ELF Assembler

User's Guide for ARC®

0136-027

ELF Assembler User's Guide for ARC®

© 1995-2007 ARC® International. All rights reserved.

ARC International

North America
3590 N. First Street, Suite 200
San Jose, CA 95134 USA
Tel. +1-408-437-3400
Fax +1-408-437-3401

Europe
Verulam Point
Station Way
St Albans, AL1 5HE
UK
Tel.+44 (0) 20-8236-2800
Fax +44 (0) 20-8236-2801

www.ARC.com

ARC Confidential and Proprietary Information

Notice

This document, material and/or software contains confidential and proprietary information of ARC International and is protected by copyright, trade secret, and other state, federal, and international laws, and may be embodied in patents issued or pending. Its receipt or possession does not convey any rights to use, reproduce, disclose its contents, or to manufacture, or sell anything it may describe. Reverse engineering is prohibited, and reproduction, disclosure, or use without specific written authorization of ARC International is strictly forbidden. ARC and the ARC logotype are trademarks of ARC International.

The product described in this manual is licensed, not sold, and may be used only in accordance with the terms of a License Agreement applicable to it. Use without a License Agreement, in violation of the License Agreement, or without paying the license fee is unlawful.

Every effort is made to make this manual as accurate as possible. However, ARC International shall have no liability or responsibility to any person or entity with respect to any liability, loss, or damage caused or alleged to be caused directly or indirectly by this manual, including but not limited to any interruption of service, loss of business or anticipated profits, and all direct, indirect, and consequential damages resulting from the use of this manual. ARC International's entire warranty and liability in respect of use of the product are set forth in the License Agreement.

ARC International reserves the right to change the specifications and characteristics of the product described in this manual, from time to time, without notice to users. For current information on changes to the product, users should read the "readme" and/or "release notes" that are contained in the distribution media. Use of the product is subject to the warranty provisions contained in the License Agreement

Licensee acknowledges that ARC International is the owner of all Intellectual Property rights in such documents and will ensure that an appropriate notice to that effect appears on all documents used by Licensee incorporating all or portions of this Documentation.

The manual may only be disclosed by Licensee as set forth below.

- Manuals marked "ARC Confidential & Proprietary" may be provided to Licensee's subcontractors under NDA. The manual may not be provided to any other third parties, including manufacturers. Examples—source code software, programmer guide, documentation.
- Manuals marked "ARC Confidential" may be provided to subcontractors or manufacturers for use in Licensed Products. Examples—product presentations, masks, non-RTL or non-source format.
- Manuals marked "Publicly Available" may be incorporated into Licensee's documentation with appropriate ARC permission. Examples—presentations and documentation that do not embody confidential or proprietary information.

U.S. Government Restricted Rights Legend

Use, duplication or disclosure by the U.S. Government is subject to restrictions as set forth in FAR 52.227.19(c)(2) or subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 and/or in similar or successor clauses in the FAR, or the DOD or NASA FAR Supplement.

CONTRACTOR/MANUFACTURER IS ARC International I. P., Inc., 3590 N. First Street, Suite 200, San Jose, California 95134.

Trademark Acknowledgments

ARC-Based, ARC-OS Changer, ARCangel, ARCform, ARChitect, ARCompact, ARCtangent, BlueForm, CASSEIA, High C/C++, High C++, iCon186, IPShield, MetaDeveloper, the MQX Embedded logo, Precise Solution, Precise/BlazeNet, Precise/EDS, Precise/MFS, Precise/MQX, Precise/MQX Test Suites, Precise/MQXsim, Precise/RTCS, Precise/RTCSsim, RTCS, SeeCode, TotalCore, Turbo186, Turbo86, V8 µ-RISC, V8 microRISC, and VAutomation are trademarks of ARC International. ARC, the ARC logo, High C, MetaWare, MQX and MQX Embedded are registered under ARC International. All other trademarks are the property of their respective owners.

0136-027 August 2007

Contents

Chapter 1 — Before you begin

In This Chapter

Technical Support

We encourage customers to visit our Technical Support site for technical information before submitting support requests via telephone. The Technical Support site is richly populated with the following:

ARCSolve — Frequently asked questions

ARCSpeak — Glossary of terms

White papers — Technical documents

Datasheets — ARC product flyers

Known Issues — Known bugs and workarounds.

Training — Future ARC Training sessions

Support — Entry point to sending in a query

Downloads — Access to useful scripts and tools

You can access this site via our corporate website at `http://www.ARC.com`, or you can access the site directly: `http://support.ARC.com`.

Note that you must register before you can submit a support request via our Extranet site.

About this book

The *ELF Assembler User's Guide for ARC®* describes how to use the ELF assembler for ARC processors.

This manual does not contain information about assembler mnemonics. It is intended to be used along with your processor documentation.

This book contains the following topics:

- Using the assembler
- ELF assembly language
- Assembler macros
- Assembler directives

Where to go for more information

The release note or `readme` file describes any special files and provides information that was not available at the time the manuals were published.

The "Where to go for more information" section in the "Before you begin" chapter of the *MetaWare C/C++ Programmer's Guide* describes the following:

- Documents in the MetaWare C/C++ Development Toolkit
- C and C++ programming documents
- Processor-specific documents
- Specifications and ABI documents

For information about the ELF Executable and Linkable Format, refer to *TIS Portable Formats Specification Version 1.1. TIS Committee, 1993*.

Document conventions

install_dir refers to the location where you installed the toolkit.

Notes

Notes point out important information.

NOTE	Names of command-line options are case-sensitive.

Cautions

Cautions tell you about commands or procedures that could have unexpected or undesirable side effects or could be dangerous to your files or your hardware.

CAUTION	Comments in assembly code can cause the preprocessor to fail if they contain C preprocessing tokens such as `#if` or `#end`, C comment delimiters, or invalid C tokens.

Chapter 2 — Using the assembler

In This Chapter

Overview

The assembler takes assembly-language text files as input and generates relocatable object files conforming to the Executable and Linkable Format (ELF). The assembler accepts input files containing instruction mnemonics as described in the *Programmer's Reference* for your ARC processor.

The assembler parses and encodes instructions for ARC processors, using different command-line invocations for ARCtangent-A4 (and previous) processors and ARCtangent-A5 (and later) processors; see Invoking the assembler directly. Differences in assembler feature availability are indicated in the text of this manual.

