment. See Table 4-13 for options.
!	if ! is present, *Rn* is updated to (*Rn* + the number of bytes transferred by the instruction). The update occurs after all the loads or stores have taken place.
Rm	is an ARM register containing an offset from the base address. If *Rm* is present, *Rn* is updated to (*Rn* + *Rm*) *after* the address is used to access memory. *Rm* cannot be SP or PC.

Table 4-13 Permitted combinations of parameters

n	*datatype*	list [a]	*align* [b]	alignment
1	8	`{Dd[]}`	-	Standard only
		`{Dd[],D(d+1)[]}`	-	Standard only
	16	`{Dd[]}`	`@16`	2-byte
		`{Dd[],D(d+1)[]}`	`@16`	2-byte
	32	`{Dd[]}`	`@32`	4-byte
		`{Dd[],D(d+1)[]}`	`@32`	4-byte
2	8	`{Dd[], D(d+1)[]}`	`@8`	byte
		`{Dd[], D(d+2)[]}`	`@8`	byte
	16	`{Dd[], D(d+1)[]}`	`@16`	2-byte
		`{Dd[], D(d+2)[]}`	`@16`	2-byte
	32	`{Dd[], D(d+1)[]}`	`@32`	4-byte
		`{Dd[], D(d+2)[]}`	`@32`	4-byte
3	8, 16, or 32	`{Dd[], D(d+1)[], D(d+2)[]}`	-	Standard only
		`{Dd[], D(d+2)[], D(d+4)[]}`	-	Standard only
4	8	`{Dd[], D(d+1)[], D(d+2)[], D(d+3)[]}`	`@32`	4-byte

Table 4-13 Permitted combinations of parameters (continued)

n	*datatype*	list [a]	*align* [b]	alignment
		`{Dd[], D(d+2)[], D(d+4)[], D(d+6)[]}`	`@32`	4-byte
	16	`{Dd[], D(d+1)[], D(d+2)[], D(d+3)[]}`	`@64`	8-byte
		`{Dd[], D(d+2)[], D(d+4)[], D(d+6)[]}`	`@64`	8-byte
	32	`{Dd[], D(d+1)[], D(d+2)[], D(d+3)[]}`	`@64` or `@128`	8-byte or 16-byte
		`{Dd[], D(d+2)[], D(d+4)[], D(d+6)[]}`	`@64` or `@128`	8-byte or 16-byte

a. Every register in the list must be in the range `D0`-`D31`.

b. *`align`* can be omitted. In this case, standard alignment rules apply, see *Alignment restrictions in load and store, element and structure instructions* on page 4-62.

See also

Reference

- *Condition codes* on page 3-163.

4.8.5 VLD*n* and VST*n* (multiple *n*-element structures)

Vector Load multiple *n*-element structures. It loads multiple *n*-element structures from memory into one or more NEON registers, with de-interleaving (unless *n* == 1). Every element of each register is loaded.

Vector Store multiple *n*-element structures. It stores multiple *n*-element structures to memory from one or more NEON registers, with interleaving (unless *n* == 1). Every element of each register is stored.

Syntax

`Vopn{cond}.datatype list, [Rn{@align}]{!}`

`Vopn{cond}.datatype list, [Rn{@align}], Rm`

where:

op	must be either `LD` or `ST`.
n	must be one of 1, 2, 3, or 4.
cond	is an optional condition code.
datatype	see Table 4-14 for options.
list	specifies the NEON register list. See Table 4-14 for options.
Rn	is the ARM register containing the base address. *Rn* cannot be PC.
align	specifies an optional alignment. See Table 4-14 for options.
!	if ! is present, *Rn* is updated to (*Rn* + the number of bytes transferred by the instruction). The update occurs after all the loads or stores have taken place.
Rm	is an ARM register containing an offset from the base address. If *Rm* is present, *Rn* is updated to (*Rn* + *Rm*) *after* the address is used to access memory. *Rm* cannot be SP or PC.

Table 4-14 Permitted combinations of parameters

n	*datatype*	list [a]	*align* [b]	*alignment*
1	8, 16, 32, or 64	`{Dd}`	`@64`	8-byte
		`{Dd, D(d+1)}`	`@64` or `@128`	8-byte or 16-byte
		`{Dd, D(d+1), D(d+2)}`	`@64`	8-byte
		`{Dd, D(d+1), D(d+2), D(d+3)}`	`@64`, `@128`, or `@256`	8-byte, 16-byte, or 32-byte
2	8, 16, or 32	`{Dd, D(d+1)}`	`@64`, `@128`	8-byte or 16-byte
		`{Dd, D(d+2)}`	`@64`, `@128`	8-byte or 16-byte
		`{Dd, D(d+1), D(d+2), D(d+3)}`	`@64`, `@128`, or `@256`	8-byte, 16-byte, or 32-byte
3	8, 16, or 32	`{Dd, D(d+1), D(d+2)}`	`@64`	8-byte
		`{Dd, D(d+2), D(d+4)}`	`@64`	8-byte
4	8, 16, or 32	`{Dd, D(d+1), D(d+2), D(d+3)}`	`@64`, `@128`, or `@256`	8-byte, 16-byte, or 32-byte
		`{Dd, D(d+2), D(d+4), D(d+6)}`	`@64`, `@128`, or `@256`	8-byte, 16-byte, or 32-byte

a. Every register in the list must be in the range `D0-D31`.

b. *`align`* can be omitted. In this case, standard alignment rules apply, see *Alignment restrictions in load and store, element and structure instructions* on page 4-62.

See also

Reference

- *Condition codes* on page 3-163.

4.9 NEON and VFP pseudo-instructions

This section contains the following subsections:

- *VLDR pseudo-instruction* on page 4-70 (NEON and VFP)
- *VLDR and VSTR (post-increment and pre-decrement)* on page 4-71 (NEON and VFP)
- *VMOV2* on page 4-72 (NEON only)
- *VAND and VORN (immediate)* on page 4-73 (NEON only)
- *VACLE and VACLT* on page 4-74 (NEON only)
- *VCLE and VCLT* on page 4-75 (NEON only).

4.9.1 VLDR pseudo-instruction

The `VLDR` pseudo-instruction loads a constant value into every element of a 64-bit NEON vector, or into a VFP single-precision or double-precision register.

——— **Note** ———

This section describes the `VLDR` *pseudo*-instruction only.

———————————

Syntax

`VLDR{`*`cond`*`}.`*`datatype Dd`*`,=`*`constant`*

`VLDR{`*`cond`*`}.`*`datatype Sd`*`,=`*`constant`*

where:

`datatype`	must be one of:
	`I`*`n`* NEON only
	`S`*`n`* NEON only
	`U`*`n`* NEON only
	`F32` NEON or VFP
	`F64` VFP only
`n`	must be one of 8, 16, 32, or 64.
`cond`	is an optional condition code.
`Dd` `or` *`Sd`*	is the extension register to be loaded.
`constant`	is an immediate value of the appropriate type for *`datatype`*.

Usage

If an instruction (for example, `VMOV`) is available that can generate the constant directly into the register, the assembler uses it. Otherwise, it generates a doubleword literal pool entry containing the constant and loads the constant using a `VLDR` instruction.

See also

Concepts

Using the Assembler:

- *NEON and VFP data types* on page 9-13.

Reference

- *VLDR and VSTR* on page 4-7
- *Condition codes* on page 3-163.

4.9.2 VLDR and VSTR (post-increment and pre-decrement)

Pseudo-instructions that load or store extension registers with post-increment and pre-decrement.

Note

There are also `VLDR` and `VSTR` instructions without post-increment and pre-decrement.

Syntax

```
op{cond}{.size} Fd, [Rn], #offset        ; post-increment
```

```
op{cond}{.size} Fd, [Rn, #-offset]!      ; pre-decrement
```

where:

op	can be: • `VLDR` - load extension register from memory • `VSTR` - store contents of extension register to memory.
cond	is an optional condition code.
size	is an optional data size specifier. Must be 32 if *Fd* is an `S` register, or 64 if *Fd* is a `D` register.
Fd	is the extension register to be loaded or saved. For a NEON instruction, it must be a doubleword (*Dd*) register. For a VFP instruction, it can be either a double precision (*Dd*) or a single precision (*Sd*) register.
Rn	is the ARM register holding the base address for the transfer.
offset	is a numeric expression that must evaluate to a numeric value at assembly time. The value must be `4` if *Fd* is an `S` register, or `8` if *Fd* is a `D` register.

Usage

The post-increment instruction increments the base address in the register by the offset value, after the transfer. The pre-decrement instruction decrements the base address in the register by the offset value, and then performs the transfer using the new address in the register. These pseudo-instructions assemble to `VLDM` or `VSTM` instructions.

See also

Reference

- *VLDR and VSTR* on page 4-7
- *VLDM, VSTM, VPOP, and VPUSH* on page 4-8
- *Condition codes* on page 3-163.

4.9.3 VMOV2

The `VMOV2` pseudo-instruction generates an immediate value and places it in every element of a NEON vector, without loading a value from a literal pool. It always assembles to exactly two instructions.

`VMOV2` can generate any 16-bit immediate value, and a restricted range of 32-bit and 64-bit immediate values.

Syntax

`VMOV2{cond}.datatype Qd, #constant`

`VMOV2{cond}.datatype Dd, #constant`

where:

datatype	must be one of: • `I8`, `I16`, `I32`, or `I64` • `S8`, `S16`, `S32`, or `S64` • `U8`, `U16`, `U32`, or `U64` • `F32`.
cond	is an optional condition code.
Qd or *Dd*	is the extension register to be loaded.
constant	is an immediate value of the appropriate type for *datatype*.

Usage

`VMOV2` typically assembles to a `VMOV` or `VMVN` instruction, followed by a `VBIC` or `VORR` instruction.

See also

Concepts

Using the Assembler:

- *NEON and VFP data types* on page 9-13.

Reference

- *VMOV, VMVN (immediate)* on page 4-26
- *VBIC and VORR (immediate)* on page 4-15
- *Condition codes* on page 3-163.

4.9.4 VAND and VORN (immediate)

`VAND` (Bitwise AND immediate) takes each element of the destination vector, performs a bitwise AND with an immediate value, and returns the result into the destination vector.

`VORN` (Bitwise OR NOT immediate) takes each element of the destination vector, performs a bitwise OR Complement with an immediate value, and returns the result into the destination vector.

Note

On disassembly, these pseudo-instructions are disassembled to the corresponding `VBIC` and `VORR` instructions, with the complementary immediate values.

Syntax

```
Vop{cond}.datatype Qd, #imm
```

```
Vop{cond}.datatype Dd, #imm
```

where:

op	must be either `VAND` or `VORN`.
cond	is an optional condition code.
datatype	must be either `I8`, `I16`, `I32`, or `I64`.
Qd or *Dd*	is the NEON register for the result.
imm	is the immediate value.

Immediate values

If *datatype* is `I16`, the immediate value must have one of the following forms:

- `0xFFXY`
- `0xXYFF`.

If *datatype* is `I32`, the immediate value must have one of the following forms:

- `0xFFFFFFXY`
- `0xFFFFXYFF`
- `0xFFXYFFFF`
- `0xXYFFFFFF`.

See also

Concepts

Using the Assembler:

- *NEON and VFP data types* on page 9-13.

Reference

- *VBIC and VORR (immediate)* on page 4-15
- *Condition codes* on page 3-163.

4.9.5 VACLE and VACLT

Vector Absolute Compare takes the absolute value of each element in a vector, and compares it with the absolute value of the corresponding element of a second vector. If the condition is true, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

Note

On disassembly, these pseudo-instructions are disassembled to the corresponding `VACGE` and `VACGT` instructions, with the operands reversed.

Syntax

VAC*op*{*cond*}.*datatype* {*Qd*}, *Qn*, *Qm*

VAC*op*{*cond*}.*datatype* {*Dd*}, *Dn*, *Dm*

where:

op must be one of:

`LE`	Absolute Less than or Equal
`LT`	Absolute Less Than.

cond is an optional condition code.

datatype must be `F32`.

Qd or *Dd* is the NEON register for the result.

The result datatype is `I32`.

Qn or *Dn* is the NEON register holding the first operand.

Qm or *Dm* is the NEON register holding the second operand.

See also

Concepts

Using the Assembler:

- *NEON and VFP data types* on page 9-13.

Reference

- *Condition codes* on page 3-163
- *VACGE and VACGT* on page 4-18.

4.9.6 VCLE and VCLT

Vector Compare takes the value of each element in a vector, and compares it with the value of the corresponding element of a second vector, or zero. If the condition is true, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

Note

On disassembly, these pseudo-instructions are disassembled to the corresponding `VCGE` and `VCGT` instructions, with the operands reversed.

Syntax

`VCop{cond}.datatype {Qd}, Qn, Qm`

`VCop{cond}.datatype {Dd}, Dn, Dm`

where:

op must be one of:

`LE`	Less than or Equal
`LT`	Less Than.

cond is an optional condition code.

datatype must be one of `S8`, `S16`, `S32`, `U8`, `U16`, `U32`, or `F32`.

Qd or *Dd* is the NEON register for the result.

The result datatype is:
- `I32` for operand datatypes `I32`, `S32`, `U32`, or `F32`
- `I16` for operand datatypes `I16`, `S16`, or `U16`
- `I8` for operand datatypes `I8`, `S8`, or `U8`.

Qn or *Dn* is the NEON register holding the first operand.

Qm or *Dm* is the NEON register holding the second operand.

See also

Concepts

Using the Assembler:
- *NEON and VFP data types* on page 9-13.

Reference
- *Condition codes* on page 3-163
- *VCLE and VCLT*.

4.10 VFP instructions

This section contains the following subsections:

4.10.1 VABS, VNEG, and VSQRT

Floating-point absolute value, negate, and square root.

These instructions can be scalar, vector, or mixed.

Syntax

`V`*`op`*`{`*`cond`*`}.F32 `*`Sd`*`, `*`Sm`*

`V`*`op`*`{`*`cond`*`}.F64 `*`Dd`*`, `*`Dm`*

where:

op	is one of `ABS`, `NEG`, or `SQRT`.
cond	is an optional condition code.
Sd, *Sm*	are the single-precision registers for the result and operand.
Dd, *Dm*	are the double-precision registers for the result and operand.

Usage

The `VABS` instruction takes the contents of *Sm* or *Dm*, clears the sign bit, and places the result in *Sd* or *Dd*. This gives the absolute value.

The `VNEG` instruction takes the contents of *Sm* or *Dm*, changes the sign bit, and places the result in *Sd* or *Dd*. This gives the negation of the value.

The `VSQRT` instruction takes the square root of the contents of *Sm* or *Dm*, and places the result in *Sd* or *Dd*.

In the case of a `VABS` and `VNEG` instruction, if the operand is a NaN, the sign bit is determined in each case as above, but no exception is produced.

Floating-point exceptions

`VABS` and `VNEG` instructions cannot produce any exceptions.

`VSQRT` instructions can produce Invalid Operation or Inexact exceptions.

See also

Concepts

Using the Assembler:

- *Control of scalar, vector, and mixed operations* on page 9-38.

Reference

- *Condition codes* on page 3-163.

4.10.2 VADD, VSUB, and VDIV

Floating-point add, subtract, and divide.

These instructions can be scalar, vector, or mixed.

Syntax

`Vop{cond}.F32 {Sd}, Sn, Sm`

`Vop{cond}.F64 {Dd}, Dn, Dm`

where:

op	is one of `ADD`, `SUB`, or `DIV`.
cond	is an optional condition code.
Sd, *Sn*, *Sm*	are the single-precision registers for the result and operands.
Dd, *Dn*, *Dm*	are the double-precision registers for the result and operands.

Usage

The `VADD` instruction adds the values in the operand registers and places the result in the destination register.

The `VSUB` instruction subtracts the value in the second operand register from the value in the first operand register, and places the result in the destination register.

The `VDIV` instruction divides the value in the first operand register by the value in the second operand register, and places the result in the destination register.

Floating-point exceptions

`VADD` and `VSUB` instructions can produce Invalid Operation, Overflow, or Inexact exceptions.

`VDIV` operations can produce Division by Zero, Invalid Operation, Overflow, Underflow, or Inexact exceptions.

See also

Concepts

Using the Assembler:

- *Control of scalar, vector, and mixed operations* on page 9-38.

Reference

- *Condition codes* on page 3-163.

4.10.3 VMUL, VMLA, VMLS, VNMUL, VNMLA, and VNMLS

Floating-point multiply and multiply accumulate, with optional negation.

These instructions can be scalar, vector, or mixed.

Syntax

```
V{N}MUL{cond}.F32 {Sd,} Sn, Sm
```

```
V{N}MUL{cond}.F64 {Dd,} Dn, Dm
```

```
V{N}MLA{cond}.F32 Sd, Sn, Sm
```

```
V{N}MLA{cond}.F64 Dd, Dn, Dm
```

```
V{N}MLS{cond}.F32 Sd, Sn, Sm
```

```
V{N}MLS{cond}.F64 Dd, Dn, Dm
```

where:

`N`	negates the final result.
cond	is an optional condition code.
Sd, *Sn*, *Sm*	are the single-precision registers for the result and operands.
Dd, *Dn*, *Dm*	are the double-precision registers for the result and operands.

Usage

The `VMUL` operation multiplies the values in the operand registers and places the result in the destination register.

The `VMLA` operation multiplies the values in the operand registers, adds the value in the destination register, and places the final result in the destination register.

The `VMLS` operation multiplies the values in the operand registers, subtracts the result from the value in the destination register, and places the final result in the destination register.

In each case, the final result is negated if the `N` option is used.

Floating-point exceptions

These instructions can produce Invalid Operation, Overflow, Underflow, Inexact, or Input Denormal exceptions.

See also

Concepts

Using the Assembler:

- *Control of scalar, vector, and mixed operations* on page 9-38.

Reference

- *Condition codes* on page 3-163.

4.10.4 VFMA, VFMS, VFNMA, VFNMS

Fused floating-point multiply accumulate and fused floating-point multiply subtract with optional negation.

These instructions are always scalar.

Syntax

`VF{N}op{cond}.F64 {Dd}, Dn, Dm`

`VF{N}op{cond}.F32 {Sd}, Sn, Sm`

where:

op	is one of `MA` or `MS`.
`N`	negates the final result.
cond	is an optional condition code.
Sd, *Sn*, *Sm*	are the single-precision registers for the result and operands.
Dd, *Dn*, *Dm*	are the double-precision registers for the result and operands.
Qd, *Qn*, *Qm*	are the double-precision registers for the result and operands.

Usage

`VFMA` multiplies the values in the operand registers, adds the value in the destination register, and places the final result in the destination register. The result of the multiply is not rounded before the accumulation.

`VFMS` multiplies the values in the operand registers, subtracts the product from the value in the destination register, and places the final result in the destination register. The result of the multiply is not rounded before the subtraction.

In each case, the final result is negated if the `N` option is used.

Floating-point exceptions

These instructions can produce Input Denormal, Invalid Operation, Overflow, Underflow, or Inexact exceptions.

See also

Concepts

Using the Assembler:

- *Control of scalar, vector, and mixed operations* on page 9-38.

Reference

- *Condition codes* on page 3-163
- *VMUL, VMLA, VMLS, VNMUL, VNMLA, and VNMLS* on page 4-79.

4.10.5 VCMP

Floating-point compare.

`VCMP` is always scalar.

Syntax

```
VCMP{cond}.F32 Sd, Sm

VCMP{cond}.F32 Sd, #0

VCMP{cond}.F64 Dd, Dm

VCMP{cond}.F64 Dd, #0
```

where:

cond	is an optional condition code.
Sd, *Sm*	are the single-precision registers holding the operands.
Dd, Dm	are the double-precision registers holding the operands.

Usage

The `VCMP` instruction subtracts the value in the second operand register (or 0 if the second operand is #0) from the value in the first operand register, and sets the VFP condition flags on the result.

Floating-point exceptions

`VCMP` instructions can produce Invalid Operation exceptions.

See also

Reference

- *Condition codes* on page 3-163.

4.10.6 VCVT (between single-precision and double-precision)

Convert between single-precision and double-precision numbers.

`VCVT` is always scalar.

Syntax

```
VCVT{cond}.F64.F32 Dd, Sm
```

```
VCVT{cond}.F32.F64 Sd, Dm
```

where:

cond	is an optional condition code.
Dd	is a double-precision register for the result.
Sm	is a single-precision register holding the operand.
Sd	is a single-precision register for the result.
Dm	is a double-precision register holding the operand.

Usage

These instructions convert the single-precision value in *Sm* to double-precision and places the result in *Dd*, or the double-precision value in *Dm* to single-precision and place the result in *Sd*.

Floating-point exceptions

These instructions can produce Invalid Operation, Input Denormal, Overflow, Underflow, or Inexact exceptions.

See also

Reference

- *Condition codes* on page 3-163.

4.10.7 VCVT (between floating-point and integer)

Convert between floating-point numbers and integers.

`VCVT` is always scalar.

Syntax

```
VCVT{R}{cond}.type.F64 Sd, Dm
```

```
VCVT{R}{cond}.type.F32 Sd, Sm
```

```
VCVT{cond}.F64.type Dd, Sm
```

```
VCVT{cond}.F32.type Sd, Sm
```

where:

`R`	makes the operation use the rounding mode specified by the FPSCR. Otherwise, the operation rounds towards zero.
cond	is an optional condition code.
type	can be either `U32` (unsigned 32-bit integer) or `S32` (signed 32-bit integer).
Sd	is a single-precision register for the result.
Dd	is a double-precision register for the result.
Sm	is a single-precision register holding the operand.
Dm	is a double-precision register holding the operand.

Usage

The first two forms of this instruction convert from floating-point to integer.

The third and fourth forms convert from integer to floating-point.

Floating-point exceptions

These instructions can produce Input Denormal, Invalid Operation, or Inexact exceptions.

See also

Reference

- *Condition codes* on page 3-163.

4.10.8 VCVT (between floating-point and fixed-point)

Convert between floating-point and fixed-point numbers.

`VCVT` is always scalar.

Syntax

```
VCVT{cond}.type.F64 Dd, Dd, #fbits
VCVT{cond}.type.F32 Sd, Sd, #fbits
VCVT{cond}.F64.type Dd, Dd, #fbits
VCVT{cond}.F32.type Sd, Sd, #fbits
```

where:

cond	is an optional condition code.
type	can be any one of:

`S16`	16-bit signed fixed-point number
`U16`	16-bit unsigned fixed-point number
`S32`	32-bit signed fixed-point number
`U32`	32-bit unsigned fixed-point number.

Sd	is a single-precision register for the operand and result.
Dd	is a double-precision register for the operand and result.
fbits	is the number of fraction bits in the fixed-point number, in the range 0-16 if *type* is `S16` or `U16`, or in the range 1-32 if *type* is `S32` or `U32`.

Usage

The first two forms of this instruction convert from floating-point to fixed-point.

The third and fourth forms convert from fixed-point to floating-point.

In all cases the fixed-point number is contained in the least significant 16 or 32 bits of the register.

Floating-point exceptions

These instructions can produce Input Denormal, Invalid Operation, or Inexact exceptions.

See also

Reference

- *Condition codes* on page 3-163.

4.10.9 VCVTB, VCVTT (half-precision extension)

Converts between half-precision and single-precision floating-point numbers in the following ways:

- `VCVTB` uses the bottom half (bits[15:0]) of the single word register to obtain or store the half-precision value
- `VCVTT` uses the top half (bits[31:16]) of the single word register to obtain or store the half-precision value.

`VCVTB` and `VCVTT` are always scalar.

Syntax

`VCVTB{`*`cond`*`}.`*`type Sd, Sm`*

`VCVTT{`*`cond`*`}.`*`type Sd, Sm`*

where:

cond	is an optional condition code.
type	can be any one of:
	`F32.F16` convert from half-precision to single-precision
	`F16.F32` convert form single-precision to half-precision.
Sd	is a single word register for the result.
Sm	is a single word register for the operand.

Architectures

The instructions are only available in VFPv3 systems with the half-precision extension.

Floating-point exceptions

These instructions can produce Input Denormal, Invalid Operation, Overflow, Underflow, or Inexact exceptions.

See also

Reference

- *Condition codes* on page 3-163.

4.10.10 VMOV

Insert a floating-point immediate value in a single-precision or double-precision register, or copy one register into another register.

This instruction is always scalar.

Syntax

```
VMOV{cond}.F32 Sd, #imm
```

```
VMOV{cond}.F64 Dd, #imm
```

```
VMOV{cond}.F32 Sd, Sm
```

```
VMOV{cond}.F64 Dd, Dm
```

where:

cond	is an optional condition code.
Sd	is the single-precision destination register.
Dd	is the double-precision destination register.
imm	is the floating-point immediate value.
Sm	is the single-precision source register.
Dm	is the double-precision source register.

Immediate values

Any number that can be expressed as $+/–n * 2^{–r}$,where n and r are integers, $16 <= n <= 31$, $0 <= r <= 7$.

Architectures

The instructions that copy immediate constants are available in VFPv3.

The instructions that copy from register are available on all VFP systems.

See also

Reference

- *Condition codes* on page 3-163.

Chapter 5
Wireless MMX Technology Instructions

The following topics describe support for Wireless MMX Technology instructions:

5.1 Introduction

Marvell Wireless MMX Technology is a set of *Single Instruction Multiple Data* (SIMD) instructions available on selected XScale processors that improve the performance of some multimedia applications. Wireless MMX Technology uses 64-bit registers to enable it to operate on multiple data elements in a packed format.

The assembler supports Marvell Wireless MMX Technology instructions to assemble code to run on the PXA270 processor. This processor implements ARMv5TE architecture, with MMX extensions. Wireless MMX Technology uses ARM coprocessors 0 and 1 to support its instruction set and data types. ARM Compiler toolchain supports Wireless MMX Technology Control and *Single Instruction Multiple Data* (SIMD) Data registers, and include new directives for Wireless MMX Technology development. There is also enhanced support for load and store instructions.

When using the assembler, be aware that:

- Wireless MMX Technology instructions are only assembled if you specify the supported processor (`armasm --device PXA270`).
- The PXA270 processor supports code written in ARM or Thumb only.
- Most Wireless MMX Technology instructions can be executed conditionally, depending on the state of the ARM flags. The Wireless MMX Technology condition codes are identical to the ARM condition codes.

Wireless MMX 2 Technology is an upgraded version of Wireless MMX Technology.

This contains information on the Wireless MMX Technology support provided by the assembler in the ARM Compiler toolchain. It does not provide a detailed description of the Wireless MMX Technology. *Wireless MMX Technology Developer Guide* contains information about the programmers' model and a full description of the Wireless MMX Technology instruction set.

See also

Reference

- *Wireless MMX Technology Developer Guide*.

5.2 ARM support for Wireless MMX technology

This section gives information on the assembler support for Wireless MMX and MMX 2 Technology. It describes:

- *Directives, WRN and WCN*
- *Frame directives*
- *Wireless MMX load and store instructions* on page 5-4
- *Wireless MMX Technology and XScale instructions* on page 5-5.

5.2.1 Directives, WRN and WCN

Directives are available to support Wireless MMX Technology:

WCN Defines a name for a specified Control register, for example:

```
speed WCN  wcgr0 ; defines speed as a symbol for control reg 0
```

WRN Defines a name for a specified SIMD Data register, for example:

```
rate WRN  wr6 ; defines rate as a symbol for data reg 6
```

Avoid conflicting uses of the same register under different names. Do not use any of the predefined register and coprocessor names.

5.2.2 Frame directives

Wireless MMX Technology registers can be used with `FRAME` directives in the usual way to add debug information into your object files. Be aware of the following restrictions:

- A warning is given if you try to push Wireless MMX Technology registers `wR0` - `wR9` or `wCGR0` - `wCGR3` onto the stack.
- Wireless MMX Technology registers cannot be used as address offsets.

5.2.3 Wireless MMX load and store instructions

Load and store byte, halfword, word or doublewords to and from Wireless MMX coprocessor registers.

Syntax

```
op<type>{cond} wRd, [Rn, #{-}offset]{!}

op<type>{cond} wRd, [Rn] {, #{-}offset}

opW{cond} wRd, label

opW wCd, [Rn, #{-}offset]{!}

opW wCd, [Rn] {, #{-}offset}

opD{cond} wRd, label

opD wRd, [Rn, {-}Rm {, LSL #imm4}]{!}      ; MMX2 only

opD wRd, [Rn], {-}Rm {, LSL #imm4}         ; MMX2 only
```

where:

op	can be either: `WLDR` Load Wireless MMX Register `WSTR` Store Wireless MMX Register.
<type>	can be any one of: `B` Byte `H` Halfword `W` Word `D` Doubleword.
cond	is an optional condition code.
wRd	is the Wireless MMX SIMD data register to load or save.
wCd	is the Wireless MMX Status and Control register to load or save.
Rn	is the register on which the memory address is based.
offset	is an immediate offset. If offset is omitted, the instruction is a zero offset instruction.
!	is an optional suffix. If ! is present, the instruction is a pre-indexed instruction.
label	is a PC-relative expression. *label* must be within +/- 1020 bytes of the current instruction.
Rm	is a register containing a value to be used as the offset. *Rm* must not be PC.
imm4	contains the number of bits to shift *Rm* left, in the range 0-15.

Loading constants into SIMD registers

The assembler also supports the `WLDRW` and `WLDRD` literal load pseudo-instructions, for example:

```
    WLDRW wr0, =0x114
```

Be aware that:

- The assembler cannot load byte and halfword literals. These produce a downgradable error. If downgraded, the instruction is converted to a `WLDRW` and a 32-bit literal is generated. This is the same as a byte literal load, but uses a 32-bit word instead.
- If the literal to be loaded is zero, and the destination is a SIMD Data register, the assembler converts the instruction to a `WZERO`.
- Doubleword loads that are not 8-byte aligned are UNPREDICTABLE.

5.2.4 Wireless MMX Technology and XScale instructions

Wireless MMX Technology instructions overlap with XScale instructions. To avoid conflicts, the assembler has the following restrictions:

- You cannot mix the XScale instructions with Wireless MMX Technology instructions in the same assembly.
- Wireless MMX Technology `TMIA` instructions have a `MIA` mnemonic that overlaps with the XScale `MIA` instructions. Be aware that:
  — `MIA acc0, *Rm*, *Rs*` is accepted in XScale, but faulted in Wireless MMX Technology.
  — `MIA wR0, *Rm*, *Rs*` and `TMIA wR0, *Rm*, *Rs*` are accepted in Wireless MMX Technology.
  — `TMIA acc0, *Rm*, *Rs*` is faulted in XScale (XScale has no `TMIA` instruction).

See also

Concepts

Using the Assembler:

- *Predeclared XScale register names* on page 3-15
- *Register-relative and PC-relative expressions* on page 8-7.

Reference

- *Condition codes* on page 3-163
- *MIA, MIAPH, and MIAxy* on page 3-95
- *MAR and MRA* on page 3-150
- *Frame directives* on page 6-37
- *FRAME PUSH* on page 6-40
- *FRAME ADDRESS* on page 6-38
- *FRAME RETURN ADDRESS* on page 6-43.

5.3 Wireless MMX instructions

Table 5-1 gives a list of the Wireless MMX Technology instruction set. The instructions are described in *Wireless MMX Technology Developer Guide*. Wireless MMX Technology registers are indicated by `wRn`, `wRd`, and ARM registers are shown as `Rn`, `Rd`.

Table 5-1 Wireless MMX Technology instructions

Mnemonic	Example
`TANDC`	`TANDCB r15`
`TBCST`	`TBCSTB wr15, r1`
`TEXTRC`	`TEXTRCB r15, #0`
`TEXTRM`	`TEXTRMUBCS r3, wr7, #7`
`TINSR`	`TINSRB wr6, r11, #0`

`TMIA, TMIAPH,`
`TMIAxy`
	`TMIANE wr1, r2, r3`
	`TMIAPH wr4, r5, r6`
	`TMIABB wr4, r5, r6`
	`MIAPHNE wr4, r5, r6`

`TMOVMSK`	`TMOVMSKBNE r14, wr15`
`TORC`	`TORCB r15`
`WACC`	`WACCBGE wr1, wr2`
`WADD`	`WADDBGE wr1, wr2, wr13`

`WALIGNI, WALIGNR`	`WALIGNI wr7, wr6, wr5,#3`
	`WALIGNR0 wr4, wr8, wr12`

`WAND, WANDN`	`WAND wr1, wr2, wr3`
	`WANDN wr5, wr5, wr9`

`WAVG2`	`WAVG2B wr3, wr6, wr9`
	`WAVG2BR wr4, wr7, wr10`

`WCMPEQ`	`WCMPEQB wr0, wr4, wr2`
`WCMPGT`	`WCMPGTUB wr0, wr4, wr2`
`WLDR`	`WLDRB wr1, [r2, #0]`
`WMAC`	`WMACU wr3, wr4, wr5`
`WMADD`	`WMADDU wr3, wr4, wr5`

`WMAX, WMIN`	`WMAXUB wr0, wr4, wr2`
	`WMINSB wr0, wr4, wr2`

`WMUL`	`WMULUL wr4, wr2, wr3`
`WOR`	`WOR wr3, wr1, wr4`
`WPACK`	`WPACKHUS wr2, wr7, wr1`
`WROR`	`WRORH wr3, wr1, wr4`
`WSAD`	`WSADB wr3, wr5, wr8`
`WSHUFH`	`WSHUFH wr8, wr15, #17`

Table 5-1 Wireless MMX Technology instructions (continued)

Mnemonic	Example

`WSLL, WSRL`	`WSLLH   wr3, wr1, wr4`
	`WSRLHG  wr3, wr1, wcgr0`

`WSRA`	`WSRAH   wr3, wr1, wr4`
	`WSRAHG  wr3, wr1, wcgr0`

`WSTR`	`WSTRB   wr1, [r2, #0]`
	`WSTRW wc1, [r2, #0]`

`WSUB`	`WSUBBGE wr1, wr2, wr13`

`WUNPCKEH, WUNPCKEL`	`WUNPCKEHUB wr0, wr4`
	`WUNPCKELSB wr0, wr4`

`WUNPCKIH, WUNPCKIL`	`WUNPCKIHB wr0, wr4, wr2`
	`WUNPCKILH wr1, wr5, wr3`

`WXOR`	`WXOR    wr3, wr1, wr4`

See also

Reference

- *Wireless MMX pseudo-instructions* on page 5-8
- *Wireless MMX Technology Developer Guide*.

5.4 Wireless MMX pseudo-instructions

Table 5-2 gives an overview of the Wireless MMX Technology pseudo-instructions. These instructions are described in the *Wireless MMX Technology Developer Guide*.