You can invoke the assembler directly, or by using the **mcc** driver command on `.s` files. Some options listed in this manual are available only when using the driver, and are labeled as driver options.

Invoking the assembler directly

This is the command-line syntax for invoking the assembler:

ARCtangent-A4 and previous

asarc [*options*] [`@`*arg_file*] *source_file*.s

ARCtangent-A5 and later

asac [*options*] [`@`*arg_file*] *source_file*.s

Each command-line invocation (**asarc** or **asac**) defines a different value for built-in symbol `$isa`.

- *options* is an optional series of command-line options. (Assembler options are described in detail in Command-line options on page 11.) Note that for the assembler, you must specify the options before the source file.

  For **asac**, include options to specify your processor series and core version, or accept the defaults.

  For both invocations, include the option to specify your core version, or accept the default.
- *arg_file* is an ASCII file with more command-line options. Note that for the assembler, you must specify the options before the source file.
- *source_file*`.s` is the name of the assembly source file (or files) being assembled, with a default extension `.s`.

NOTE If you invoke the assembler without a source-file name, you get the command-line help screen.

Whitespace is required between elements in a command line, except between an option and its argument; for example, either **-o** *`hello.o`* or **-ohello.o** is acceptable.

Invoking the assembler using the driver

You can invoke the assembler by using the same driver you use to invoke the compiler and giving `.s` files as arguments. This is the driver syntax for invoking the assembler:

`mcc` [*options*] [`@`*arg_file*] *source_file*`.s`

- *options* is an optional series of command-line options. Include options to specify your processor series and core version, or accept the defaults.

  Assembler options that are not identified as driver options in Command-line options on page 11 must be preceded by option **-Hasopt=**. Note that for the assembler, you must specify the options before the source file.

- *arg_file* is an ASCII file with more command-line options. Note that for the assembler, you must specify the options before the source file.

- *source_file*`.s` is the name of the assembly source file (or files) being assembled, with a default extension `.s`.

The output file name

By default, the assembler generates an object file with the same file name as the assembly source file, but with a `.o` extension:

source_file.o

To specify a different output-file name, use command-line option **-o**.

You can assemble more than one source file with a single command. The assembler concatenates all the source files into one object file with the same file name as the last source file specified. For example, the following command generates one output file, `srcfile3.o`, which contains object code for all three source files:

ARCtangent-A4

asarc srcfile1.s srcfile2.s srcfile3.s

ARCtangent-A5 and later

asac srcfile1.s srcfile2.s srcfile3.s

Generating an output listing

The assembler generates an assembly-language output listing only if you specify command-line option **-l**. The assembler directs the listing to standard output, unless you redirect the listing to an output file. See Command-line option reference on page 12 for more information about option **-l**.

Preprocessing

The ELF assembler includes a macro preprocessor, but its preprocessing directives differ from C-style preprocessing directives. If your source files contain C-style preprocessing directives, you must invoke the MetaWare C/C++ driver with **mcc** driver command-line option **-Hasmcpp** to preprocess these files.

See the *MetaWare C/C++ Programmer's Guide* for information about driver option **-Hasmcpp**.

CAUTION	Comments in assembly code can cause the preprocessor to fail if they contain C preprocessing tokens such as **#if** or **#end**, C comment delimiters, or invalid C tokens.

Command-line options

Assembler command-line options specify how to assemble source files and what output to generate.

You must place assembler options on the command line before the name of the source files you want them to affect.

> **NOTE** Names of command-line options are case-sensitive.

To see a listing of all the assembler command-line options, use option **-h**.

Passing assembler options from the driver

You can use assembler command-line options with the **asarc** (or **asac**) command or with the **mcc** driver command. To pass assembler options to the assembler using the driver, use driver command-line option **-Hasopt** (see the *MetaWare C/C++ Programmer's Guide* for more information about option **-Hasopt**).

Passing assembler options from the IDE (Windows hosts)

If you are using an ARC International assembler hosted on Windows, you can pass assembler options directly from the IDE without using the driver or command line. For more information, see IDE user's guide or online help.

Command-line option reference

This section presents detailed listings of assembler command-line options. For an abbreviated listing, see Appendix A — Quick Options List.

-%reg — Specify that register names must be preceded by a percent sign

Specifies that register names must be preceded by a percent sign (%), the register-name identifier, causing the assembler to recognize any name that does not begin with % as belonging to a user-defined identifier, not a register. By default, the assembler recognizes the name of any register, with or without the percent sign, as a register name.

Synonym for **-percent_reg**.

See Using register names as identifiers on page 26 for more information.

-a4 — Specify the ARCtangent-A4 processor series

Option **-a4** specifies the ARCtangent-A4 processor series. Option **-a4** is the default when you invoke the assembler with the **asarc** command.

You can also specify a core version using a **-core*** option (**-core8** is the default).

-a5 — Specify the ARCtangent-A5 processor series

Option **-a5** specifies the ARCtangent-A5 processor series.

You can also specify a core version using a **-core*** option (**-core3** is the default).

-ac32 — Use 32-bit encodings (ARCtangent-A5 only)

When invoked using **asac** for ARCtangent-A5, the assembler uses the ARCtangent-A5 instruction-set architecture, a mixture of 16- and 32-bit instructions. Option **-ac32** causes the assembler to use 32-bit encodings for all instructions.

-arc600 | -a6 — Specify the ARC 600 processor series

Use option **-arc600** (synonym: **-a6**) when assembling code for the ARC 600 processor series. Option **-a6** is the default when you invoke the assembler with the **asac** command.

Core version **-core*** is specified by default with this option.

-arc700 | -a7 — Specify the ARC 700 processor series

Use option **-arc700** (synonym: **-a7**) when assembling code for the ARC 700 processor series. Core version **-core*** is specified by default with this option.

-be — Assemble using big-endian format

Causes the assembler to generate object code in big-endian format. The default is little-endian mode.

-c — Suppress display of the copyright message

Suppresses the display of the copyright message that normally appears when you invoke the assembler.

-core* — Specify core version

Specifies the target core version of the specified processor series (**-a***).

For example, **-a5 -core3** specifies core version 3 of the ARCtangent-A5 processor series.

-D*name*[=*n*] — Define an identifier and assign a value to it

Defines identifier *name* and, if you specify a value *n*, assigns *n* to *name*. If you do not specify *n*, the value of *name* defaults to `1` (one). Specifying option **-D*name*=*n*** is the same as putting the following directive in your assembly source file:

```
.define name, n
```

> **NOTE** You can pass this option using the driver without the need for option **-Hasopt**. For more information, see Passing assembler options from the driver on page 12.