Table 5-2 Wireless MMX Technology pseudo-instructions

Mnemonic	Brief description	Example
`TMCR`	Moves the contents of source register, *Rn*, to Control register, *wCn*. Maps onto the ARM `MCR` coprocessor instruction (page 3-127).	`TMCR wc1, r10`
`TMCRR`	Moves the contents of two source registers, *RnLo* and *RnHi*, to destination register, *wRd*. Do not use `R15` for either *RnLo* or *RnHi*. Maps onto the ARM `MCRR` coprocessor instruction (page 3-127).	`TMCRR wr4, r5, r6`
`TMRC`	Moves the contents of Control register, *wCn*, to destination register, *Rd*. Do not use `R15` for *Rd*. Maps onto the ARM `MRC` coprocessor instruction (page 3-128).	`TMRC r1, wc2`
`TMRRC`	Moves the contents of source register, *wRn*, to two destination registers, *RdLo* and *RdHi*. Do not use `R15` for either destination register. *RdLo* and *RdHi* must be distinct registers, otherwise the result is UNPREDICTABLE. Maps onto the ARM `MRRC` coprocessor instruction (page 3-128).	`TMRRC r1, r0, wr2`
`WMOV`	Moves the contents of source register, *wRn*, to destination register, *wRd*. This instruction is a form of `WOR` (see Table 5-1 on page 5-6).	`WMOV wr1, wr8`
`WZERO`	Zeros destination register, *wRd*. This instruction is a form of `WANDN` (see Table 5-1 on page 5-6).	`WZERO wr1`

See also

Reference

- *Wireless MMX Technology Developer Guide*
- Chapter 3 *ARM and Thumb Instructions*.

Chapter 6
Directives Reference

The following topics describe the directives that are provided by the ARM assembler, `armasm`:

——— **Note** ———

None of these directives are available in the inline assemblers in the ARM C and C++ compilers.

————————

6.1 Alphabetical list of directives

Table 6-1 shows a complete list of the directives. Use it to locate individual directives.

Table 6-1 Location of directives

Directive	See	Directive	See	Directive	See
`ALIAS`	page 6-58	`EQU`	page 6-66	`LTORG`	page 6-16
`ALIGN`	page 6-59	`EXPORT` *or* `GLOBAL`	page 6-67	`MACRO` *and* `MEND`	page 6-30
`ARM` *and* `CODE32`	page 6-56	`EXPORTAS`	page 6-69	`MAP`	page 6-17
`AREA`	page 6-61	`EXTERN`	page 6-71	`MEND` *see* `MACRO`	page 6-30
`ASSERT`	page 6-50	`FIELD`	page 6-18	`MEXIT`	page 6-33
`ATTR`	page 6-64	`FRAME ADDRESS`	page 6-38	`NOFP`	page 6-75
`CN`	page 6-11	`FRAME POP`	page 6-39	`OPT`	page 6-52
`CODE16`	page 6-56	`FRAME PUSH`	page 6-40	`PRESERVE8` *see* `REQUIRE8`	page 6-76
`COMMON`	page 6-28	`FRAME REGISTER`	page 6-41	`PROC` *see* `FUNCTION`	page 6-47
`CP`	page 6-12	`FRAME RESTORE`	page 6-42	`QN`	page 6-13
`DATA`	page 6-28	`FRAME SAVE`	page 6-44	`RELOC`	page 6-8
`DCB`	page 6-20	`FRAME STATE REMEMBER`	page 6-45	`REQUIRE`	page 6-75
`DCD` *and* `DCDU`	page 6-21	`FRAME STATE RESTORE`	page 6-46	`REQUIRE8` *and* `PRESERVE8`	page 6-76
`DCDO`	page 6-22	`FRAME UNWIND ON` *or* `OFF`	page 6-47	`RLIST`	page 6-10
`DCFD` *and* `DCFDU`	page 6-23	`FUNCTION` *or* `PROC`	page 6-47	`RN`	page 6-9
`DCFS` *and* `DCFSU`	page 6-24	`GBLA`, `GBLL`, *and* `GBLS`	page 6-4	`ROUT`	page 6-77
`DCI`	page 6-25	`GET` *or* `INCLUDE`	page 6-70	`SETA`, `SETL`, *and* `SETS`	page 6-7
`DCQ` *and* `DCQU`	page 6-26	`GLOBAL` *see* `EXPORT`	page 6-67	`SN`	page 6-13
`DCW` *and* `DCWU`	page 6-27	`IF`, `ELSE`, `ENDIF`, *and* `ELIF`	page 6-34	`SPACE` or `FILL`	page 6-19
`DN`	page 6-13	`IMPORT`	page 6-71	`SUBT`	page 6-54
`ELIF`, `ELSE` *see* `IF`	page 6-34	`INCBIN`	page 6-73	`THUMB`	page 6-56
`END`	page 6-65	`INCLUDE` *see* `GET`	page 6-70	`THUMBX`	page 6-56
`ENDFUNC` *or* `ENDP`	page 6-49	`INFO`	page 6-51	`TTL`	page 6-54
`ENDIF` *see* `IF`	page 6-34	`KEEP`	page 6-74	`WHILE` *and* `WEND`	page 6-36
`ENTRY`	page 6-65	`LCLA`, `LCLL`, *and* `LCLS`	page 6-6		

6.2 Symbol definition directives

This section describes the following directives:

- *GBLA, GBLL, and GBLS* on page 6-4

  Declare a global arithmetic, logical, or string variable.
- *LCLA, LCLL, and LCLS* on page 6-6

  Declare a local arithmetic, logical, or string variable.
- *SETA, SETL, and SETS* on page 6-7

  Set the value of an arithmetic, logical, or string variable.
- *RELOC* on page 6-8

  Encode an ELF relocation in an object file.
- *RN* on page 6-9

  Define a name for a specified register.
- *RLIST* on page 6-10

  Define a name for a set of general-purpose registers.
- *CN* on page 6-11

  Define a coprocessor register name.
- *CP* on page 6-12

  Define a coprocessor name.
- *QN, DN, and SN* on page 6-13

  Define a double-precision or single-precision VFP register name.

6.2.1 GBLA, GBLL, and GBLS

The `GBLA` directive declares a global arithmetic variable, and initializes its value to 0.

The `GBLL` directive declares a global logical variable, and initializes its value to `{FALSE}`.

The `GBLS` directive declares a global string variable and initializes its value to a null string, "".

Syntax

`<gblx> variable`

where:

`<gblx>`	is one of `GBLA`, `GBLL`, or `GBLS`.
`variable`	is the name of the variable. *`variable`* must be unique among symbols within a source file.

Usage

Using one of these directives for a variable that is already defined re-initializes the variable to the same values given above.

The scope of the variable is limited to the source file that contains it.

Set the value of the variable with a `SETA`, `SETL`, or `SETS` directive.

Global variables can also be set with the `--predefine` assembler command line option.

Examples

Example 6-1 declares a variable `objectsize`, sets the value of `objectsize` to `0xFF`, and then uses it later in a `SPACE` directive.

Example 6-1

```
            GBLA    objectsize    ; declare the variable name
objectsize  SETA    0xFF          ; set its value
            .
            .                     ; other code
            .
            SPACE   objectsize    ; quote the variable
```

Example 6-2 shows how to declare and set a variable when you invoke `armasm`. Use this when you want to set the value of a variable at assembly time. `--pd` is a synonym for `--predefine`.

Example 6-2

```
armasm --predefine "objectsize SETA 0xFF" sourcefile -o objectfile
```

See also

Reference

- *SETA, SETL, and SETS* on page 6-7
- *LCLA, LCLL, and LCLS* on page 6-6
- *Assembler command line options* on page 2-3.

6.2.2 LCLA, LCLL, and LCLS

The LCLA directive declares a local arithmetic variable, and initializes its value to 0.

The LCLL directive declares a local logical variable, and initializes its value to {FALSE}.

The LCLS directive declares a local string variable, and initializes its value to a null string, "".

Syntax

<lclx> variable

where:

<lclx>	is one of LCLA, LCLL, or LCLS.
variable	is the name of the variable. *variable* must be unique within the macro that contains it.

Usage

Using one of these directives for a variable that is already defined re-initializes the variable to the same values given above.

The scope of the variable is limited to a particular instantiation of the macro that contains it.

Set the value of the variable with a SETA, SETL, or SETS directive.

Example

```
        MACRO                           ; Declare a macro
$label  message $a                      ; Macro prototype line
        LCLS    err                     ; Declare local string
                                        ; variable err.
err     SETS    "error no: "            ; Set value of err
$label  ; code
        INFO    0, "err":CC::STR:$a     ; Use string
        MEND
```

See also

Reference

- *SETA, SETL, and SETS* on page 6-7
- *MACRO and MEND* on page 6-30
- *GBLA, GBLL, and GBLS* on page 6-4.

6.2.3 SETA, SETL, and SETS

The `SETA` directive sets the value of a local or global arithmetic variable.

The `SETL` directive sets the value of a local or global logical variable.

The `SETS` directive sets the value of a local or global string variable.

Syntax

`variable <setx> expr`

where:

`<setx>`	is one of `SETA`, `SETL`, or `SETS`.
`variable`	is the name of a variable declared by a `GBLA`, `GBLL`, `GBLS`, `LCLA`, `LCLL`, or `LCLS` directive.
`expr`	is an expression that is: • numeric, for `SETA` • logical, for `SETL` • string, for `SETS`.

Usage

You must declare *`variable`* using a global or local declaration directive before using one of these directives.

You can also predefine variable names on the command line.

Examples

```
                GBLA    VersionNumber
VersionNumber   SETA    21
                GBLL    Debug
Debug           SETL    {TRUE}
                GBLS    VersionString
VersionString   SETS    "Version 1.0"
```

See also

Concepts

Using the Assembler:

- *Numeric expressions* on page 8-16
- *Logical expressions* on page 8-19
- *String expressions* on page 8-14.

Reference

- *Assembler command line options* on page 2-3
- *LCLA, LCLL, and LCLS* on page 6-6
- *GBLA, GBLL, and GBLS* on page 6-4.

6.2.4 RELOC

The `RELOC` directive explicitly encodes an ELF relocation in an object file.

Syntax

`RELOC` *`n`*, *`symbol`*

`RELOC` *`n`*

where:

`n`	must be an integer in the range 0 to 255 or one of the relocation names defined in the Application Binary Interface for the ARM Architecture.
`symbol`	can be any PC-relative label.

Usage

Use `RELOC` *`n`*, *`symbol`* to create a relocation with respect to the address labeled by *`symbol`*.

If used immediately after an ARM or Thumb instruction, `RELOC` results in a relocation at that instruction. If used immediately after a `DCB`, `DCW`, or `DCD`, or any other data generating directive, `RELOC` results in a relocation at the start of the data. Any addend to be applied must be encoded in the instruction or in the data.

If the assembler has already emitted a relocation at that place, the relocation is updated with the details in the `RELOC` directive, for example:

```
DCD     sym2 ; R_ARM_ABS32 to sym32
RELOC   55   ; ... makes it R_ARM_ABS32_NOI
```

`RELOC` is faulted in all other cases, for example, after any non-data generating directive, `LTORG`, `ALIGN`, or as the first thing in an `AREA`.

Use `RELOC` *`n`* to create a relocation with respect to the anonymous symbol, that is, symbol 0 of the symbol table. If you use `RELOC` *`n`* without a preceding assembler generated relocation, the relocation is with respect to the anonymous symbol.

Examples

```
IMPORT  impsym
LDR     r0,[pc,#-8]
RELOC   4, impsym
DCD     0
RELOC   2, sym
DCD     0,1,2,3,4       ; the final word is relocated
RELOC   38,sym2         ; R_ARM_TARGET1
DCD     impsym
RELOC   R_ARM_TARGET1   ; relocation code 38
```

See also

Reference

- Application Binary Interface for the ARM Architecture, `http://infocenter.arm.com/help/topic/com.arm.doc.subset.swdev.abi/index.html`.

6.2.5 RN

The `RN` directive defines a register name for a specified register.

Syntax

```
name RN expr
```

where:

name	is the name to be assigned to the register. *name* cannot be the same as any of the predefined names.
expr	evaluates to a register number from 0 to 15.

Usage

Use `RN` to allocate convenient names to registers, to help you to remember what you use each register for. Be careful to avoid conflicting uses of the same register under different names.

Examples

```
regname     RN  11  ; defines regname for register 11
sqr4        RN  r6  ; defines sqr4 for register 6
```

See also

Reference

Using the Assembler:

- *Predeclared core register names* on page 3-13
- *Predeclared extension register names* on page 3-14
- *Predeclared XScale register names* on page 3-15
- *Predeclared coprocessor names* on page 3-16.

6.2.6 RLIST

The `RLIST` (register list) directive gives a name to a set of general-purpose registers.

Syntax

`name` `RLIST {`*`list-of-registers`*`}`

where:

`name` is the name to be given to the set of registers. *`name`* cannot be the same as any of the predefined names.

`list-of-registers`

is a comma-delimited list of register names and register ranges. The register list must be enclosed in braces.

Usage

Use `RLIST` to give a name to a set of registers to be transferred by the `LDM` or `STM` instructions.

`LDM` and `STM` always put the lowest physical register numbers at the lowest address in memory, regardless of the order they are supplied to the `LDM` or `STM` instruction. If you have defined your own symbolic register names it can be less apparent that a register list is not in increasing register order.

Use the `--diag_warning 1206` assembler option to ensure that the registers in a register list are supplied in increasing register order. If registers are not supplied in increasing register order, a warning is issued.

Example

```
Context RLIST   {r0-r6,r8,r10-r12,pc}
```

See also

Reference

Using the Assembler:
- *Predeclared core register names* on page 3-13
- *Predeclared extension register names* on page 3-14
- *Predeclared XScale register names* on page 3-15
- *Predeclared coprocessor names* on page 3-16.

6.2.7 CN

The `CN` directive defines a name for a coprocessor register.

Syntax

`name` `CN` *`expr`*

where:

`name`	is the name to be defined for the coprocessor register. *`name`* cannot be the same as any of the predefined names.
`expr`	evaluates to a coprocessor register number from 0 to 15.

Usage

Use `CN` to allocate convenient names to registers, to help you remember what you use each register for.

Note

Avoid conflicting uses of the same register under different names.

The names c0 to c15 are predefined.

Example

```
power    CN  6        ; defines power as a symbol for
                      ; coprocessor register 6
```

See also

Reference

Using the Assembler:

- *Predeclared core register names* on page 3-13
- *Predeclared extension register names* on page 3-14
- *Predeclared XScale register names* on page 3-15
- *Predeclared coprocessor names* on page 3-16.

6.2.8 CP

The `CP` directive defines a name for a specified coprocessor. The coprocessor number must be within the range 0 to 15.

Syntax

`name` `CP` *`expr`*

where:

`name`	is the name to be assigned to the coprocessor. *`name`* cannot be the same as any of the predefined names.
`expr`	evaluates to a coprocessor number from 0 to 15.

Usage

Use `CP` to allocate convenient names to coprocessors, to help you to remember what you use each one for.

Note

Avoid conflicting uses of the same coprocessor under different names.

The names p0 to p15 are predefined for coprocessors 0 to 15.

Example

```
dmu    CP  6       ; defines dmu as a symbol for
                   ; coprocessor 6
```

See also

Reference

Using the Assembler:

- *Predeclared core register names* on page 3-13
- *Predeclared extension register names* on page 3-14
- *Predeclared XScale register names* on page 3-15
- *Predeclared coprocessor names* on page 3-16.

6.2.9 QN, DN, and SN

The `QN` directive defines a name for a specified 128-bit extension register.

The `DN` directive defines a name for a specified 64-bit extension register.

The `SN` directive defines a name for a specified single-precision VFP register.

Syntax

name directive expr{.*type*}{[*x*]}

where:

directive	is `QN`, `DN`, or `SN`.
name	is the name to be assigned to the extension register. *name* cannot be the same as any of the predefined names.
expr	Can be: • an expression that evaluates to a number in the range: — 0-15 if you are using DN in VFPv2 or QN in NEON — 0-31 otherwise. • a predefined register name, or a register name that has already been defined in a previous directive.
type	is any NEON or VFP datatype.
[*x*]	is only available for NEON code. [*x*] is a scalar index into a register.

type and [*x*] are *Extended notation*.

Usage

Use `QN`, `DN`, or `SN` to allocate convenient names to extension registers, to help you to remember what you use each one for.

——— **Note** ———

Avoid conflicting uses of the same register under different names.

————————————

You cannot specify a vector length in a `DN` or `SN` directive.

Examples

```
energy  DN  6   ; defines energy as a symbol for
                ; VFP double-precision register 6
mass    SN  16  ; defines mass as a symbol for
                ; VFP single-precision register 16
```

Extended notation examples

```
varA    DN      d1.U16
varB    DN      d2.U16
varC    DN      d3.U16
        VADD    varA,varB,varC       ; VADD.U16 d1,d2,d3
index   DN      d4.U16[0]
result  QN      q5.I32
        VMULL   result,varA,index    ; VMULL.U16 q5,d1,d3[2]
```

See also

Reference

Using the Assembler:

- *Predeclared core register names* on page 3-13
- *Predeclared extension register names* on page 3-14
- *Predeclared XScale register names* on page 3-15
- *Predeclared coprocessor names* on page 3-16
- *Extended notation* on page 9-21
- *Extended notation examples* on page 6-13
- *NEON and VFP data types* on page 9-13
- *VFP directives and vector notation* on page 9-39.

6.3 Data definition directives

This section describes the following directives to allocate memory, define data structures, set initial contents of memory:

- *LTORG* on page 6-16

  Set an origin for a literal pool.
- *MAP* on page 6-17

  Set the origin of a storage map.
- *FIELD* on page 6-18

  Define a field within a storage map.
- *SPACE or FILL* on page 6-19

  Allocate a zeroed block of memory.
- *DCB* on page 6-20

  Allocate bytes of memory, and specify the initial contents.
- *DCD and DCDU* on page 6-21

  Allocate words of memory, and specify the initial contents.
- *DCDO* on page 6-22

  Allocate words of memory, and specify the initial contents as offsets from the static base register.
- *DCFD and DCFDU* on page 6-23

  Allocate doublewords of memory, and specify the initial contents as double-precision floating-point numbers.
- *DCFS and DCFSU* on page 6-24

  Allocate words of memory, and specify the initial contents as single-precision floating-point numbers.
- *DCI* on page 6-25

  Allocate words of memory, and specify the initial contents. Mark the location as code not data.
- *DCQ and DCQU* on page 6-26

  Allocate doublewords of memory, and specify the initial contents as 64-bit integers.
- *DCW and DCWU* on page 6-27

  Allocate halfwords of memory, and specify the initial contents.
- *COMMON* on page 6-28

  Allocate a block of memory at a symbol, and specify the alignment.
- *DATA* on page 6-28

  Mark data within a code section. Obsolete, for backwards compatibility only.

6.3.1 LTORG

The `LTORG` directive instructs the assembler to assemble the current literal pool immediately.

Syntax

`LTORG`

Usage

The assembler assembles the current literal pool at the end of every code section. The end of a code section is determined by the `AREA` directive at the beginning of the following section, or the end of the assembly.

These default literal pools can sometimes be out of range of some `LDR`, `VLDR`, and `WLDR` pseudo-instructions. Use `LTORG` to ensure that a literal pool is assembled within range.

Large programs can require several literal pools. Place `LTORG` directives after unconditional branches or subroutine return instructions so that the processor does not attempt to execute the constants as instructions.

The assembler word-aligns data in literal pools.

Example

```
        AREA    Example, CODE, READONLY
start   BL      func1
func1                           ; function body
        ; code
        LDR     r1,=0x55555555  ; => LDR R1, [pc, #offset to Literal Pool 1]
        ; code
        MOV     pc,lr           ; end function
        LTORG                   ; Literal Pool 1 contains literal &55555555.
data    SPACE   4200            ; Clears 4200 bytes of memory,
                                ; starting at current location.
        END                     ; Default literal pool is empty.
```

See also

Reference

- *LDR pseudo-instruction* on page 3-159
- *VLDR pseudo-instruction* on page 4-70
- *Wireless MMX load and store instructions* on page 5-4.

6.3.2 MAP

The `MAP` directive sets the origin of a storage map to a specified address. The storage-map location counter, `{VAR}`, is set to the same address. `^` is a synonym for `MAP`.

Syntax

`MAP` *`expr`*`{,`*`base-register`*`}`

where:

`expr`
: is a numeric or PC-relative expression:

- If *`base-register`* is not specified, *`expr`* evaluates to the address where the storage map starts. The storage map location counter is set to this address.
- If *`expr`* is PC-relative, you must have defined the label before you use it in the map. The map requires the definition of the label during the first pass of the assembler.

`base-register`
: specifies a register. If *`base-register`* is specified, the address where the storage map starts is the sum of *`expr`*, and the value in *`base-register`* at runtime.

Usage

Use the `MAP` directive in combination with the `FIELD` directive to describe a storage map.

Specify *`base-register`* to define register-relative labels. The base register becomes implicit in all labels defined by following `FIELD` directives, until the next `MAP` directive. The register-relative labels can be used in load and store instructions.

The `MAP` directive can be used any number of times to define multiple storage maps.

The `{VAR}` counter is set to zero before the first `MAP` directive is used.

Examples

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

See also

Concepts

- *How the assembler works* on page 2-4 *in Using the Assembler*
- *Directives that can be omitted in pass 2 of the assembler* on page 2-6 *in Using the Assembler*.

Reference

- *FIELD* on page 6-18.

6.3.3 FIELD

The `FIELD` directive describes space within a storage map that has been defined using the `MAP` directive. `#` is a synonym for `FIELD`.

Syntax

`{label} FIELD expr`

where:

label	is an optional label. If specified, *label* is assigned the value of the storage location counter, `{VAR}`. The storage location counter is then incremented by the value of *expr*.
expr	is an expression that evaluates to the number of bytes to increment the storage counter.

Usage

If a storage map is set by a `MAP` directive that specifies a *base-register*, the base register is implicit in all labels defined by following `FIELD` directives, until the next `MAP` directive. These register-relative labels can be quoted in load and store instructions.

Examples

The following example shows how register-relative labels are defined using the `MAP` and `FIELD` directives.

```
    MAP     0,r9        ; set {VAR} to the address stored in R9
    FIELD   4           ; increment {VAR} by 4 bytes
Lab FIELD   4           ; set Lab to the address [R9 + 4]
                        ; and then increment {VAR} by 4 bytes
    LDR     r0,Lab      ; equivalent to LDR r0,[r9,#4]
```

When using the `MAP` and `FIELD` directives, you must ensure that the values are consistent in both passes. The following example shows a use of `MAP` and `FIELD` that cause inconsistent values for the symbol `x`. In the first pass `sym` is not defined, so `x` is at 0x04+`R9`. In the second pass, `sym` is defined, so `x` is at 0x00+`R0`. This example results in an assembly error.

```
    MAP 0, r0
    if :LNOT: :DEF: sym
      MAP 0, r9
      FIELD 4  ; x is at 0x04+R9 in first pass
    ENDIF
x   FIELD 4    ; x is at 0x00+R0 in second pass
sym LDR r0, x  ; inconsistent values for x results in assembly error
```

See also

Concepts

- *How the assembler works* on page 2-4 *in Using the Assembler*
- *Directives that can be omitted in pass 2 of the assembler* on page 2-6 *in Using the Assembler*.

Reference

- *MAP* on page 6-17.

6.3.4 SPACE or FILL

The `SPACE` directive reserves a zeroed block of memory. `%` is a synonym for `SPACE`.

The `FILL` directive reserves a block of memory to fill with the given value.

Syntax

`{label} SPACE expr`

`{label} FILL expr{,value{,valuesize}}`

where:

label	is an optional label.
expr	evaluates to the number of bytes to fill or zero.
value	evaluates to the value to fill the reserved bytes with. *value* is optional and if omitted, it is 0. *value* must be 0 in a NOINIT area.
valuesize	is the size, in bytes, of *value*. It can be any of 1, 2, or 4. *valuesize* is optional and if omitted, it is 1.

Usage

Use the `ALIGN` directive to align any code following a `SPACE` or `FILL` directive.

Example

```
        AREA    MyData, DATA, READWRITE
data1   SPACE   255       ; defines 255 bytes of zeroed store
data2   FILL    50,0xAB,1 ; defines 50 bytes containing 0xAB
```

See also

Concepts

Using the Assembler:

- *Numeric expressions* on page 8-16.

Reference

- *DCB* on page 6-20
- *DCD and DCDU* on page 6-21
- *DCDO* on page 6-22
- *DCW and DCWU* on page 6-27
- *ALIGN* on page 6-59.

6.3.5 DCB

The DCB directive allocates one or more bytes of memory, and defines the initial runtime contents of the memory. = is a synonym for DCB.

Syntax

```
{label} DCB expr{,expr}...
```

where:

expr is either:

- a numeric expression that evaluates to an integer in the range –128 to 255.
- a quoted string. The characters of the string are loaded into consecutive bytes of store.

Usage

If DCB is followed by an instruction, use an ALIGN directive to ensure that the instruction is aligned.

Example

Unlike C strings, ARM assembler strings are not nul-terminated. You can construct a nul-terminated C string using DCB as follows:

```
C_string   DCB  "C_string",0
```

See also

Concepts

Using the Assembler:

- *Numeric expressions* on page 8-16.

Reference

- *DCD and DCDU* on page 6-21
- *DCQ and DCQU* on page 6-26
- *DCW and DCWU* on page 6-27
- *SPACE or FILL* on page 6-19
- *ALIGN* on page 6-59.

6.3.6 DCD and DCDU

The `DCD` directive allocates one or more words of memory, aligned on four-byte boundaries, and defines the initial runtime contents of the memory.

`&` is a synonym for `DCD`.

`DCDU` is the same, except that the memory alignment is arbitrary.

Syntax

`{label} DCD{U} expr{,expr}`

where:

expr	is either: • a numeric expression. • a PC-relative expression.

Usage

`DCD` inserts up to three bytes of padding before the first defined word, if necessary, to achieve four-byte alignment.

Use `DCDU` if you do not require alignment.

Examples

```
data1   DCD     1,5,20      ; Defines 3 words containing
                            ; decimal values 1, 5, and 20
data2   DCD     mem06 + 4   ; Defines 1 word containing 4 +
                            ; the address of the label mem06
        AREA    MyData, DATA, READWRITE
        DCB     255         ; Now misaligned ...
data3   DCDU    1,5,20      ; Defines 3 words containing
                            ; 1, 5 and 20, not word aligned
```

See also

Concepts

Using the Assembler:

- *Numeric expressions* on page 8-16.

Reference

- *DCB* on page 6-20
- *DCI* on page 6-25
- *DCW and DCWU* on page 6-27
- *DCQ and DCQU* on page 6-26
- *SPACE or FILL* on page 6-19.

6.3.7 DCDO

The DCDO directive allocates one or more words of memory, aligned on four-byte boundaries, and defines the initial runtime contents of the memory as an offset from the *static base register*, sb (R9).

Syntax

`{label} DCDO expr{,expr}...`

where:

expr	is a register-relative expression or label. The base register must be sb.

Usage

Use DCDO to allocate space in memory for static base register relative relocatable addresses.

Example

```
        IMPORT  externsym
        DCDO    externsym   ; 32-bit word relocated by offset of
                            ; externsym from base of SB section.
```

6.3.8 DCFD and DCFDU

The `DCFD` directive allocates memory for word-aligned double-precision floating-point numbers, and defines the initial runtime contents of the memory. Double-precision numbers occupy two words and must be word aligned to be used in arithmetic operations.

`DCFDU` is the same, except that the memory alignment is arbitrary.

Syntax

`{label} DCFD{U} fpliteral{,fpliteral}...`

where:

`fpliteral`	is a double-precision floating-point literal.

Usage

The assembler inserts up to three bytes of padding before the first defined number, if necessary, to achieve four-byte alignment.

Use `DCFDU` if you do not require alignment.

The word order used when converting *`fpliteral`* to internal form is controlled by the floating-point architecture selected. You cannot use `DCFD` or `DCFDU` if you select the `--fpu none` option.

The range for double-precision numbers is:

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

Examples

```
        DCFD    1E308,-4E-100
        DCFDU   10000,-.1,3.1E26
```

See also

Concepts

Using the Assembler:

- *Floating-point literals* on page 8-18.

Reference

- *DCFS and DCFSU* on page 6-24.

6.3.9 DCFS and DCFSU

The `DCFS` directive allocates memory for word-aligned single-precision floating-point numbers, and defines the initial runtime contents of the memory. Single-precision numbers occupy one word and must be word aligned to be used in arithmetic operations.

`DCFSU` is the same, except that the memory alignment is arbitrary.

Syntax

```
{label} DCFS{U} fpliteral{,fpliteral}...
```

where:

`fpliteral`	is a single-precision floating-point literal.

Usage

`DCFS` inserts up to three bytes of padding before the first defined number, if necessary to achieve four-byte alignment.

Use `DCFSU` if you do not require alignment.

The range for single-precision values is:

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

Examples

```
        DCFS    1E3,-4E-9
        DCFSU   1.0,-.1,3.1E6
```

See also

Concepts

Using the Assembler:

- *Floating-point literals* on page 8-18.

Reference

- *DCFD and DCFDU* on page 6-23.

6.3.10 DCI

In ARM code, the DCI directive allocates one or more words of memory, aligned on four-byte boundaries, and defines the initial runtime contents of the memory.

In Thumb code, the DCI directive allocates one or more halfwords of memory, aligned on two-byte boundaries, and defines the initial runtime contents of the memory.

Syntax

```
{label} DCI{.W} expr{,expr}
```

where:

expr	is a numeric expression.
.W	if present, indicates that four bytes must be inserted in Thumb code.

Usage

The DCI directive is very like the DCD or DCW directives, but the location is marked as code instead of data. Use DCI when writing macros for new instructions not supported by the version of the assembler you are using.

In ARM code, DCI inserts up to three bytes of padding before the first defined word, if necessary, to achieve four-byte alignment. In Thumb code, DCI inserts an initial byte of padding, if necessary, to achieve two-byte alignment.

You can use DCI to insert a bit pattern into the instruction stream. For example, use:

```
DCI 0x46c0
```

to insert the Thumb operation MOV r8,r8.

Example macro

```
    MACRO           ; this macro translates newinstr Rd,Rm
                    ; to the appropriate machine code
    newinst     $Rd,$Rm
    DCI         0xe16f0f10 :OR: ($Rd:SHL:12) :OR: $Rm
    MEND
```

32-bit Thumb example

```
    DCI.W   0xf3af8000   ; inserts 32-bit NOP, 2-byte aligned.
```

See also

Concepts

Using the Assembler:

- *Numeric expressions* on page 8-16.

Reference

- *DCD and DCDU* on page 6-21
- *DCW and DCWU* on page 6-27.

6.3.11 DCQ and DCQU

The `DCQ` directive allocates one or more eight-byte blocks of memory, aligned on four-byte boundaries, and defines the initial runtime contents of the memory.

`DCQU` is the same, except that the memory alignment is arbitrary.

Syntax

`{label} DCQ{U} {-}literal{,{-}literal}...`

where:

literal	is a 64-bit numeric literal. The range of numbers permitted is 0 to 2^{64}–1. In addition to the characters normally permitted in a numeric literal, you can prefix *literal* with a minus sign. In this case, the range of numbers permitted is -2^{63} to –1. The result of specifying `-n` is the same as the result of specifying $2^{64}-n$.

Usage

`DCQ` inserts up to three bytes of padding before the first defined eight-byte block, if necessary, to achieve four-byte alignment.

Use `DCQU` if you do not require alignment.

Examples

```
        AREA    MiscData, DATA, READWRITE
data    DCQ     -225,2_101     ; 2_101 means binary 101.
        DCQU    number+4       ; number must already be defined.
```

See also

Concepts

Using the Assembler:
- *Numeric literals* on page 8-17.

Reference
- *DCB* on page 6-20
- *DCD and DCDU* on page 6-21
- *DCW and DCWU* on page 6-27
- *SPACE or FILL* on page 6-19.

6.3.12 DCW and DCWU

The `DCW` directive allocates one or more halfwords of memory, aligned on two-byte boundaries, and defines the initial runtime contents of the memory.

`DCWU` is the same, except that the memory alignment is arbitrary.

Syntax

`{label} DCW{U} expr{,expr}...`

where:

expr	is a numeric expression that evaluates to an integer in the range –32768 to 65535.

Usage

`DCW` inserts a byte of padding before the first defined halfword if necessary to achieve two-byte alignment.

Use `DCWU` if you do not require alignment.

Examples

```
data    DCW     -225,2*number   ; number must already be defined
        DCWU    number+4
```

See also

Concepts

Using the Assembler:

- *Numeric expressions* on page 8-16.

Reference

- *DCB* on page 6-20
- *DCD and DCDU* on page 6-21
- *DCQ and DCQU* on page 6-26
- *SPACE or FILL* on page 6-19.

6.3.13 COMMON

The COMMON directive allocates a block of memory, of the defined size, at the specified symbol. You specify how the memory is aligned. If alignment is omitted, the default alignment is 4. If size is omitted, the default size is 0.

You can access this memory as you would any other memory, but no space is allocated in object files.

Syntax

COMMON *symbol*{,*size*{,*alignment*}} {[*attr*]}

where:

symbol	is the symbol name. The symbol name is case-sensitive.
size	is the number of bytes to reserve.
alignment	is the alignment.
attr	can be any one of:

DYNAMIC	sets the ELF symbol visibility to STV_DEFAULT.
PROTECTED	sets the ELF symbol visibility to STV_PROTECTED.
HIDDEN	sets the ELF symbol visibility to STV_HIDDEN.
INTERNAL	sets the ELF symbol visibility to STV_INTERNAL.

Usage

The linker allocates the required space as zero initialized memory during the link stage. You cannot define, IMPORT or EXTERN a symbol that has already been created by the COMMON directive. In the same way, if a symbol has already been defined or used with the IMPORT or EXTERN directive, you cannot use the same symbol for the COMMON directive.

Example

```
    LDR       r0, =xyz
    COMMON    xyz,255,4   ; defines 255 bytes of ZI store, word-aligned
```

Incorrect examples

```
    COMMON    foo,4,4
    COMMON    bar,4,4
foo DCD       0           ; cannot define label with same name as COMMON
    IMPORT    bar         ; cannot import label with same name as COMMON
```

6.3.14 DATA

The DATA directive is no longer required. It is ignored by the assembler.

6.4 Assembly control directives

This section describes the following directives to control conditional assembly, looping, inclusions, and macros:

- *MACRO and MEND* on page 6-30
- *MEXIT* on page 6-33
- *IF, ELSE, ENDIF, and ELIF* on page 6-34
- *WHILE and WEND* on page 6-36.

6.4.1 Nesting directives

The following structures can be nested to a total depth of 256:

- `MACRO` definitions
- `WHILE...WEND` loops
- `IF...ELSE...ENDIF` conditional structures
- `INCLUDE` file inclusions.

The limit applies to all structures taken together, regardless of how they are nested. The limit is *not* 256 of each type of structure.

6.4.2 MACRO and MEND

The `MACRO` directive marks the start of the definition of a macro. Macro expansion terminates at the `MEND` directive.

Syntax

Two directives are used to define a macro. The syntax is:

```
         MACRO
{$label}  macroname{$cond} {$parameter{,$parameter}...}
         ; code
         MEND
```

where:

$label	is a parameter that is substituted with a symbol given when the macro is invoked. The symbol is usually a label.
macroname	is the name of the macro. It must not begin with an instruction or directive name.
$cond	is a special parameter designed to contain a condition code. Values other than valid condition codes are permitted.
$parameter	is a parameter that is substituted when the macro is invoked. A default value for a parameter can be set using this format: *$parameter="default value"* Double quotes must be used if there are any spaces within, or at either end of, the default value.