-Eo — Send error messages to standard output

Causes the assembler to write error messages to `stdout` instead of to `stderr`.

-errors *n* — Set the assembler error limit

Sets the assembler error limit; that is, the maximum number of errors that the assembler allows before quitting. The default assembler error limit is 25.

Specifying a higher error limit is useful when, for example, you debug source with multiple complex macros — it is difficult to determine where an error occurs without the help of the listing, which will not be produced if the error limit is reached.

-extregs — Define extension core registers

Defines all extension core registers with the default names `%r32`, `%r33`, etc.

-f*flag*[*flag*...] — Set listing flags

Sets listing flags that control the contents and appearance of the assembly source listing.

Option **-f** has the same effect as assembler directive **.lflags**, except that it can be applied only to the module as a whole.

NOTE Option **-f** takes effect only if you also specify option **-l**.

Table 1 Assembler listing flags

Flag	Function	Default
`1`	Enable pass-one listing (useful if you want to see the results of macro expansion).	Off
`2`	Enable pass-two listing.	On
`c`	List instructions that were not assembled because of conditional assembly statements.	Off
`f`	Write source listing to a file with the same name as the source file, with a `.lst` extension.	Off
`g`	List local label symbols in the symbol and cross-reference tables.	Off
`h`	Suppress page headings in the listing file	Off
`i`	List include files in the source listing.	Off
`l`*n*	Set line width of the source listing to *n* spaces ($40 \le n \le 255$).	*n*=132
`m`	List macros and show expansions in source listing.	Off
`o`	List data-storage overflow.	Off
`p`*n*	Set page length of the source listing to *n* lines ($20 \le n \le 255$; setting *n*=0 (zero) means no pagination).	*n*=55
`s`	Show the symbol table in the source listing.	Off
`x`	Show the cross-reference table in the source listing.	Off

Setting and toggling listing flags

To turn On a flag, list it after option **-f**. To turn Off a flag, list it after option **-f** and put an *n* before it. To set the *l* and *p* flags, insert an integer value after them in the flag list.

For example, the command **-f***imns*`l80` causes the assembler to do the following:

- List include files in the source listing (flag *i*).
- Include macro expansions in the source listing (flag *m*).
- Not show the symbol table in the source listing (flag *ns*).
- Set source listing line-width to 80 characters (flag `l80`).

-g — Generate debugging information for assembly source files

Causes the assembler to annotate the assembled object file with line-number information. This information allows the debugger to display the actual assembly source file, complete with comments and declarations, instead of disassembled instructions (which can look quite different from the original instructions in the assembly source file).

Option **-g** provides no symbolic or type information.

NOTE You can pass this option using the driver without the need for option **-Hasopt**. For more information, see Passing assembler options from the driver on page 12.

-h — Display command-line option help screen

Displays a screen listing of the assembler command-line options, their arguments, and what they do.

-I*include_dir*[,*include_dir...*] — Specify directory to be searched for .include files

Directs the assembler to search directory *include_dir* for **.include** files with relative addresses. The assembler searches first the current working directory, then any directories specified with option **-I**.

> **NOTE** You can pass this option using the driver without the need for option **-Hasopt**. For more information, see Passing assembler options from the driver on page 12.

-l — Generate an assembly output listing

Directs the assembler to generate an assembly output listing.

The assembler writes this listing to standard output unless you redirect it to a list file.

For information about controlling the contents and appearance of the listing, see option **-f** and directive **.lflags**.

-L — Place private labels in the output symbol table

Causes the assembler to place private labels in the output-file symbol table.

A private label is one that begins with a period; for example, `.my_label`.

-le — Assemble using little-endian format

Causes the assembler to generate object code in little-endian format (the default).

-makeof=*filename* — Generate makefile-dependency file

Directs the assembler to generate a makefile-dependency file with the name *filename*.

See also option **-make_rel** and toggle `normalize_file_names`.

-make_rel — Exclude absolute pathnames from makefile-dependency info

Excludes absolute pathnames from the makefile-dependency information.

See also option **-makeof** and toggle `normalize_file_names`.

-mathu —Perform expression evaluations using unsigned math.

Instructs the assembler to perform expression evaluations using unsigned math.

-noesc — Disable backslash (C-style) escape character

Disables the backslash (C-style) escape character in literal strings. When you specify option **-noesc**, the assembler treats the backslash as a normal character.

> **NOTE** Option **-noesc** disables the backslash escape character over an entire assembly module. To enable or disable backslash escapes within assembly source, use directive **.option** with the *noesc* and *esc* arguments.

-o *object_file* — Override the default object-file name

Overrides the default assembler-generated object-file name. The default is the name of the assembly source file with a `.o` extension.

For example, the following command generates an object file `sort_1.obj` (rather than the default, `sort.o`):

ARCtangent-A4

asarc -o sort_1.obj sort.s

ARCtangent-A5 and later

asac -o sort_1.obj sort.s

-offwarn=*warning_no*[,*warning_no*...] — Turn off selected warnings

Disables selected warnings. The argument to this option is a comma-separated list of warning numbers.

The warning numbers are displayed by the assembler when the warning is issued. If a warning does not include a warning number, the warning cannot be disabled.

Example

```
-offwarn=121,128
```

-on | -off — Turn one or more toggles on or off

Option **-on** turns on one or more toggles, and option **-off** turns off one or more toggles. To use these options for more than one toggle, separate the toggles using commas as shown in the following examples:

```
-on=emit_cfa,emit_td
```

```
-off=emit_cfa,emit_td
```

The **-on** and **-off** options support the following toggles:

dwarf_comp_dir	Default: on. When this toggle is on, the assembler emits the full directory path of the source file in the DWARF debug information generated when option **-g** is specified. Turn this toggle off to prevent the compilation directory from being emitted.
emit_cfa	Default: on. When this toggle is off, the assembler does not emit any canonical-frame-address information, but still recognizes CFA directives.
emit_td	Default: on. Emit `$t` and `$d` symbols to delimit instruction mnemonics from data declaration directives. The disassembler uses these symbols to avoid disassembling raw data as instructions. The assembler does not emit the symbols if you turn the toggle off.
esc	Default: on. Allows C-style escapes in strings. May also be specified as the **-noesc** option.
expand_suffixes	Default: off. Processes instruction suffixes through the macro processor. This allows the creation of user-defined suffixes using **.define**.
mathu	Default: off. Directs the assembler to perform expression evaluation using unsigned math. Use of signed math is the default. May also be specified as the **-mathu** option.
normalize_file_names	Default: off. Normalizes file names in a dependency file. When a file name is *normalized*, redundant `../` relative directories are removed from the path and file name. For example, with `normalize_file_names` turned on, the path and file name `../proj_a/source/../../debug/wrx.c` becomes simply `../debug/wrx.c`. See also options **-makeof** and **-make_rel**.

percent_reg	Default: off. Toggle the requirement for registers to be prefixed with a '%' character. Using this toggle may be useful for inline assembly macros, for example, where the compiler emits code requiring the use of '%' with registers, but you might not wish to follow this convention in the inline assembly macros you write. Note however that if you turn off assembler toggle `percent_reg` you must not have identifiers in your program that match the names of any registers you are using with the toggle turned off. See also option **-%reg**.
require_ declarations	Default: off. Does not permit undefined or undeclared symbols (same as **-require_declarations** option).
show_errline	Default: off. When issuing an error message, include the line of assembly source that caused the error.

-percent_reg — Specify that register names must be preceded by a percent sign

Synonym for **-%reg**.

-Q {y|n} — Specify whether assembler version-number information appears in the object file

Option **-Q y** places assembler version-number information in the comment section of the generated object file. Option **-Q n** suppresses placement of this information.

Option **-Q n** is the default.

-require_declarations — Do not permit undefined or undeclared symbols

Causes the assembler to emit an error if symbols are referenced without being declared or defined. Exported symbols must be defined; imported symbols must be declared with either **.weak** or **.extern**.

-U*name* — Undefine an identifier

Undefines an identifier *name* previously defined with option **-D**. In the following example, `abc` is defined for `file1.s`, but not defined for `file2.s`:

ARCtangent-A4

```
asarc -Dabc file1.s -Uabc file2.s
```

ARCtangent-A5 and later

```
asac -Dabc file1.s -Uabc file2.s
```

-v — Print summary of assembler statistics

Causes the assembler to write to standard output a summary of statistics about the program being assembled.

-w — Suppress warning messages

Tells the assembler not to emit warning messages. You should use this option only after you have determined that the conditions warned about are acceptable.

-wn — Warn about instructions with no effect

Displays a warning message for instructions that do not produce a result and do not change the processor's status flags. Such instructions are equivalent to **nop**s.

-Xadds — Enable ADDS and SUBS instructions

mcc driver option

Option **-Xadds** enables support for the **ADDS** (saturating add) and **SUBS** (saturating subtract) instructions. This option also defines the preprocessor macro **__Xadds** to the compiler and the assembler.

Option **-Xadds** is a driver option; for information on how to use it see Invoking the assembler using the driver on page 10.

NOTE The file *install_dir*/bin/arcexlib.s contains the assembler definition of the Extensions Library and DSP extensions; it is used by the **mcc** driver when assembling code with the **-X** family of driver options.

When this option is used with ARCtangent-A4 and previous (**-a4**), the assembler produces information in the object file that causes the runtime startup initialization to verify the presence of the extension by checking the build-configuration register (BCR).

-Xbarrel_shifter — Enable barrel shifter instructions

mcc driver option

Option **-Xbarrel_shifter** enables support for barrel-shifter instructions. This option also defines preprocessor macro **__Xbarrel_shifter** to the compiler and the assembler.

Option **-Xbarrel_shifter** is a driver option; for information on how to use it see Invoking the assembler using the driver on page 10.

NOTE The file *install_dir*/bin/arcexlib.s contains the assembler definition of the Extensions Library and DSP extensions; it is used by the **mcc** driver when assembling code with the **-X** family of driver options.

When this option is used with ARCtangent-A4 and previous (**-a4**), the assembler produces information in the object file that causes the runtime startup initialization to verify the presence of the extension by checking the build-configuration register (BCR).

-Xdpfp — Enable double-precision floating-point extensions

mcc driver option

For ARC 600 and Later

Option **-Xdpfp** enables support for the double precision floating point extensions. This option also defines preprocessor macro **__Xdpfp** to the compiler and the assembler.

Option **-Xdpfp** is a driver option; for information on how to use it see Invoking the assembler using the driver on page 10.

NOTE The file *install_dir*/bin/arcexlib.s contains the assembler definition of the Extensions Library and DSP extensions; it is used by the **mcc** driver when assembling code with the **-X** family of driver options.

-Xdsp_packa — Use DSP Pack A Extension Instructions

mcc driver option for ARCtangent-A5 and later

Option **-Xdsp_packa** enables support for the DSP Pack A extension instructions and defines preprocessor macro **__Xdsp_packa**. Use this option when invoking the **mcc** driver on assembly files containing DSP Pack A extension instructions.

Option **-Xdsp_packa** is a driver option; for information on how to use it see Invoking the assembler using the driver on page 10.

> **NOTE** The file *install_dir*/bin/arcexlib.s contains the assembler definition of the Extensions Library and DSP extensions; it is used by the **mcc** driver when assembling code with the **-X** family of driver options.

-Xmin_max — Enable MIN and MAX instructions (ARCtangent-A4 only)

mcc driver option (use with -a4)

Option **-Xmin_max** enables support for the **MIN** and **MAX** instructions. This option also defines preprocessor macro **__Xmin_max** to the compiler and the assembler. (For ARCtangent-A5, **MIN** and **MAX** are base-case instructions and require no special option.)

Option **-Xmin_max** is a driver option; for information on how to use it see Invoking the assembler using the driver on page 10.

> **NOTE** The file *install_dir*/bin/arcexlib.s contains the assembler definition of the Extensions Library and DSP extensions; it is used by the **mcc** driver when assembling code with the **-X** family of driver options.

When you specify this option, the assembler produces information in the object file that causes the runtime startup initialization to verify the presence of the extension by checking the build-configuration register (BCR).

-Xmul32x16 — Multiply using 32x16 extension instructions

Option **-Xmul32x16** enables the 32x16 multiply extension instructions. Option **-Xmult32**, if also specified, takes precedence over **-Xmul32x16**.