Usage

If you start any `WHILE...WEND` loops or `IF...ENDIF` conditions within a macro, they must be closed before the `MEND` directive is reached. You can use `MEXIT` to enable an early exit from a macro, for example, from within a loop.

Within the macro body, parameters such as *$label*, *$parameter* or *$cond* can be used in the same way as other variables. They are given new values each time the macro is invoked. Parameters must begin with `$` to distinguish them from ordinary symbols. Any number of parameters can be used.

$label is optional. It is useful if the macro defines internal labels. It is treated as a parameter to the macro. It does not necessarily represent the first instruction in the macro expansion. The macro defines the locations of any labels.

Use `|` as the argument to use the default value of a parameter. An empty string is used if the argument is omitted.

In a macro that uses several internal labels, it is useful to define each internal label as the base label with a different suffix.

Use a dot between a parameter and following text, or a following parameter, if a space is not required in the expansion. Do not use a dot between preceding text and a parameter.

You can use the *$cond* parameter for condition codes. Use the unary operator `:REVERSE_CC:` to find the inverse condition code, and `:CC_ENCODING:` to find the 4-bit encoding of the condition code.

Macros define the scope of local variables.

Macros can be nested.

Examples

```
 ; macro definition
                MACRO                  ; start macro definition
$label          xmac    $p1,$p2
                ; code
$label.loop1    ; code
                ; code
                BGE     $label.loop1
$label.loop2    ; code
                BL      $p1
                BGT     $label.loop2
                ; code
                ADR     $p2
                ; code
                MEND                   ; end macro definition
 ; macro invocation
abc             xmac    subr1,de       ; invoke macro
                ; code                 ; this is what is
abcloop1        ; code                 ; is produced when
                ; code                 ; the xmac macro is
                BGE     abcloop1       ; expanded
abcloop2        ; code
                BL      subr1
                BGT     abcloop2
                ; code
                ADR     de
                ; code
```

Using a macro to produce assembly-time diagnostics:

```
        MACRO                       ; Macro definition
        diagnose  $param1="default"  ; This macro produces
        INFO      0,"$param1"       ; assembly-time diagnostics
        MEND                        ; (on second assembly pass)
 ; macro expansion
        diagnose           ; Prints blank line at assembly-time
        diagnose "hello"   ; Prints "hello" at assembly-time
        diagnose |         ; Prints "default" at assembly-time
```

——— **Note** ———

When variables are also being passed in as arguments, use of `|` might leave some variables unsubstituted. To workaround this, define the `|` in a `LCLS` or `GBLS` variable and pass this variable as an argument instead of `|`. For example:

```
        MACRO                ; Macro definition
        m2 $a,$b=r1,$c       ; The default value for $b is r1
        add $a,$b,$c         ; The macro adds $b and $c and puts result in $a
        MEND                 ; Macro end

        MACRO                ; Macro definition
        m1 $a,$b             ; This macro adds $b to r1 and puts result in $a
        LCLS def             ; Declare a local string variable for |
def     SETS "|"             ; Define |
        m2 $a,$def,$b        ; Invoke macro m2 with $def instead of |
                             ; to use the default value for the second argument.
        MEND                 ; Macro end
```

Conditional macro example

```
        AREA    codx, CODE, READONLY

; macro definition

        MACRO
        Return$cond
        [ {ARCHITECTURE} <> "4"
          BX$cond lr
          |
          MOV$cond pc,lr
        ]
        MEND

; macro invocation

fun     PROC
        CMP      r0,#0
        MOVEQ    r0,#1
        ReturnEQ
        MOV      r0,#0
        Return
        ENDP

        END
```

See also

Concepts

Using the Assembler:

- *Use of macros* on page 5-30
- *Assembly time substitution of variables* on page 8-6.

Reference

- *MEXIT* on page 6-33
- *Nesting directives* on page 6-29
- *GBLA, GBLL, and GBLS* on page 6-4
- *LCLA, LCLL, and LCLS* on page 6-6.

6.4.3 MEXIT

The `MEXIT` directive is used to exit a macro definition before the end.

Usage

Use `MEXIT` when you require an exit from within the body of a macro. Any unclosed `WHILE...WEND` loops or `IF...ENDIF` conditions within the body of the macro are closed by the assembler before the macro is exited.

Example

```
        MACRO
$abc    example abc     $param1,$param2
        ; code
        WHILE condition1
            ; code
            IF condition2
                ; code
                MEXIT
            ELSE
                ; code
            ENDIF
        WEND
        ; code
        MEND
```

See also

Reference

- *MACRO and MEND* on page 6-30.

6.4.4 IF, ELSE, ENDIF, and ELIF

The `IF` directive introduces a condition that is used to decide whether to assemble a sequence of instructions and directives. `[` is a synonym for `IF`.

The `ELSE` directive marks the beginning of a sequence of instructions or directives that you want to be assembled if the preceding condition fails. `|` is a synonym for `ELSE`.

The `ENDIF` directive marks the end of a sequence of instructions or directives that you want to be conditionally assembled. `]` is a synonym for `ENDIF`.

The `ELIF` directive creates a structure equivalent to `ELSE IF`, without the requirement for nesting or repeating the condition.

Syntax

```
IF logical-expression          …;code
{ELSE          …;code}     ENDIF
```

where:

`logical-expression`

is an expression that evaluates to either `{TRUE}` or `{FALSE}`.

Usage

Use `IF` with `ENDIF`, and optionally with `ELSE`, for sequences of instructions or directives that are only to be assembled or acted on under a specified condition.

`IF...ENDIF` conditions can be nested.

Using ELIF

Without using `ELIF`, you can construct a nested set of conditional instructions like this:

```
IF logical-expression
    instructions
ELSE
    IF logical-expression2
        instructions
    ELSE
        IF logical-expression3
            instructions
        ENDIF
    ENDIF
ENDIF
```

A nested structure like this can be nested up to 256 levels deep.

You can write the same structure more simply using `ELIF`:

```
IF logical-expression
    instructions
ELIF logical-expression2
    instructions
ELIF logical-expression3
    instructions
ENDIF
```

This structure only adds one to the current nesting depth, for the `IF...ENDIF` pair.

Examples

Example 6-3 assembles the first set of instructions if NEWVERSION is defined, or the alternative set otherwise.

Example 6-3 Assembly conditional on a variable being defined

```
        IF :DEF:NEWVERSION
            ; first set of instructions or directives
        ELSE
            ; alternative set of instructions or directives
        ENDIF
```

Invoking armasm as follows defines NEWVERSION, so the first set of instructions and directives are assembled:

```
armasm --predefine "NEWVERSION SETL {TRUE}" test.s
```

Invoking armasm as follows leaves NEWVERSION undefined, so the second set of instructions and directives are assembled:

```
armasm test.s
```

Example 6-4 assembles the first set of instructions if NEWVERSION has the value {TRUE}, or the alternative set otherwise.

Example 6-4 Assembly conditional on a variable value

```
        IF NEWVERSION = {TRUE}
            ; first set of instructions or directives
        ELSE
            ; alternative set of instructions or directives
        ENDIF
```

Invoking armasm as follows causes the first set of instructions and directives to be assembled:

```
armasm --predefine "NEWVERSION SETL {TRUE}" test.s
```

Invoking armasm as follows causes the second set of instructions and directives to be assembled:

```
armasm --predefine "NEWVERSION SETL {FALSE}" test.s
```

See also

Concepts

Using the Assembler:

- *Relational operators* on page 8-27.

Reference

- *Using ELIF* on page 6-34
- *Nesting directives* on page 6-29.

6.4.5 WHILE and WEND

The `WHILE` directive starts a sequence of instructions or directives that are to be assembled repeatedly. The sequence is terminated with a `WEND` directive.

Syntax

`WHILE` *`logical-expression`*

`code`

`WEND`

where:

`logical-expression`

is an expression that can evaluate to either `{TRUE}` or `{FALSE}`.

Usage

Use the `WHILE` directive, together with the `WEND` directive, to assemble a sequence of instructions a number of times. The number of repetitions can be zero.

You can use `IF...ENDIF` conditions within `WHILE...WEND` loops.

`WHILE...WEND` loops can be nested.

Example

```
        GBLA count                  ; declare local variable
count   SETA    1                   ; you are not restricted to
        WHILE   count <= 4          ; such simple conditions
count   SETA    count+1             ; In this case,
            ; code                  ; this code will be
            ; code                  ; repeated four times
        WEND
```

See also

Concepts

Using the Assembler:

- *Logical expressions* on page 8-19.

Reference

- *Nesting directives* on page 6-29.

6.5 Frame directives

This section describes the following directives:

Correct use of these directives:

- enables the `armlink --callgraph` option to calculate stack usage of assembler functions.

  The following rules are used to determine stack usage:
  - If a function is not marked with `PROC` or `ENDP`, stack usage is unknown.
  - If a function is marked with `PROC` or `ENDP` but with no `FRAME PUSH` or `FRAME POP`, stack usage is assumed to be zero. This means that there is no requirement to manually add `FRAME PUSH 0` or `FRAME POP 0`.
  - If a function is marked with `PROC` or `ENDP` and with `FRAME PUSH n` or `FRAME POP n`, stack usage is assumed to be `n` bytes.
- helps you to avoid errors in function construction, particularly when you are modifying existing code
- enables the assembler to alert you to errors in function construction
- enables backtracing of function calls during debugging
- enables the debugger to profile assembler functions.

If you require profiling of assembler functions, but do not want frame description directives for other purposes:

- you must use the `FUNCTION` and `ENDFUNC`, or `PROC` and `ENDP`, directives
- you can omit the other `FRAME` directives
- you only have to use the `FUNCTION` and `ENDFUNC` directives for the functions you want to profile.

In DWARF, the canonical frame address is an address on the stack specifying where the call frame of an interrupted function is located.

6.5.1 FRAME ADDRESS

The `FRAME ADDRESS` directive describes how to calculate the canonical frame address for following instructions. You can only use it in functions with `FUNCTION` and `ENDFUNC` or `PROC` and `ENDP` directives.

Syntax

`FRAME ADDRESS` *`reg`*`[,`*`offset`*`]`

where:

`reg` is the register on which the canonical frame address is to be based. This is SP unless the function uses a separate frame pointer.

`offset` is the offset of the canonical frame address from *`reg`*. If *`offset`* is zero, you can omit it.

Usage

Use `FRAME ADDRESS` if your code alters which register the canonical frame address is based on, or if it changes the offset of the canonical frame address from the register. You must use `FRAME ADDRESS` immediately after the instruction that changes the calculation of the canonical frame address.

Note

If your code uses a single instruction to save registers and alter the stack pointer, you can use `FRAME PUSH` instead of using both `FRAME ADDRESS` and `FRAME SAVE`.

If your code uses a single instruction to load registers and alter the stack pointer, you can use `FRAME POP` instead of using both `FRAME ADDRESS` and `FRAME RESTORE`.

Example

```
_fn     FUNCTION        ; CFA (Canonical Frame Address) is value
                        ; of SP on entry to function
        PUSH    {r4,fp,ip,lr,pc}
        FRAME PUSH {r4,fp,ip,lr,pc}
        SUB     sp,sp,#4            ; CFA offset now changed
        FRAME ADDRESS sp,24         ; - so we correct it
        ADD     fp,sp,#20
        FRAME ADDRESS fp,4          ; New base register
        ; code using fp to base call-frame on, instead of SP
```

See also

Reference

- *FRAME POP* on page 6-39
- *FRAME PUSH* on page 6-40.

6.5.2 FRAME POP

Use the `FRAME POP` directive to inform the assembler when the callee reloads registers. You can only use it within functions with `FUNCTION` and `ENDFUNC` or `PROC` and `ENDP` directives.

You do not have to do this after the last instruction in a function.

Syntax

There are three alternative syntaxes for `FRAME POP`:

```
FRAME POP {reglist}
```

```
FRAME POP {reglist},n
```

```
FRAME POP n
```

where:

reglist	is a list of registers restored to the values they had on entry to the function. There must be at least one register in the list.
n	is the number of bytes that the stack pointer moves.

Usage

`FRAME POP` is equivalent to a `FRAME ADDRESS` and a `FRAME RESTORE` directive. You can use it when a single instruction loads registers and alters the stack pointer.

You must use `FRAME POP` immediately after the instruction it refers to.

If *n* is not specified or is zero, the assembler calculates the new offset for the canonical frame address from {*reglist*}. It assumes that:

- each ARM register popped occupies four bytes on the stack
- each VFP single-precision register popped occupies four bytes on the stack, plus an extra four-byte word for each list
- each VFP double-precision register popped occupies eight bytes on the stack, plus an extra four-byte word for each list.

See also

Reference

- *FRAME ADDRESS* on page 6-38
- *FRAME RESTORE* on page 6-42.

6.5.3 FRAME PUSH

Use the FRAME PUSH directive to inform the assembler when the callee saves registers, normally at function entry. You can only use it within functions with FUNCTION and ENDFUNC or PROC and ENDP directives.

Syntax

There are two alternative syntaxes for FRAME PUSH:

FRAME PUSH {*reglist*}

FRAME PUSH {*reglist*},*n*

FRAME PUSH *n*

where:

reglist	is a list of registers stored consecutively below the canonical frame address. There must be at least one register in the list.
n	is the number of bytes that the stack pointer moves.

Usage

FRAME PUSH is equivalent to a FRAME ADDRESS and a FRAME SAVE directive. You can use it when a single instruction saves registers and alters the stack pointer.

You must use FRAME PUSH immediately after the instruction it refers to.

If *n* is not specified or is zero, the assembler calculates the new offset for the canonical frame address from {*reglist*}. It assumes that:

- each ARM register pushed occupies four bytes on the stack
- each VFP single-precision register pushed occupies four bytes on the stack, plus an extra four-byte word for each list
- each VFP double-precision register popped occupies eight bytes on the stack, plus an extra four-byte word for each list.

Example

```
p   PROC ; Canonical frame address is SP + 0
    EXPORT  p
    PUSH    {r4-r6,lr}
         ; SP has moved relative to the canonical frame address,
         ; and registers R4, R5, R6 and LR are now on the stack
    FRAME PUSH {r4-r6,lr}
         ; Equivalent to:
         ; FRAME ADDRESS    sp,16       ; 16 bytes in {R4-R6,LR}
         ; FRAME SAVE    {r4-r6,lr},-16
```

See also

Reference

- *FRAME ADDRESS* on page 6-38
- *FRAME SAVE* on page 6-44.

6.5.4 FRAME REGISTER

Use the `FRAME REGISTER` directive to maintain a record of the locations of function arguments held in registers. You can only use it within functions with `FUNCTION` and `ENDFUNC` or `PROC` and `ENDP` directives.

Syntax

```
FRAME REGISTER reg1,
   reg2
```

where:

reg1	is the register that held the argument on entry to the function.
reg2	is the register in which the value is preserved.

Usage

Use the `FRAME REGISTER` directive when you use a register to preserve an argument that was held in a different register on entry to a function.

6.5.5 FRAME RESTORE

Use the `FRAME RESTORE` directive to inform the assembler that the contents of specified registers have been restored to the values they had on entry to the function. You can only use it within functions with `FUNCTION` and `ENDFUNC` or `PROC` and `ENDP` directives.

Syntax

`FRAME RESTORE {`*`reglist`*`}`

where:

`reglist`	is a list of registers whose contents have been restored. There must be at least one register in the list.

Usage

Use `FRAME RESTORE` immediately after the callee reloads registers from the stack. You do not have to do this after the last instruction in a function.

`reglist` can contain integer registers or floating-point registers, but not both.

——— Note ———

If your code uses a single instruction to load registers and alter the stack pointer, you can use `FRAME POP` instead of using both `FRAME RESTORE` and `FRAME ADDRESS`.

————————————

See also

Reference

- *FRAME POP* on page 6-39.

6.5.6 FRAME RETURN ADDRESS

The `FRAME RETURN ADDRESS` directive provides for functions that use a register other than LR for their return address. You can only use it within functions with `FUNCTION` and `ENDFUNC` or `PROC` and `ENDP` directives.

Note

Any function that uses a register other than LR for its return address is not AAPCS compliant. Such a function must not be exported.

Syntax

```
FRAME RETURN ADDRESS reg
```

where:

reg	is the register used for the return address.

Usage

Use the `FRAME RETURN ADDRESS` directive in any function that does not use LR for its return address. Otherwise, a debugger cannot backtrace through the function.

Use `FRAME RETURN ADDRESS` immediately after the `FUNCTION` or `PROC` directive that introduces the function.

6.5.7 FRAME SAVE

The `FRAME SAVE` directive describes the location of saved register contents relative to the canonical frame address. You can only use it within functions with `FUNCTION` and `ENDFUNC` or `PROC` and `ENDP` directives.

Syntax

```
FRAME SAVE {reglist}, offset
```

where:

`reglist`	is a list of registers stored consecutively starting at *`offset`* from the canonical frame address. There must be at least one register in the list.

Usage

Use `FRAME SAVE` immediately after the callee stores registers onto the stack.

`reglist` can include registers which are not required for backtracing. The assembler determines which registers it requires to record in the DWARF call frame information.

Note

If your code uses a single instruction to save registers and alter the stack pointer, you can use `FRAME PUSH` instead of using both `FRAME SAVE` and `FRAME ADDRESS`.

See also

Reference

- *FRAME PUSH* on page 6-40.

6.5.8 FRAME STATE REMEMBER

The FRAME STATE REMEMBER directive saves the current information on how to calculate the canonical frame address and locations of saved register values. You can only use it within functions with FUNCTION and ENDFUNC or PROC and ENDP directives.