> **NOTE** The file *install_dir*/bin/arcexlib.s contains the assembler definition of the Extensions Library and DSP extensions; it is used by the **mcc** driver when assembling code with the **-X** family of driver options.

-Xmul_mac — Enable MULMAC extension instructions

> **NOTE** The MULMAC extension instructions have been deprecated; use the XMAC extension instructions instead (options **-Xxmac_16**, **-Xxmac_d16**, and **-Xxmac_24**).

-Xmult32 — Enable scoreboarded multiplier instructions

mcc driver option

Option **-Xmult32** enables support for the 32-by-32 scoreboarded multiplier. This option also defines preprocessor macro **__Xmult32** to the compiler and the assembler.

Option **-Xmult32** is a driver option; for information on how to use it see Invoking the assembler using the driver on page 10.

> **NOTE** The file *install_dir*/bin/arcexlib.s contains the assembler definition of the Extensions Library and DSP extensions; it is used by the **mcc** driver when assembling code with the **-X** family of driver options.

When this option is used with processors prior to ARC 700, the assembler produces information in the object file that causes the runtime startup initialization to verify the presence of the extension by checking the build-configuration register (BCR).

-Xnorm — Enable NORM instruction

mcc driver option

Option **-Xnorm** enables support for the **NORM** instruction. This option also defines preprocessor macro **__Xnorm** to the compiler and the assembler.

Option **-Xnorm** is a driver option; for information on how to use it see Invoking the assembler using the driver on page 10.

> **NOTE** The file *install_dir*/bin/arcexlib.s contains the assembler definition of the Extensions Library and DSP extensions; it is used by the **mcc** driver when assembling code with the **-X** family of driver options.

When this option is used with processors prior to ARC 700, the assembler produces information in the object file that causes the runtime startup initialization to verify the presence of the extension by checking the build-configuration register (BCR).

-Xscratch_ram — Enable scratch RAM extension

mcc driver option

Option **-Xscratch_ram** enables support for the scratch-RAM extension. This option also defines preprocessor macro **__Xscratch_ram** to the compiler and the assembler.

Option **-Xscratch_ram** is a driver option; for information on how to use it see Invoking the assembler using the driver on page 10.

> **NOTE** The file *install_dir*/bin/arcexlib.s contains the assembler definition of the Extensions Library and DSP extensions; it is used by the **mcc** driver when assembling code with the **-X** family of driver options.

-Xspfp — Enable native floating point extensions

For ARC 600 and Later

Option **-Xspfp** enables the native single-precision floating point extensions.

-Xswap — Enable SWAP instruction

mcc driver option

Option **-Xswap** enables support for the **SWAP** instruction. This option defines preprocessor macro **__Xswap** to the compiler and the assembler.

Option **-Xswap** is a driver option; for information on how to use it see Invoking the assembler using the driver on page 10.

NOTE The file *install_dir*/bin/arcexlib.s contains the assembler definition of the Extensions Library and DSP extensions; it is used by the **mcc** driver when assembling code with the **-X** family of driver options.

When this option is used with processors prior to ARC 700, the assembler produces information in the object file that causes the runtime startup initialization to verify the presence of the extension by checking the build-configuration register (BCR).

-Xvbfdw — Enable support for dual Viterbi instruction

mcc driver option for ARCtangent-A5 and later

Option **-Xvbfdw** enables support for the dual-word Viterbi decoder butterfly instruction. This option affects only the assembler and defines preprocessor macro `__Xvbfdw`. This option accelerates Viterbi decoding algorithms by performing multiple add-compare-select operations in hardware.

Option **-Xvbfdw** is a driver option; for information on how to use it see Invoking the assembler using the driver on page 10.

NOTE The file *install_dir*/bin/arcexlib.s contains the assembler definition of the Extensions Library and DSP extensions; it is used by the **mcc** driver when assembling code with the **-X** family of driver options.

-Xxmac_16 — Enable 16-by-16 multiply accumulate extension

mcc driver option

Option **-Xxmac_16** enables support for the 16-by-16 multiply accumulate extension. This option defines preprocessor macros **__Xxmac_16** and **__Xxmac** to the compiler and the assembler.

Option **-Xxmac_16** is a driver option; for information on how to use it see Invoking the assembler using the driver on page 10.

NOTE The file *install_dir*/bin/arcexlib.s contains the assembler definition of the Extensions Library and DSP extensions; it is used by the **mcc** driver when assembling code with the **-X** family of driver options.

When you use this option with ARCtangent-A4 and previous (**-a4**), the assembler produces information in the object file that causes the runtime startup initialization to verify the presence of the extension by checking the build-configuration register (BCR).

-Xxmac_d16 — Enable dual 16-by-16 multiply accumulate extension

mcc driver option

Option **-Xxmac_d16** enables support for the dual 16-by-16 multiply accumulate extension. This option defines preprocessor macros **__Xxmac_d16** and **__Xxmac** to the compiler and the assembler.

Option **-Xxmac_d16** is a driver option; for information on how to use it see Invoking the assembler using the driver on page 10.

NOTE The file *install_dir*/bin/arcexlib.s contains the assembler definition of the Extensions Library and DSP extensions; it is used by the **mcc** driver when assembling code with the **-X** family of driver options.

When you specify this option, the assembler produces information in the object file that causes the runtime startup initialization to verify the presence of the extension by checking the build-configuration register (BCR).

-Xxmac_24 — Enable 24-by-24 multiply accumulate extension

mcc driver option

Option **-Xxmac_24** enables support for the 24-by-24 multiply accumulate extension. This option defines preprocessor macros **__Xxmac_24** and **__Xxmac** to the compiler and the assembler.

Option **-Xxmac_24** is a driver option; for information on how to use it see Invoking the assembler using the driver on page 10.

NOTE The file *install_dir*/bin/arcexlib.s contains the assembler definition of the Extensions Library and DSP extensions; it is used by the **mcc** driver when assembling code with the **-X** family of driver options.

When you specify this option, the assembler produces information in the object file that causes the runtime startup initialization to verify the presence of the extension by checking the build-configuration register (BCR).

-Xxy — Enable XY memory extension

mcc driver option

Option **-Xxy** enables support for the XY memory extension. This option defines preprocessor macro **__Xxy** to the compiler and the assembler.

Option **-Xxy** is a driver option; for information on how to use it see Invoking the assembler using the driver on page 10.

NOTE The file *install_dir*/bin/arcexlib.s contains the assembler definition of the Extensions Library and DSP extensions; it is used by the **mcc** driver when assembling code with the **-X** family of driver options.