Syntax

```
FRAME STATE REMEMBER
```

Usage

During an inline exit sequence the information about calculation of canonical frame address and locations of saved register values can change. After the exit sequence another branch can continue using the same information as before. Use FRAME STATE REMEMBER to preserve this information, and FRAME STATE RESTORE to restore it.

These directives can be nested. Each FRAME STATE RESTORE directive must have a corresponding FRAME STATE REMEMBER directive.

Example

```
        ; function code
        FRAME STATE REMEMBER
            ; save frame state before in-line exit sequence
        POP     {r4-r6,pc}
            ; do not have to FRAME POP here, as control has
            ; transferred out of the function
        FRAME STATE RESTORE
            ; end of exit sequence, so restore state
exitB   ; code for exitB
        POP     {r4-r6,pc}
        ENDP
```

See also

Reference

- *FRAME STATE RESTORE* on page 6-46
- *FUNCTION or PROC* on page 6-47.

6.5.9 FRAME STATE RESTORE

The `FRAME STATE RESTORE` directive restores information about how to calculate the canonical frame address and locations of saved register values. You can only use it within functions with `FUNCTION` and `ENDFUNC` or `PROC` and `ENDP` directives.

Syntax

```
FRAME STATE RESTORE
```

See also

Reference

- *FRAME STATE REMEMBER* on page 6-45
- *FUNCTION or PROC* on page 6-47.

6.5.10 FRAME UNWIND ON

The `FRAME UNWIND ON` directive instructs the assembler to produce *unwind* tables for this and subsequent functions.

Syntax

```
FRAME UNWIND ON
```

Usage

You can use this directive outside functions. In this case, the assembler produces *unwind* tables for all following functions until it reaches a `FRAME UNWIND OFF` directive.

Note

A `FRAME UNWIND` directive is not sufficient to turn on exception table generation. Furthermore a `FRAME UNWIND` directive, without other `FRAME` directives, is not sufficient information for the assembler to generate the *unwind* information.

See also

Reference

- *--exceptions* on page 2-13
- *--exceptions_unwind* on page 2-13.

6.5.11 FRAME UNWIND OFF

The `FRAME UNWIND OFF` directive instructs the assembler to produce *nounwind* tables for this and subsequent functions.

Syntax

```
FRAME UNWIND OFF
```

Usage

You can use this directive outside functions. In this case, the assembler produces *nounwind* tables for all following functions until it reaches a `FRAME UNWIND ON` directive.

See also

Reference

- *--exceptions* on page 2-13
- *--exceptions_unwind* on page 2-13.

6.5.12 FUNCTION or PROC

The `FUNCTION` directive marks the start of a function. `PROC` is a synonym for `FUNCTION`.

Syntax

```
label FUNCTION [{reglist1} [, {reglist2}]]
```

where:

`reglist1`	is an optional list of callee-saved ARM registers. If `reglist1` is not present, and your debugger checks register usage, it will assume that the AAPCS is in use. If you use empty brackets, this informs the debugger that all ARM registers are caller-saved.
`reglist2`	is an optional list of callee-saved VFP registers. If you use empty brackets, this informs the debugger that all VFP registers are caller-saved.

Usage

Use `FUNCTION` to mark the start of functions. The assembler uses `FUNCTION` to identify the start of a function when producing DWARF call frame information for ELF.

`FUNCTION` sets the canonical frame address to be R13 (SP), and the frame state stack to be empty.

Each `FUNCTION` directive must have a matching `ENDFUNC` directive. You must not nest `FUNCTION` and `ENDFUNC` pairs, and they must not contain `PROC` or `ENDP` directives.

You can use the optional *`reglist`* parameters to inform the debugger about an alternative procedure call standard, if you are using your own. Not all debuggers support this feature. See your debugger documentation for details.

If you specify an empty *`reglist`*, using {}, this indicates that all registers for the function are caller-saved. Typically you do this when writing a reset vector where the values in all registers are unknown on execution. This avoids problems in a debugger if it tries to construct a backtrace from the values in the registers.

Note

`FUNCTION` does not automatically cause alignment to a word boundary (or halfword boundary for Thumb). Use `ALIGN` if necessary to ensure alignment, otherwise the call frame might not point to the start of the function.

Examples

```
        ALIGN      ; ensures alignment
dadd    FUNCTION   ; without the ALIGN directive, this might not be word-aligned
        EXPORT  dadd
        PUSH       {r4-r6,lr}    ; this line automatically word-aligned
        FRAME PUSH {r4-r6,lr}
        ; subroutine body
        POP        {r4-r6,pc}
        ENDFUNC
func6   PROC {r4-r8,r12},{D1-D3} ; non-AAPCS-conforming function
        ...
        ENDP
func7   FUNCTION {} ; another non-AAPCS-conforming function
        ...
        ENDFUNC
```

See also

Reference

- *FRAME ADDRESS* on page 6-38
- *FRAME STATE RESTORE* on page 6-46
- *ALIGN* on page 6-59.

6.5.13 ENDFUNC or ENDP

The `ENDFUNC` directive marks the end of an AAPCS-conforming function. `ENDP` is a synonym for `ENDFUNC`.

See also

Reference

- *FUNCTION or PROC* on page 6-47.

6.6 Reporting directives

This section describes the following directives:

- *ASSERT*
  generates an error message if an assertion is false during assembly.
- *INFO* on page 6-51
  generates diagnostic information during assembly.
- *OPT* on page 6-52
  sets listing options.
- *TTL and SUBT* on page 6-54
  insert titles and subtitles in listings.

6.6.1 ASSERT

The `ASSERT` directive generates an error message during assembly if a given assertion is false.

Syntax

`ASSERT` *`logical-expression`*

where:

`logical-expression`

is an assertion that can evaluate to either `{TRUE}` or `{FALSE}`.

Usage

Use `ASSERT` to ensure that any necessary condition is met during assembly.

If the assertion is false an error message is generated and assembly fails.

Example

```
        ASSERT  label1 <= label2    ; Tests if the address
                                    ; represented by label1
                                    ; is <= the address
                                    ; represented by label2.
```

See also

Reference

- *INFO* on page 6-51.

6.6.2 INFO

The `INFO` directive supports diagnostic generation on either pass of the assembly.

! is very similar to `INFO`, but has less detailed reporting.

Syntax

`INFO` *`numeric-expression, string-expression`*`{, `*`severity`*`}`

where:

`numeric-expression`

is a numeric expression that is evaluated during assembly. If the expression evaluates to zero:

- no action is taken during pass one
- *`string-expression`* is printed as a warning during pass two if *`severity`* is 1
- *`string-expression`* is printed as a message during pass two if *`severity`* is 0 or not specified.

If the expression does not evaluate to zero:

- *`string-expression`* is printed as an error message and the assembly fails irrespective of whether *`severity`* is specified or not (non-zero values for *`severity`* are reserved in this case).

`string-expression`

is an expression that evaluates to a string.

`severity`

is an optional number that controls the severity of the message. Its value can be either 0 or 1. All other values are reserved.

Usage

`INFO` provides a flexible means of creating custom error messages.

Examples

```
INFO    0, "Version 1.0"
IF endofdata <= label1
    INFO    4, "Data overrun at label1"
ENDIF
```

See also

Concepts

Using the Assembler:

- *Numeric expressions* on page 8-16
- *String expressions* on page 8-14.

Reference

- *ASSERT* on page 6-50.

6.6.3 OPT

The `OPT` directive sets listing options from within the source code.

Syntax

`OPT` *`n`*

where:

`n`	is the `OPT` directive setting. Table 6-2 lists valid settings.

Table 6-2 OPT directive settings

OPT n	Effect
1	Turns on normal listing.
2	Turns off normal listing.
4	Page throw. Issues an immediate form feed and starts a new page.
8	Resets the line number counter to zero.
16	Turns on listing for `SET`, `GBL` and `LCL` directives.
32	Turns off listing for `SET`, `GBL` and `LCL` directives.
64	Turns on listing of macro expansions.
128	Turns off listing of macro expansions.
256	Turns on listing of macro invocations.
512	Turns off listing of macro invocations.
1024	Turns on the first pass listing.
2048	Turns off the first pass listing.
4096	Turns on listing of conditional directives.
8192	Turns off listing of conditional directives.
16384	Turns on listing of `MEND` directives.
32768	Turns off listing of `MEND` directives.

Usage

Specify the `--list=` assembler option to turn on listing.

By default the `--list=` option produces a normal listing that includes variable declarations, macro expansions, call-conditioned directives, and `MEND` directives. The listing is produced on the second pass only. Use the `OPT` directive to modify the default listing options from within your code.

You can use `OPT` to format code listings. For example, you can specify a new page before functions and sections.

Example

```
        AREA    Example, CODE, READONLY
start   ; code
        ; code
        BL      func1
        ; code
        OPT 4                ; places a page break before func1
func1   ; code
```

See also

Reference

- *--list=file* on page 2-17.

6.6.4 TTL and SUBT

The `TTL` directive inserts a title at the start of each page of a listing file. The title is printed on each page until a new `TTL` directive is issued.

The `SUBT` directive places a subtitle on the pages of a listing file. The subtitle is printed on each page until a new `SUBT` directive is issued.

Syntax

`TTL` *`title`*

`SUBT` *`subtitle`*

where:

`title`	is the title.
`subtitle`	is the subtitle.

Usage

Use the `TTL` directive to place a title at the top of the pages of a listing file. If you want the title to appear on the first page, the `TTL` directive must be on the first line of the source file.

Use additional `TTL` directives to change the title. Each new `TTL` directive takes effect from the top of the next page.

Use `SUBT` to place a subtitle at the top of the pages of a listing file. Subtitles appear in the line below the titles. If you want the subtitle to appear on the first page, the `SUBT` directive must be on the first line of the source file.

Use additional `SUBT` directives to change subtitles. Each new `SUBT` directive takes effect from the top of the next page.

Examples

```
        TTL     First Title     ; places a title on the first
                                ; and subsequent pages of a
                                ; listing file.
        SUBT    First Subtitle  ; places a subtitle on the
                                ; second and subsequent pages
                                ; of a listing file.
```

6.7 Instruction set and syntax selection directives

This section describes the following directives:

- *ARM, THUMB, THUMBX, CODE16 and CODE32* on page 6-56.

6.7.1 ARM, THUMB, THUMBX, CODE16 and CODE32

The `ARM` directive and the `CODE32` directive are synonyms. They instruct the assembler to interpret subsequent instructions as ARM instructions, using either the UAL or the pre-UAL ARM assembler language syntax.

The `THUMB` directive instructs the assembler to interpret subsequent instructions as Thumb instructions, using the UAL syntax.

The `THUMBX` directive instructs the assembler to interpret subsequent instructions as ThumbEE instructions, using the UAL syntax.

The `CODE16` directive instructs the assembler to interpret subsequent instructions as Thumb instructions, using the pre-UAL assembly language syntax.

If necessary, these directives also insert up to three bytes of padding to align to the next word boundary for ARM, or up to one byte of padding to align to the next halfword boundary for Thumb or ThumbEE.

Syntax

```
ARM
THUMB
THUMBX
CODE16
CODE32
```

Usage

In files that contain code using different instruction sets:

- `ARM` must precede any ARM code. `CODE32` is a synonym for `ARM`.
- `THUMB` must precede Thumb code written in UAL syntax.
- `THUMBX` must precede ThumbEE code written in UAL syntax.
- `CODE16` must precede Thumb code written in pre-UAL syntax.

These directives do not assemble to any instructions. They also do not change the state. They only instruct the assembler to assemble ARM, Thumb, or ThumbEE instructions as appropriate, and insert padding if necessary.

Example

This example shows how `ARM` and `THUMB` can be used to switch state and assemble both ARM and Thumb instructions in a single area.

```
        AREA ToThumb, CODE, READONLY    ; Name this block of code
        ENTRY                           ; Mark first instruction to execute
        ARM                             ; Subsequent instructions are ARM
start
        ADR     r0, into_thumb + 1      ; Processor starts in ARM state
        BX      r0                      ; Inline switch to Thumb state
        THUMB                           ; Subsequent instructions are Thumb
into_thumb
        MOVS    r0, #10                 ; New-style Thumb instructions
```

6.8 Miscellaneous directives

This section describes the following directives:

- *ALIAS* on page 6-58
- *ALIGN* on page 6-59
- *AREA* on page 6-61
- *ATTR* on page 6-64
- *END* on page 6-65
- *ENTRY* on page 6-65
- *EQU* on page 6-66
- *EXPORT or GLOBAL* on page 6-67
- *EXPORTAS* on page 6-69
- *GET or INCLUDE* on page 6-70
- *IMPORT and EXTERN* on page 6-71
- *INCBIN* on page 6-73
- *KEEP* on page 6-74
- *NOFP* on page 6-75
- *REQUIRE* on page 6-75
- *REQUIRE8 and PRESERVE8* on page 6-76
- *ROUT* on page 6-77.

6.8.1 ALIAS

The `ALIAS` directive creates an alias for a symbol.

Syntax

`ALIAS` *`name`*`,` *`aliasname`*

where:

`name`	is the name of the symbol to create an alias for
`aliasname`	is the name of the alias to be created.

Usage

The symbol *`name`* must already be defined in the source file before creating an alias for it. Properties of *`name`* set by the `EXPORT` directive will not be inherited by *`aliasname`*, so you must use `EXPORT` on *`aliasname`* if you want to make the alias available outside the current source file. Apart from the properties set by the `EXPORT` directive, *`name`* and *`aliasname`* are identical.

Example

```
baz
bar PROC
    BX lr
    ENDP
    ALIAS  bar,foo    ; foo is an alias for bar
    EXPORT bar
    EXPORT foo        ; foo and bar have identical properties
                      ; because foo was created using ALIAS
    EXPORT baz        ; baz and bar are not identical
                      ; because the size field of baz is not set
```

Incorrect example

```
    EXPORT bar
    IMPORT car
    ALIAS  bar,foo ; ERROR - bar is not defined yet
    ALIAS  car,boo ; ERROR - car is external
bar PROC
    BX lr
    ENDP
```

See also

Reference

- *Data definition directives* on page 6-15
- *EXPORT or GLOBAL* on page 6-67.

6.8.2 ALIGN

The `ALIGN` directive aligns the current location to a specified boundary by padding with zeros or `NOP` instructions.

Syntax

`ALIGN {`*`expr`*`{,`*`offset`*`{,`*`pad`*`{,`*`padsize`*`}}}}`

where:

expr	is a numeric expression evaluating to any power of 2 from 2^0 to 2^{31}
offset	can be any numeric expression
pad	can be any numeric expression
padsize	can be 1, 2 or 4.

Operation

The current location is aligned to the next lowest address of the form:

`offset + n * expr`

n is any integer which the assembler selects to minimise padding.

If *expr* is not specified, `ALIGN` sets the current location to the next word (four byte) boundary. The unused space between the previous and the new current location are filled with:

- copies of *pad*, if *pad* is specified
- `NOP` instructions, if all the following conditions are satisfied:
  - *pad* is not specified
  - the `ALIGN` directive follows ARM or Thumb instructions
  - the current section has the `CODEALIGN` attribute set on the `AREA` directive
- zeros otherwise.

pad is treated as a byte, halfword, or word, according to the value of *padsize*. If *padsize* is not specified, *pad* defaults to bytes in data sections, halfwords in Thumb code, or words in ARM code.

Usage

Use `ALIGN` to ensure that your data and code is aligned to appropriate boundaries. This is typically required in the following circumstances:

- The `ADR` Thumb pseudo-instruction can only load addresses that are word aligned, but a label within Thumb code might not be word aligned. Use `ALIGN 4` to ensure four-byte alignment of an address within Thumb code.
- Use `ALIGN` to take advantage of caches on some ARM processors. For example, the ARM940T has a cache with 16-byte lines. Use `ALIGN 16` to align function entries on 16-byte boundaries and maximize the efficiency of the cache.
- `LDRD` and `STRD` doubleword data transfers must be eight-byte aligned. Use `ALIGN 8` before memory allocation directives such as `DCQ` if the data is to be accessed using `LDRD` or `STRD`.
- A label on a line by itself can be arbitrarily aligned. Following ARM code is word-aligned (Thumb code is halfword aligned). The label therefore does not address the code correctly. Use `ALIGN 4` (or `ALIGN 2` for Thumb) before the label.

Alignment is relative to the start of the ELF section where the routine is located. The section must be aligned to the same, or coarser, boundaries. The ALIGN attribute on the AREA directive is specified differently.

Examples

```
        AREA    cacheable, CODE, ALIGN=3
rout1   ; code         ; aligned on 8-byte boundary
        ; code
        MOV     pc,lr  ; aligned only on 4-byte boundary
        ALIGN   8      ; now aligned on 8-byte boundary
rout2   ; code
```

In the following example, the ALIGN directive tells the assembler that the next instruction is word aligned and offset by 3 bytes. The 3 byte offset is counted from the previous word aligned address, resulting in the second DCB placed in the last byte of the same word and 2 bytes of padding are to be added.

```
        AREA    OffsetExample, CODE
        DCB     1      ; This example places the two bytes in the first
        ALIGN   4,3    ; and fourth bytes of the same word.
        DCB     1      ; The second DCB is offset by 3 bytes from the first DCB
```

In the following example, the ALIGN directive tells the assembler that the next instruction is word aligned and offset by 2 bytes. Here, the 2 byte offset is counted from the next word aligned address, so the value *n* is set to 1 (*n*=0 clashes with the third DCB). This time three bytes of padding are to be added.

```
        AREA    OffsetExample1, CODE
        DCB     1      ; In this example, n cannot be 0 because it clashes with
        DCB     1      ; the 3rd DCB. The assembler sets n to 1.
        DCB     1
        ALIGN   4,2    ; The next instruction is word aligned and offset by 2.
        DCB     2
```

In the following example, the DCB directive makes the PC misaligned. The ALIGN directive ensures that the label subroutine1 and the following instruction are word aligned.

```
        AREA    Example, CODE, READONLY
start   LDR     r6,=label1
        ; code
        MOV     pc,lr
label1  DCB     1      ; PC now misaligned
        ALIGN          ; ensures that subroutine1 addresses
subroutine1            ; the following instruction.
        MOV r5,#0x5
```

See also

Reference

- *Data definition directives* on page 6-15
- *AREA* on page 6-61
- *Examples*.