When you specify this option, the assembler produces information in the object file that causes the runtime startup initialization to verify the presence of the extension by checking the build-configuration register (BCR).

Chapter 3 — ELF assembly language

In This Chapter

Lexical features of the ELF assembler

A program for the ELF assembler is made up of statements written in the symbolic machine language specific to the target microprocessor(s).

Statement fields

An assembly language statement contains the following fields:

[*label*:] *opcode* [*operands*] [; *comment*]

The label field

A *label* is a location marker. See Labels on page 28 for more information.

The opcode field

An *opcode* is typically an assembly-language mnemonic for a microprocessor instruction.

For more about assembly-language mnemonics, see the *Programmer's Reference* for your ARC processor.

The opcode field can also contain an assembler directive (also called a pseudo-operation) or a user-defined macro instead of an instruction mnemonic. The opcode field can begin in any column.

See Assembler directives on page 43 for a listing of directives available for the assembler. See Assembler macros on page 37 for instructions on how to write your own macros.

The operands field

An *operand* is an argument for the instruction in the opcode field. Whitespace separates the operands field from the opcode field.

An operand can be an identifier or a constant, or an expression containing one or more identifiers or constants.

The comment field

A *comment* contains information about the statement or group of statements to which it is attached. The comment field is optional. If a comment follows an opcode or operand, whitespace separates it from the opcode or operands field. A comment by itself on a line can begin in any column, provided it is preceded by the comment delimiter.

A comment begins with a semicolon (;) or a pound sign (#).

The comment continues for the rest of the line and ends with a line feed.

The ELF assembler character set

The assembler recognizes the following characters:

- alphabetic characters: A through Z, a through z
- numeric characters (decimal digits): 0 (zero) through 9
- the special characters listed below:

Table 2 Elf assembler character set

Character	Name
&	Ampersand
*	Asterisk
@	At sign

Table 2 Elf assembler character set

Character (con't)	Name (con't)
\	Backslash
^	Caret
:	Colon
,	Comma
$	Dollar sign
"	Double quote
=	Equal sign
!	Exclamation point
<	Left angle bracket
(	Left parenthesis
[	Left square bracket
-	Minus sign
%	Percent sign
.	Period
+	Plus sign
#	Pound sign
?	Question mark
>	Right angle bracket
)	Right parenthesis
]	Right square bracket
;	Semicolon
'	Single quote
/	Slash
	Space
	Tab
~	Tilde
_	Underscore
|	Vertical bar

Some special characters have a predefined function in assembly language:

- A single period (.) represents the current location counter.
- A semicolon (;) marks the beginning of a comment.
- A grave accent or backquote (`) separates statements on a line.
- A backslash (\) at the end of a line is an escaped newline — the line continues onto the next line.

Identifiers

Identifiers are names of variables, labels, functions, macros, registers, and instruction mnemonics. This section describes the rules to which the assembler requires identifiers to conform.

Identifiers may contain the following:

- alphabetic characters: A through Z, a through z
- numeric characters: 0 (zero) through 9
- these special characters:

- $ (dollar sign)
- . (period)
- ? (question mark)
- _ (underscore)
- % (percent sign)

Identifiers may not start with numerals 0 (zero) through 9.

Identifiers are case-sensitive, with the exception of register names, opcode mnemonics, and macro names.

These are examples of valid identifiers:

```
month
.size
L14
_main
display__4DateFv
```

Symbols

A *symbol* is an identifier that can be used as an operand in an assembly statement.

Reserved symbols

The following symbols are *reserved*; that is, you cannot redefine them.

- A period (.) by itself, which represents the location counter
- General-purpose registers `r0` through `r31`, including `gp` (`r26`), `fp` (`r27`), `sp` (`r28`), and, for ARCtangent-A5, `pcl` (`r63`)
- Extension registers `r32` through `r59`
- Loop-count register `lp_count`
- Link registers `ilink1`, `ilink2`, and `blink`
- Names of auxiliary registers, such as `status`, `status32`, `pc`, `lp_end`, `semaphore`, etc.

Using register names as identifiers

By default, the ELF assembler recognizes any register name as a register name. Register names are not case-sensitive, and do not have to be preceded by the % (percent sign).

Any register name preceded by % is unambiguous and can only be a register name; it can never be mistaken for an identifier, because C and C++ do not allow identifier names that start with a percent sign.

To make the assembler accept as a user-defined symbol something that it would otherwise recognize as a register name without the percent sign, use one of these methods:

- Specify option **-%reg** or option **-percent_reg** on the command line.
- Place the directive **.option** "*%reg*" or **.option** "*percent_reg*" in the assembly code.
- Turn on toggle `percent_reg`.

You can then use symbols with names like `r0`, `r5`, and so on in your assembly code.

NOTE By default, assembly source code generated by the MetaWare C/C++ compiler contains only the percent-sign form of register names. The compiler passes option **-%reg** to the assembler to ensure that the assembler accepts names without the percent sign as user-defined identifiers. That is, for compiler-generated assembly code, all names without the percent sign are by default identifier names, not register names.

Built-in symbols

The assembler supports the following built-in symbols:

Table 3 Built-in symbols

Variable	Value
`$architecture`	The string value of the core version number
`$core_version`	Integer value representing the core version specified by option **-core*** or directive **.option**
`$cpu`	The string value "`arc`"
`$endian`	One of two string values, `big` or `little`, depending on the endian mode currently active
`$false`	Integer value `0` (zero)
`$isa`	The string value of the instruction-set architecture. When assembling for ARCtangent-A4 or earlier (**asarc**), the value is `ARC`; when assembling for ARCtangent-A5 or later (**asac**), the value is `ARCompact`.
`$macro`	Name of the macro currently being expanded (see Symbol $macro on page 38 for more information)
`$narg`	Number of arguments with which the current macro was called (see Symbol $narg on page 40 for more information)
`$suffix`	Suffix with which the current macro was invoked (see Symbol $suffix on page 40 for more information)
`$true`	Integer value `1` (one)

These symbols can be used as operands in assembly statements or macro definitions; for example:

```
.ifeqs $endian, "big"
   .set BIG_ENDIAN, $true
.else
   .set BIG_ENDIAN, $false
.endif
```

Built-in functions

A *built-in function* is an identifier that tests for a given condition and returns a value based on the result.

The assembler supports the following built-in functions:

Table 4 Built-in function

Variable	Value
$defined(*name***)**	Return true (`1`) or false (`0`), depending on whether *name* is defined
$reg(*expression***)**	Convert *expression* to a register and return the register. The value of *expression* must map to a valid machine register.
$regval(*reg***)**	Return the numeric value of register *reg*.

Labels

A *label* is a symbol that is associated with an offset within a control section. Such a symbol is "relocatable" because its final address is not known until the linker maps the location of the associated control section.

The operand of a branch instruction is typically a label.

A label is typically associated with the location of an instruction or data specification. You define a label using a definition with the following syntax:

label-name`:` *instruction* `...`

A label definition can begin in any column, but it must be the first item on the line. Place a colon (:) or two colons (::) after *label-name*. Using two colons makes the label global.

Examples of Label Definitions

```
.text
   ...
Func:: push_s  "%blink
   ...
   .data
my_word:     .word 0
my_halfword: .half 0
```

There are three types of labels:

- regular
- local
- numeric

Regular labels

A *regular label* can be defined only once, because it is global to its module and must retain the same value throughout the module.

To make a regular label accessible to other modules, include it in a **.global** or **.comm** directive inside its module, or place a double colon after its definition:

```
main::
```

Names of regular labels follow the general syntax for identifiers, as defined in Identifiers on page 25.

These are examples of regular labels:

```
main:
LOODATA:
L208.day:
display__4DateFv:
```

Local labels

A *local label* begins with a $, followed by one to six decimal digits.

This is an example of a local label:

```
$00010:
```

A local label has limited scope, so you can redefine it as often as you need to. The scope of a local label is one of the following:

- the body of a macro
- the body of a **.rep**, **.irep**, or **.irepc** directive
- a **.include** file

The scope of a local label ends with the next regular label.

An undefined local label causes the assembler to emit an error.

Numeric labels

A *numeric label* is a single digit in the range 0 (zero) to 9; for example:

```
1:      mov %r12, %r15
7:      nop
```

A numeric label has limited scope, so you can redefine it as often as you need to.

A reference to a numeric label consists of a single digit followed by either *b* (for backward) or *f* (for forward):

- *n**b*** refers to the nearest numeric label *n* defined before the reference.
- *n**f*** refers to the nearest numeric label *n* defined after the reference.

For example, the code in Example 1 has the same meaning as the code in Example 2:

Example 1

```
    b 1f
        nop
1:      b 3f
        nop
2:      b 1f
3:      b 2b
1:      nop
```

Example 2

```
    b L1
        nop
L1:     b L3
        nop
L2:     b L1a
L3:     b L2
L1a:    nop
```

The location counter

The *location counter* is a variable in which the address of the current byte being assembled is stored. The assembler uses the location counter to assign addresses to assembled bytes.

You can also make use of the location counter. You use it like any other variable, except that you cannot place it in the label field of an instruction.

The symbol for the location counter is a period (.). When the source code is assembled, the assembler replaces the period with the address of the current byte.

CAUTION	The assembler warns if it must force the alignment of an instruction to a four-byte boundary. It does not warn when you use the location counter to make a jump to a misaligned address; it just jumps to the previous word boundary.

Constants

The assembler recognizes the following types of constants:

- integer constants
- floating-point (real) constants
- string constants

Integer constants

The assembler recognizes binary, decimal, octal, and hexadecimal integer constants. Integer constants have the following prefixes:

Table 5 Integer constants

Base	Prefix	Example
Binary	0B or 0b	0B101011, 0b101011
Octal	0 (zero)	053
Decimal	None	43
Hexadecimal	0X or 0x	0X2B, 0x2b

If an integer has no leading 0 (zero) or other prefix, the assembler assumes it is decimal.

Integer constants can be preceded by a unary plus or minus.

The assembler converts integers of any base to a two's-complement binary representation.

Floating-point (real) constants

The assembler recognizes both standard decimal formats and exponential formats as floating-point constants.

These are all floating-point constants:

```
4.0
3.14159
9e+7
5.374E-4
```

In any floating-point constant, if a decimal point is present, at least one digit must appear to the left of the decimal point. The digit can be 0 (zero).

You can put an underscore before the exponent to increase readability: 1.0782_e+2. Embedded whitespace (space or tab) is not allowed.

Floating-point constants can be preceded by a unary plus or minus.

The assembler converts floating-point constants to an IEEE-format floating-point representation.

Use floating-point constants only with floating-point assembler directives **.float** and **.double**.

NOTE It is possible to use a numeric constant without a decimal point or an exponential designation (for example, 7) with the directives **.float** and **.double**.

The assembler interprets such a constant as an integer, and does not convert it to a floating-point format. This allows you to enter floating-point constants as hex or decimal bit patterns.

String constants

A *string constant* is a sequence of characters of any length, enclosed in double quotes; for example, *"This is a string constant!"*. You use string constants as arguments with the following assembler directives:

.ascii	**.ident**	**.pushsect**	**.string**
.asciz	**.incbin**	**.sbttl**	**.title**
.err	**.include**	**.section**	**.version**
.file	**.print**	**.seg**	**.warn**

Restrictions

String constants can contain any ASCII character, with the following restrictions:

- If single or double quotes are to be interpreted as ordinary characters rather than as delimiters, they must be preceded by a backslash (\): *"This is a \"character\" string."*
- If the backslash is to be interpreted as an ordinary character, it must be preceded by another backslash: *"This \\ is a character."*
- You must express ASCII control characters with these character combinations:

Table 6 Restrictions

Control character	ASCII value	Character combination
Alert	0x07	\a
Backspace	0x08	\b
Form feed	0x0c	\f
Newline	0x0a	\n
Carriage return	0x0d	\r
Tab	0x09	\t
Vertical tab	0x0b	\v
NULL	0x00	\0

NOTE Option **-noesc** inhibits use of the \ character combinations.

Octal or hexadecimal notation

In addition to using ASCII characters themselves in string constants, you can specify the ASCII value of the character in either octal or hexadecimal notation.

- An octal value is expressed as up to three octal characters preceded by a backslash (\).
- A hexadecimal value is expressed as up to two hexadecimal characters preceded by a backslash and a lowercase x.

For example, instead of 'Q' in a string constant, you can use either '\121' or '\x51.'