6.8.3 AREA

The `AREA` directive instructs the assembler to assemble a new code or data section. Sections are independent, named, indivisible chunks of code or data that are manipulated by the linker.

Syntax

`AREA sectionname{,attr}{,attr}...`

where:

sectionname
: is the name to give to the section.

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

Certain names are conventional. For example, `|.text|` is used for code sections produced by the C compiler, or for code sections otherwise associated with the C library.

attr
: are one or more comma-delimited section attributes. Valid attributes are:

`ALIGN=`*expression*

By default, ELF sections are aligned on a four-byte boundary. *expression* can have any integer value from 0 to 31. The section is aligned on a $2^{expression}$-byte boundary. For example, if *expression* is 10, the section is aligned on a 1KB boundary.

This is not the same as the way that the `ALIGN` *directive is specified.*

——— **Note** ———

Do not use `ALIGN=0` or `ALIGN=1` for ARM code sections.

Do not use `ALIGN=0` for Thumb code sections.

———————————

`ASSOC=`*section*

section specifies an associated ELF section. *sectionname* must be included in any link that includes *section*

`CODE` Contains machine instructions. `READONLY` is the default.

`CODEALIGN`

Causes the assembler to insert `NOP` instructions when the `ALIGN` directive is used after ARM or Thumb instructions within the section, unless the `ALIGN` directive specifies a different padding.

`COMDEF` Is a common section definition. This ELF section can contain code or data. It must be identical to any other section of the same name in other source files.

Identical ELF sections with the same name are overlaid in the same section of memory by the linker. If any are different, the linker generates a warning and does not overlay the sections.

`COMGROUP=`*symbol_name*

Is the signature that makes the `AREA` part of the named ELF section group. See the `GROUP=`*symbol_name* for more information. The `COMGROUP` attribute marks the ELF section group with the `GRP_COMDAT` flag.

`COMMON` Is a common data section. You must not define any code or data in it. It is initialized to zeros by the linker. All common sections with the same name are overlaid in the same section of memory by the linker. They do not all have to be the same size. The linker allocates as much space as is required by the largest common section of each name.

`DATA` Contains data, not instructions. `READWRITE` is the default.

`FINI_ARRAY`

Sets the ELF type of the current area to `SHT_FINI_ARRAY`.

`GROUP=`*`symbol_name`*

Is the signature that makes the `AREA` part of the named ELF section group. It must be defined by the source file, or a file included by the source file. All `AREAS` with the same *`symbol_name`* signature are part of the same group. Sections within a group are kept or discarded together.

`INIT_ARRAY`

Sets the ELF type of the current area to `SHT_INIT_ARRAY`.

`LINKORDER=`*`section`*

Specifies a relative location for the current section in the image. It ensures that the order of all the sections with the `LINKORDER` attribute, with respect to each other, is the same as the order of the corresponding named *`sections`* in the image.

`MERGE=`*`n`* Indicates that the linker can merge the current section with other sections with the `MERGE=`*`n`* attribute. *n* is the size of the elements in the section, for example *n* is 1 for characters. You must not assume that the section will be merged because the attribute does not force the linker to merge the sections.

`NOALLOC` Indicates that no memory on the target system is allocated to this area.

`NOINIT` Indicates that the data section is uninitialized, or initialized to zero. It contains only space reservation directives `SPACE` or `DCB`, `DCD`, `DCDU`, `DCQ`, `DCQU`, `DCW`, or `DCWU` with initialized values of zero. You can decide at link time whether an area is uninitialized or zero initialized.

`PREINIT_ARRAY`

Sets the ELF type of the current area to `SHT_PREINIT_ARRAY`.

`READONLY` Indicates that this section must not be written to. This is the default for Code areas.

`READWRITE` Indicates that this section can be read from and written to. This is the default for Data areas.

`SECFLAGS=`*`n`*

Adds one or more ELF flags, denoted by *n*, to the current section.

`SECTYPE=`*`n`*

Sets the ELF type of the current section to *n*.

`STRINGS` Adds the `SHF_STRINGS` flag to the current section. To use the `STRINGS` attribute, you must also use the `MERGE=1` attribute. The contents of the section must be strings that are nul-terminated using the `DCB` directive.

Usage

Use the AREA directive to subdivide your source file into ELF sections. You can use the same name in more than one AREA directive. All areas with the same name are placed in the same ELF section. Only the attributes of the first AREA directive of a particular name are applied.

You should normally use separate ELF sections for code and data. However, you can put data in code sections. Large programs can usually be conveniently divided into several code sections. Large independent data sets are also usually best placed in separate sections.

The scope of local labels is defined by AREA directives, optionally subdivided by ROUT directives.

There must be at least one AREA directive for an assembly.

Note

The assembler emits R_ARM_TARGET1 relocations for the DCD and DCDU directives if the directive uses PC-relative expressions and is in any of the PREINIT_ARRAY, FINI_ARRAY, or INIT_ARRAY ELF sections. You can override the relocation using the RELOC directive after each DCD or DCDU directive. If this relocation is used, read-write sections might become read-only sections at link time if the platform ABI permits this.

Example

The following example defines a read-only code section named Example.

```
    AREA    Example,CODE,READONLY   ; An example code section.
            ; code
```

See also

Concepts

Using the Assembler:

- *ELF sections and the AREA directive* on page 4-5.

Concepts

Using the Linker:

- Chapter 4 *Image structure and generation*.

Reference

- *ALIGN* on page 6-59
- *RELOC* on page 6-8
- *DCD and DCDU* on page 6-21.

6.8.4 ATTR

The `ATTR` set directives set values for the ABI build attributes.

The `ATTR` scope directives specify the scope for which the set value applies to.

Syntax

`ATTR FILESCOPE`

`ATTR SCOPE` *`name`*

`ATTR` *`settype tagid`*`,` *`value`*

where:

`name` is a section name or symbol name.

`settype` can be any of:

- `SETVALUE`
- `SETSTRING`
- `SETCOMPATIBLEWITHVALUE`
- `SETCOMPATIBLEWITHSTRING`

`tagid` is an attribute tag name (or its numerical value) defined in the ABI for the ARM Architecture.

`value` depends on *`settype`*:

- is a 32-bit integer value when *`settype`* is `SETVALUE` or `SETCOMPATIBLEWITHVALUE`
- is a nul-terminated string when *`settype`* is `SETSTRING` or `SETCOMPATIBLEWITHSTRING`

Usage

The `ATTR` set directives following the `ATTR FILESCOPE` directive apply to the entire object file. The `ATTR` set directives following the `ATTR SCOPE` *`name`* directive apply only to the named section or symbol.

For tags that expect an integer, you must use `SETVALUE` or `SETCOMPATIBLEWITHVALUE`. For tags that expect a string, you must use `SETSTRING` or `SETCOMPATIBLEWITHSTRING`.

Use `SETCOMPATIBLEWITHVALUE` and `SETCOMPATIBLEWITHSTRING` to set tag values which the object file is also compatible with.

Examples

```
ATTR  SETSTRING Tag_CPU_raw_name, "Cortex-A8"
ATTR  SETVALUE  Tag_VFP_arch, 3   ; VFPv3 instructions were permitted.
ATTR  SETVALUE  10, 3             ; 10 is the numerical value of
                                  ; Tag_VFP_arch.
```

See also

Reference

- *Addenda to, and Errata in, the ABI for the ARM Architecture*, `http://infocenter.arm.com/help/topic/com.arm.doc.ihi0045-/index.html`.

6.8.5 END

The END directive informs the assembler that it has reached the end of a source file.

Syntax

```
END
```

Usage

Every assembly language source file must end with END on a line by itself.

If the source file has been included in a parent file by a GET directive, the assembler returns to the parent file and continues assembly at the first line following the GET directive.

If END is reached in the top-level source file during the first pass without any errors, the second pass begins.

If END is reached in the top-level source file during the second pass, the assembler finishes the assembly and writes the appropriate output.

See also

Reference

- *GET or INCLUDE* on page 6-70.

6.8.6 ENTRY

The ENTRY directive declares an entry point to a program.

Syntax

```
ENTRY
```

Usage

You must specify at least one ENTRY point for a program. If no ENTRY exists, a warning is generated at link time.

You must not use more than one ENTRY directive in a single source file. Not every source file has to have an ENTRY directive. If more than one ENTRY exists in a single source file, an error message is generated at assembly time.

Example

```
        AREA    ARMex, CODE, READONLY
        ENTRY                 ; Entry point for the application
```

6.8.7 EQU

The `EQU` directive gives a symbolic name to a numeric constant, a register-relative value or a PC-relative value. `*` is a synonym for `EQU`.

Syntax

`name` `EQU` *`expr`*`{,` *`type`*`}`

where:

`name`	is the symbolic name to assign to the value.
`expr`	is a register-relative address, a PC-relative address, an absolute address, or a 32-bit integer constant.
`type`	is optional. *`type`* can be any one of:

- `ARM`
- `THUMB`
- `CODE32`
- `CODE16`
- `DATA`

You can use *`type`* only if *`expr`* is an absolute address. If *`name`* is exported, the *`name`* entry in the symbol table in the object file will be marked as `ARM`, `THUMB`, `CODE32`, `CODE16`, or `DATA`, according to *`type`*. This can be used by the linker.

Usage

Use `EQU` to define constants. This is similar to the use of **`#define`** to define a constant in C.

Examples

```
abc EQU 2             ; assigns the value 2 to the symbol abc.
xyz EQU label+8       ; assigns the address (label+8) to the
                      ; symbol xyz.
fiq EQU 0x1C, CODE32  ; assigns the absolute address 0x1C to
                      ; the symbol fiq, and marks it as code
```

See also

Reference

- *KEEP* on page 6-74
- *EXPORT or GLOBAL* on page 6-67.

6.8.8 EXPORT or GLOBAL

The `EXPORT` directive declares a symbol that can be used by the linker to resolve symbol references in separate object and library files. `GLOBAL` is a synonym for `EXPORT`.

Syntax

```
EXPORT {[WEAK]}

EXPORT symbol {[SIZE=n]}

EXPORT symbol {[type{,set}]}

EXPORT symbol [attr{,type{,set}}{,SIZE=n}]

EXPORT symbol [WEAK{,attr}{,type{,set}}{,SIZE=n}]
```

where:

symbol is the symbol name to export. The symbol name is case-sensitive. If *symbol* is omitted, all symbols are exported.

`WEAK` *symbol* is only imported into other sources if no other source exports an alternative *symbol*. If `[WEAK]` is used without *symbol*, all exported symbols are weak.

attr can be any one of:

`DYNAMIC`	sets the ELF symbol visibility to `STV_DEFAULT`.
`PROTECTED`	sets the ELF symbol visibility to `STV_PROTECTED`.
`HIDDEN`	sets the ELF symbol visibility to `STV_HIDDEN`.
`INTERNAL`	sets the ELF symbol visibility to `STV_INTERNAL`.

type specifies the symbol type:

`DATA`	*symbol* is treated as data when the source is assembled and linked.
`CODE`	*symbol* is treated as code when the source is assembled and linked.
`ELFTYPE=`*n*	*symbol* is treated as a particular ELF symbol, as specified by the value of *n*, where *n* can be any number from `0` to `15`.

If unspecified, the assembler determines the most appropriate *type*. Usually the assembler determines the correct type so there is no need to specify the *type*.

set specifies the instruction set:

`ARM`	*symbol* is treated as an ARM symbol.
`THUMB`	*symbol* is treated as a Thumb symbol.

If unspecified, the assembler determines the most appropriate set.

n specifies the size and can be any 32-bit value. If the `SIZE` attribute is not specified, the assembler calculates the size:

- For `PROC` and `FUNCTION` symbols, the size is set to the size of the code until its `ENDP` or `ENDFUNC`.
- For other symbols, the size is the size of instruction or data on the same source line. If there is no instruction or data, the size is zero.

Usage

Use `EXPORT` to give code in other files access to symbols in the current file.

Use the [WEAK] attribute to inform the linker that a different instance of *symbol* takes precedence over this one, if a different one is available from another source. You can use the [WEAK] attribute with any of the symbol visibility attributes.

Example

```
        AREA    Example,CODE,READONLY
        EXPORT  DoAdd           ; Export the function name
                                ; to be used by external
                                ; modules.
DoAdd   ADD     r0,r0,r1
```

Symbol visibility can be overridden for duplicate exports. In the following example, the last EXPORT takes precedence for both binding and visibility:

```
        EXPORT  SymA[WEAK]      ; Export as weak-hidden
        EXPORT  SymA[DYNAMIC]   ; SymA becomes non-weak dynamic.
```

The following examples show the use of the SIZE attribute:

```
    EXPORT symA [SIZE=4]
    EXPORT symA [DATA, SIZE=4]
```

See also

Reference

- *IMPORT and EXTERN* on page 6-71.
- *ELF for the ARM Architecture ABI*, http://infocenter/help/topic/com.arm.doc.ihi0044-/index.html.

6.8.9 EXPORTAS

The `EXPORTAS` directive enables you to export a symbol to the object file, corresponding to a different symbol in the source file.

Syntax

`EXPORTAS` *`symbol1`*`,` *`symbol2`*

where:

`symbol1`	is the symbol name in the source file. *`symbol1`* must have been defined already. It can be any symbol, including an area name, a label, or a constant.
`symbol2`	is the symbol name you want to appear in the object file.

The symbol names are case-sensitive.

Usage

Use `EXPORTAS` to change a symbol in the object file without having to change every instance in the source file.

Examples

```
    AREA data1, DATA       ; starts a new area data1
    AREA data2, DATA       ; starts a new area data2
    EXPORTAS data2, data1  ; the section symbol referred to as data2 will
                           ; appear in the object file string table as data1.
one EQU  2
    EXPORTAS one, two
    EXPORT one             ; the symbol 'two' will appear in the object
                           ; file's symbol table with the value 2.
```

See also

Reference

- *EXPORT or GLOBAL* on page 6-67.

6.8.10 GET or INCLUDE

The `GET` directive includes a file within the file being assembled. The included file is assembled at the location of the `GET` directive. `INCLUDE` is a synonym for `GET`.

Syntax

`GET` *`filename`*

where:

`filename`	is the name of the file to be included in the assembly. The assembler accepts pathnames in either UNIX or MS-DOS format.

Usage

`GET` is useful for including macro definitions, `EQU`s, and storage maps in an assembly. When assembly of the included file is complete, assembly continues at the line following the `GET` directive.

By default the assembler searches the current place for included files. The current place is the directory where the calling file is located. Use the `-i` assembler command line option to add directories to the search path. File names and directory names containing spaces must not be enclosed in double quotes (" ").

The included file can contain additional `GET` directives to include other files.

If the included file is in a different directory from the current place, this becomes the current place until the end of the included file. The previous current place is then restored.

`GET` cannot be used to include object files.

Examples

```
    AREA    Example, CODE, READONLY
    GET     file1.s              ; includes file1 if it exists
                                 ; in the current place.
    GET     c:\project\file2.s   ; includes file2
    GET     c:\Program files\file3.s  ; space is permitted
```

See also

Reference

- *INCBIN* on page 6-73
- *Nesting directives* on page 6-29.

6.8.11 IMPORT and EXTERN

These directives provide the assembler with a name that is not defined in the current assembly.

Syntax

`directive symbol {[SIZE=n]}`

`directive symbol {[type]}`

`directive symbol [attr{,type}{,SIZE=n}]`

`directive symbol [WEAK{,attr}{,type}{,SIZE=n}]`

where:

`directive` can be either:

`IMPORT`	imports the symbol unconditionally.
`EXTERN`	imports the symbol only if it is referred to in the current assembly.

`symbol` is a symbol name defined in a separately assembled source file, object file, or library. The symbol name is case-sensitive.

`WEAK` prevents the linker generating an error message if the symbol is not defined elsewhere. It also prevents the linker searching libraries that are not already included.

`attr` can be any one of:

`DYNAMIC`	sets the ELF symbol visibility to `STV_DEFAULT`.
`PROTECTED`	sets the ELF symbol visibility to `STV_PROTECTED`.
`HIDDEN`	sets the ELF symbol visibility to `STV_HIDDEN`.
`INTERNAL`	sets the ELF symbol visibility to `STV_INTERNAL`.

`type` specifies the symbol type:

`DATA`	*`symbol`* is treated as data when the source is assembled and linked.
`CODE`	*`symbol`* is treated as code when the source is assembled and linked.
`ELFTYPE=n`	*`symbol`* is treated as a particular ELF symbol, as specified by the value of *n*, where *n* can be any number from `0` to `15`.

If unspecified, the linker determines the most appropriate type.

`n` specifies the size and can be any 32-bit value. If the `SIZE` attribute is not specified, the assembler calculates the size:

- For `PROC` and `FUNCTION` symbols, the size is set to the size of the code until its `ENDP` or `ENDFUNC`.
- For other symbols, the size is the size of instruction or data on the same source line. If there is no instruction or data, the size is zero.

Usage

The name is resolved at link time to a symbol defined in a separate object file. The symbol is treated as a program address. If `[WEAK]` is not specified, the linker generates an error if no corresponding symbol is found at link time.

If [WEAK] is specified and no corresponding symbol is found at link time:

- If the reference is the destination of a B or BL instruction, the value of the symbol is taken as the address of the following instruction. This makes the B or BL instruction effectively a NOP.
- Otherwise, the value of the symbol is taken as zero.

Example

The example tests to see if the C++ library has been linked, and branches conditionally on the result.

```
AREA    Example, CODE, READONLY
EXTERN  __CPP_INITIALIZE[WEAK]  ; If C++ library linked, gets the address of
                                ; __CPP_INITIALIZE function.
LDR     r0,=__CPP_INITIALIZE    ; If not linked, address is zeroed.
CMP     r0,#0                   ; Test if zero.
BEQ     nocplusplus             ; Branch on the result.
```

The following examples show the use of the SIZE attribute:

```
EXTERN symA [SIZE=4]
EXTERN symA [DATA, SIZE=4]
```

See also

Reference

- *ELF for the ARM Architecture*, http://infocenter.arm.com/help/topic/com.arm.doc.ihi0044-/index.html.
- *EXPORT or GLOBAL* on page 6-67.

6.8.12 INCBIN

The `INCBIN` directive includes a file within the file being assembled. The file is included as it is, without being assembled.

Syntax

`INCBIN` *`filename`*

where:

`filename`	is the name of the file to be included in the assembly. The assembler accepts pathnames in either UNIX or MS-DOS format.

Usage

You can use `INCBIN` to include executable files, literals, or any arbitrary data. The contents of the file are added to the current ELF section, byte for byte, without being interpreted in any way. Assembly continues at the line following the `INCBIN` directive.

By default, the assembler searches the current place for included files. The current place is the directory where the calling file is located. Use the `-i` assembler command line option to add directories to the search path. File names and directory names containing spaces must not be enclosed in double quotes (" ").

Example

```
        AREA    Example, CODE, READONLY
        INCBIN  file1.dat               ; includes file1 if it
                                        ; exists in the
                                        ; current place.
        INCBIN  c:\project\file2.txt    ; includes file2
```

6.8.13 KEEP

The `KEEP` directive instructs the assembler to retain local symbols in the symbol table in the object file.

Syntax

`KEEP {`*`symbol`*`}`

where:

symbol	is the name of the local symbol to keep. If *symbol* is not specified, all local symbols are kept except register-relative symbols.

Usage

By default, the only symbols that the assembler describes in its output object file are:

- exported symbols
- symbols that are relocated against.

Use `KEEP` to preserve local symbols that can be used to help debugging. Kept symbols appear in the ARM debuggers and in linker map files.

`KEEP` cannot preserve register-relative symbols.

Example

```
label   ADC     r2,r3,r4
        KEEP    label       ; makes label available to debuggers
        ADD     r2,r2,r5
```

See also

Reference

- *MAP* on page 6-17.

6.8.14 NOFP

The `NOFP` directive ensures that there are no floating-point instructions in an assembly language source file.

Syntax

```
NOFP
```

Usage

Use `NOFP` to ensure that no floating-point instructions are used in situations where there is no support for floating-point instructions either in software or in target hardware.

If a floating-point instruction occurs after the `NOFP` directive, an `Unknown opcode` error is generated and the assembly fails.

If a `NOFP` directive occurs after a floating-point instruction, the assembler generates the error:

```
Too late to ban floating point instructions
```

and the assembly fails.

6.8.15 REQUIRE

The `REQUIRE` directive specifies a dependency between sections.

Syntax

`REQUIRE` *`label`*

where:

`label`	is the name of the required label.

Usage

Use `REQUIRE` to ensure that a related section is included, even if it is not directly called. If the section containing the `REQUIRE` directive is included in a link, the linker also includes the section containing the definition of the specified label.

6.8.16 REQUIRE8 and PRESERVE8

The `REQUIRE8` directive specifies that the current file requires eight-byte alignment of the stack. It sets the REQ8 build attribute to inform the linker.

The `PRESERVE8` directive specifies that the current file preserves eight-byte alignment of the stack. It sets the PRES8 build attribute to inform the linker.

The linker checks that any code that requires eight-byte alignment of the stack is only called, directly or indirectly, by code that preserves eight-byte alignment of the stack.

Syntax

`REQUIRE8 {`*`bool`*`}`

`PRESERVE8 {`*`bool`*`}`

where:

`bool`	is an optional Boolean constant, either `{TRUE}` or `{FALSE}`.

Usage

Where required, if your code preserves eight-byte alignment of the stack, use `PRESERVE8` to set the PRES8 build attribute on your file. If your code does not preserve eight-byte alignment of the stack, use `PRESERVE8 {FALSE}` to ensure that the PRES8 build attribute is not set. If there are multiple `REQUIRE8` or `PRESERVE8` directives in a file, the assembler uses the value of the last directive.

——— **Note** ———

If you omit both `PRESERVE8` and `PRESERVE8 {FALSE}`, the assembler decides whether to set the PRES8 build attribute or not, by examining instructions that modify the SP. ARM recommends that you specify `PRESERVE8` explicitly.

You can enable a warning with:

```
armasm --diag_warning 1546
```

This gives you warnings like:

```
"test.s", line 37: Warning: A1546W: Stack pointer update potentially
                           breaks 8 byte stack alignment
      37 00000044         STMFD    sp!,{r2,r3,lr}
```

———————————

Examples

```
REQUIRE8
REQUIRE8     {TRUE}     ; equivalent to REQUIRE8
REQUIRE8     {FALSE}    ; equivalent to absence of REQUIRE8
PRESERVE8    {TRUE}     ; equivalent to PRESERVE8
PRESERVE8    {FALSE}    ; NOT exactly equivalent to absence of PRESERVE8
```

See also

Concepts

- *8 Byte Stack Alignment*, `http://infocenter.arm.com/help/topic/com.arm.doc.faqs/ka4127.html`.

Reference

- *Assembler command line options* on page 2-3.

6.8.17 ROUT

The `ROUT` directive marks the boundaries of the scope of local labels.

Syntax

`{name} ROUT`

where:

name	is the name to be assigned to the scope.

Usage

Use the `ROUT` directive to limit the scope of local labels. This makes it easier for you to avoid referring to a wrong label by accident. The scope of local labels is the whole area if there are no `ROUT` directives in it.

Use the *name* option to ensure that each reference is to the correct local label. If the name of a label or a reference to a label does not match the preceding `ROUT` directive, the assembler generates an error message and the assembly fails.

Example

```
            ; code
routineA    ROUT            ; ROUT is not necessarily a routine
            ; code
3routineA   ; code          ; this label is checked
            ; code
            BEQ     %4routineA   ; this reference is checked
            ; code
            BGE     %3      ; refers to 3 above, but not checked
            ; code
4routineA   ; code          ; this label is checked
            ; code
otherstuff  ROUT            ; start of next scope
```

See also

Concepts

Using the Assembler:

- *Local labels* on page 8-12.

Reference

- *AREA* on page 6-61.

Appendix A
Revisions for Assembler Reference

The following technical changes have been made to *Assembler Reference*.

Table A-1 Differences between issue F and issue G

Change	Topics affected
Updated the description of `--untyped_local_labels`.	*--untyped_local_labels* on page 2-25
Added the `ERET` instruction.	*ERET* on page 3-37
Mentioned that the `MVN` instruction exists in 16-bit Thumb.	*MOV and MVN* on page 3-62
Added a figure showing the operation of `VSHL` and updated the figures for `VSLI` and `VSRI`.	• *VSHL, VQSHL, VQSHLU, and VSHLL (by immediate)* on page 4-35 • *VSLI and VSRI* on page 4-40
Added links to the NEON and VFP data types topic from the associated NEON and VFP instructions.	Various NEON and VFP instructions
Mentioned that the `FUNCTION` directive can accept an empty reglist.	*FUNCTION or PROC* on page 6-47

Table A-2 Differences between issue E and issue F

Change	Topics affected
Added a note that the `--device` option is deprecated.	• *--device=list* on page 2-9 • *--device=name* on page 2-10
Modified the description of `--licretry`.	*--licretry* on page 2-17
Where appropriate: • changed Thumb-2 to 32-bit Thumb • changed Thumb-2EE to ThumbEE.	Various topics
Changed the minor version component of the integer reported by the `--version_number` option from one to two digits.	*--version_number* on page 2-25
Modified the description of `--vsn`.	*--vsn* on page 2-26
Mentioned a restriction on using `LSL` in an IT block with a zero value for *sh*.	*ASR, LSL, LSR, ROR, and RRX* on page 3-72
Clarified the range of addresses accessible to the `ADRL` pseudo-instruction in ARM state.	*ADRL pseudo-instruction* on page 3-156

Table A-3 Differences between issue D and issue E

Change	Topics affected
Added `SC300` and `SC000` to table of `--compatible` options.	*--compatible=name* on page 2-7

Table A-4 Differences between issue C and issue D

Change	Topics affected
In the summary table, changed instruction mnemonics from: • `VQRSHR` to `VQRSHR{U}N` • `VQSHR` to `VQSHR{U}N` • `VRSUBH` to `VRSUBHN` • `VSUBH` to `VSUBHN`. • `VRADDH` to `VRADDHN`.	Table 4-1 on page 4-2
Added `GBLA count` to the example.	*WHILE and WEND* on page 6-36
Changed FPv4_SP to FPv4-SP.	*--fpu=name* on page 2-14
Added ARM Glossary to other information.	Chapter 1 *Conventions and feedback*
Made changes to ALinknames for `MRS`, `MSR`, `SEV`, `SYS`, and `NOP` instructions.	• *MSR* on page 3-129 • *MRS* on page 3-130 • *SYS* on page 3-131 • *SEV, WFE, WFI, and YIELD* on page 3-145 • *NOP* on page 3-144
Added links to Memory access instructions in the `LDR` instruction pages.	*Memory access instructions* on page 3-9

Table A-5 Differences between issue B and issue C

Change	Topics affected
Changed the restrictions to say that `Rt` must be even-numbered only in `LDREXD` and `STREXD` instructions.	*LDREX and STREX* on page 3-40
Mentioned the additional cases where SP and PC are deprecated.	• *LDREX and STREX* on page 3-40 • *ADD, SUB, RSB, ADC, SBC, and RSC* on page 3-51 • *MOV and MVN* on page 3-62 • *B, BL, BX, BLX, and BXJ* on page 3-117 • *LDC, LDC2, STC, and STC2* on page 3-132
Mentioned that deprecation of SP and PC is only in ARMv6T2 and above.	Various instructions
Added example of inconsistent use of MAP and FIELD directives.	*FIELD* on page 6-18
Added note that the option is not required if you are using the ARM Compiler toolchain with DS-5.	• *--workdir=directory* on page 2-26 • *--project=filename* on page 2-21 • *--no_project* on page 2-20 • *--reinitialize_workdir* on page 2-23
Changed `--cpu PXA270` to `--device PXA270`.	*Introduction* on page 5-2

Table A-6 Differences between issue A and issue B

Change	Topics affected
Updated the description of `--cpu=name`.	*--cpu=name* on page 2-8
Added the option `--execstack`.	*--execstack* on page 2-13
Added the option `--no_execstack`.	*--no_execstack* on page 2-18
Added the option `--fpmode=none`.	*--fpmode=model* on page 2-13
Updated the description of `--show_cmdline`.	*--show_cmdline* on page 2-23
Updated the instruction summary table and footnotes with ARMv7E-M.	*Instruction summary* on page 3-2
Replaced "profile" with "architecture" when referring to ARMv6-M, ARMv7-M, ARMv7-R, and ARMv7-A in the instruction summary table and in the architecture sections of the instruction descriptions.	*Instruction summary* on page 3-2
Mentioned register-controlled shift in the description of Operand2.	*Operand2 as a register with optional shift* on page 3-47
Added register restrictions to `ADR` (PC-relative).	*ADR (PC-relative)* on page 3-24
Added register restrictions and deprecation information in `LDR` and `STR` (immediate offset).	*LDR and STR (immediate offset)* on page 3-11
Identified the ARM only instruction syntaxes in `LDR` and `STR` (register offset).	*LDR and STR (register offset)* on page 3-14
Added register restrictions and deprecation information, use of SP, and use of PC in `LDR` and `STR` (register offset).	*LDR and STR (register offset)* on page 3-14
Noted that PC-relative `STR` is available but deprecated.	*LDR (PC-relative)* on page 3-19

Table A-6 Differences between issue A and issue B (continued)

Change	Topics affected
Added information about deprecation and use of SP in `LDR` (PC-relative).	*LDR (PC-relative)* on page 3-19
In Restrictions of reglist in ARM instructions, added that reglist containing both PC and LR in ARM `LDM` is deprecated.	*LDM and STM* on page 3-30
Added Restrictions of reglist in ARM instructions.	*PUSH and POP* on page 3-33
Added register restriction for `Rn` and moved the statement "`Rm` must not be PC" to this section.	*PLD, PLDW, and PLI* on page 3-28
Added restrictions on reglist in `LDM` and `STM`.	*LDM and STM* on page 3-30
Added the statement "must not be PC" for each of the registers in the syntax.	*SWP and SWPB* on page 3-44
Linked to `SUBS pc, lr` from Use of PC in ARM instructions.	*ADD, SUB, RSB, ADC, SBC, and RSC* on page 3-51
Removed the caution against the use of the S suffix when using PC as `Rd` in User or System mode.	*ADD, SUB, RSB, ADC, SBC, and RSC* on page 3-51
Mentioned the deprecated instructions that use PC.	*ADD, SUB, RSB, ADC, SBC, and RSC* on page 3-51
Added more syntaxes that are only present in ARM code and described the additional items in the syntax.	*SUBS pc, lr* on page 3-55
Documented the valid forms of the `SUBS` instruction in ARM and Thumb, and added the caution to not use these instructions in User or System mode.	*SUBS pc, lr* on page 3-55
Linked to `SUBS pc, lr` from Use of PC in ARM instructions and from See also section.	*AND, ORR, EOR, BIC, and ORN* on page 3-57
Removed the caution against the use of the S suffix when using PC as `Rd` in User or System mode.	*AND, ORR, EOR, BIC, and ORN* on page 3-57
Added Register restrictions section to say `Rn` cannot be PC in instructions that write back to `Rn`.	*LDC, LDC2, STC, and STC2* on page 3-132
Mentioned that `Rt` cannot be PC.	*MCR, MCR2, MCRR, and MCRR2* on page 3-127
Mentioned that `Rm` cannot be PC.	*MSR* on page 3-139
Linked to `SUBS pc, lr` from Use of PC in ARM instructions.	*MOV and MVN* on page 3-62
Removed the caution against the use of the S suffix when using PC as `Rd` in User or System mode.	*MOV and MVN* on page 3-62
Mentioned the deprecated instructions that use PC.	*MOV and MVN* on page 3-62
Mentioned that SP is not permitted in Thumb `TST` and `TEQ` instructions, and is deprecated in ARM `TST` and `TEQ` instructions.	*TST and TEQ* on page 3-66
Added that `SEL` is available in ARMv7E-M.	*SEL* on page 3-68
Added that `Rn` must be different from `Rd` in `MUL` and `MLA` before ARMv6.	*MUL, MLA, and MLS* on page 3-77

Table A-6 Differences between issue A and issue B (continued)

Change	Topics affected
Added that `Rn` must be different from `RdLo` and `RdHi` before ARMv6.	*UMULL, UMLAL, SMULL, and SMLAL* on page 3-79
Added that the Thumb instructions are available in ARMv7E-M.	• *SMULxy and SMLAxy* on page 3-81 • *SMULWy and SMLAWy* on page 3-83 • *SMLALxy* on page 3-84 • *SMUAD{X} and SMUSD{X}* on page 3-86 • *SMMUL, SMMLA, and SMMLS* on page 3-88 • *SMLAD and SMLSD* on page 3-90 • *SMLALD and SMLSLD* on page 3-92 • *UMAAL* on page 3-94 • *QADD, QSUB, QDADD, and QDSUB* on page 3-98 • *Parallel add and subtract* on page 3-103 • *USAD8 and USADA8* on page 3-105 • *SSAT16 and USAT16* on page 3-107 • *SXT, SXTA, UXT, and UXTA* on page 3-112 • *PKHBT and PKHTB* on page 3-114
`DBG` is available in ARMv6K and above in ARM, and in ARMv6T2 and above in Thumb. Also mentioned that `DBG` executes as `NOP` in ARMv6K and ARMv6T2.	*DBG* on page 3-147
Added figures 4-4 and 4-5 for the operation of `VSLI` and `VSRI`.	*VSLI and VSRI* on page 4-40
Added tables showing the register state before and after operation of `VUZP` and `VZIP`.	*VUZP, VZIP* on page 4-32
Added that `n` can be a defined relocation name and add a related example in the examples section.	*RELOC* on page 6-8
Added note for macro workaround when using |.	*MACRO and MEND* on page 6-30
Clarified the message to say that error generation is during assembly rather than second pass of the assembly.	*ASSERT* on page 6-50
Added `ALIAS` directive, and included it in the summary table.	*ALIAS* on page 6-58
Clarified that `n` is any integer, and described the examples in the examples sections.	*ALIGN* on page 6-59
Clarified the description of `COMGROUP` and `GROUP`.	*AREA* on page 6-61
Added note about R_ARM_TARGET1.	*AREA* on page 6-61
Added link to 8 Byte Stack Alignment in See also section.	*REQUIRE8 and PRESERVE8* on page 6-76
Added `/hardfp` and `/softfp` values to the `--apcs` option and added link to the `--apcs` option in the Compiler Reference.	*--apcs=qualifier…qualifier* on page 2-5
Changed `Rn` to `Rm` in "`Rd`, `Rn`, `Rm` and `Ra` must not be PC".	*USAD8 and USADA8* on page 3-105