Expressions

An *expression* is a series of one or more operands (identifiers, constants, and subexpressions) separated by arithmetic, logical, and/or relational operators; for example:

```
index <= 255
a + b
```

Expressions are used as operands with assembler instruction mnemonics and some assembler directives. See the *Programmer's Reference* for your processor for details about specific assembler instruction mnemonics. See Assembler directives on page 43 for information about ELF assembler directives.

Forcing evaluation order

The assembler evaluates an expression from left to right, taking into account the precedence of the operators. Once it has completed the evaluation, the assembler replaces the expression with the resulting value.

You can force the assembler to evaluate the expression in a specific order by enclosing subexpressions in parentheses; these subexpressions are evaluated before the rest of the expression.

Operators and operator precedence

Table 7 Arithmetic operators in order of precedence shows the arithmetic operators supported by the assembler, in their order of precedence, where a lower precedence number indicates a higher precedence of evaluation (that is, number 1 has the highest precedence).

Table 7 Arithmetic operators in order of precedence

Precedence	Operator	Operation
1	**( )**	Overrides precedence of any other operator
2	+	Unary plus
2	-	Unary minus
2	~	Unary bitwise logical NOT
2	**!**	Unary logical NOT
3	/	Division
3	**%**	Modulus
3	*	Multiplication
4	+	Addition
4	-	Subtraction
5	<<	Left shift

Table 7 Arithmetic operators in order of precedence

Precedence	Operator	Operation
5	>>	Right shift
6	<	Less than
6	>	Greater than
6	<=	Less than or equal to
6	>=	Greater than or equal to
7	==	Equal to
7	<>	Not equal to
7	!=	Not equal to
8	&	Bitwise logical AND
9	^	Bitwise logical XOR
10	|	Bitwise logical OR
11	&&	Logical AND
12	^^	Logical exclusive OR
13	||	Logical OR

Assignment syntax

An assignment passes the value of an expression to an identifier.

The assembler recognizes the following forms of assignment syntax:

identifier `=` *expression*
identifier `:=` *expression*
identifier `=:` *expression*

Attributes

Attributes modify references to identifiers. You can use them to indicate different relocation options.

Identifier attributes begin with the “at” sign (@) and use the following syntax:

ident`@`*attribute*

You can also apply an attribute to an expression. When you do so, you must enclose the expression in parentheses:

`(`*expression*`)@`*attribute*

The assembler supports the attributes listed in Table 8 Identifier attributes.

Table 8 Identifier attributes

Attribute	Description	Relocation Entry Generated
`@l`	ARCtangent-A5 and later: four-byte-aligns *ident*	`R_ARC_W`
`@h30`	Right-shifts the location of *ident* by two bits (equivalent to *ident* >> 2). Only the highest 30 bits of *ident*’s location are kept.	`R_ARC_H30`
`@s9`	ARCtangent-A5 and later: Assume that *ident* fits in a signed 9-bit field.	None; informational only

Table 8 Identifier attributes (con't)

Attribute	Description	Relocation Entry Generated
`@s10`	ARCtangent-A5 and later: Assume that *ident* fits in a signed 10-bit field.	None; informational only
`@s11`	ARCtangent-A5 and later: Assume that *ident* fits in a signed 11-bit field.	None; informational only
`@sda`	Offset of an identifier in the `.sdata` or `.sbss` section.	`R_ARC_SDA`
`@sectoff`	Offset of an identifier relative to the start of the section.	`R_ARC_SECTOFF`
`@sectoff_s9`	Offset of an identifier relative to the start of a section; see Note.	
`@sectoff_u8`	Offset of an identifier relative to the start of a section; see Note.	
`@u10`	ARCtangent-A5 and later: Assume that *ident* fits in an unsigned 10-bit field.	None; informational only

NOTE Relocatable addresses, such as label references, can only appear in expressions containing the **+** and **-** operators and **>>** 2. The linker is able to resolve only relocatable expressions containing **+** and **-** offsets and **>>** 2. Any **>>**2 operation must be applied *after* any offset (**+**/**-**) operation, as in `(label+4)>>2`.

Using attributes for customized small-data sections

The assembler supports section-relative symbol attributes `@sectoff_s9` and `@sectoff_u8` to allow hand-customized small-data sections. Using these attributes with load and store instructions allows the instructions to be encoded as 32-bit instructions rather than as 64-bit instructions when the register used as the second operand contains the base address of the section (for `@sectoff_s9`, the base address +256).

For example:

```
ld   r0, [var]                ; 64-bit instruction
                              ; (var is a limm)
ld   r0, [r10, var@sectoff_u8] ; 32-bit instruction
                              ; (var is a shimm)
```

You can use `@sectoff_u8` to access `var` and other variables in the section as long as `r10` (in the example) contains the base address of the section (for `@sectoff_s9`, the base address +256). For maximum benefit, the register should retain the base address for as many accesses as possible.

Choosing an attribute

Attribute `@sectoff_u8` provides access to variables relative to the base address with a maximum addressable region (section) of 256 bytes for ARCtangent-A4 and 1024 bytes for ARCtangent-A5.

Attribute `@sectoff_s9` provides access to variables relative to the base address +256 with a maximum addressable region (section) of 512 bytes for ARCtangent-A4 and 1280 bytes for ARCtangent-A5.

NOTE For ARCtangent-A5 and later, `@sectoff_u8` and `@sectoff_s9` may be used only with load and store instructions.

For ARCtangent-A4, you can also use these attributes as the third operand of an **add** instruction to create pointers to data within the small-data section.

For example:

```
add   r0, r10, var@sectoff_u8  ; 32-bit instruction
                               ;  (var is a shimm)
```

Register `r0` now contains a pointer to `var`.

Load and store instruction ranges for ARCtangent-A5 and later

For ARCtangent-A5 and later, `@sectoff_u8` and `@sectoff_s9` may be used only with load and store instructions. However, load and store instructions have a larger range on ARCtangent-A5, due to the availability of address scaling. The ranges for load and store instructions are shown in Table 9 ARCtangent-A5 and later load and store ranges for @sectoff_u8 and Table 10 ARCtangent-A5 and later load and store ranges for @sectoff_s9.

Table 9 ARCtangent-A5 and later load and store ranges for @sectoff_u8

Instruction	Range	Total
ldb or **stb**	0 to 255	256 bytes
ldw or **stw**	0 to 511	512 bytes
ld or **st**	0 to 1020	1024 bytes

Table 10 ARCtangent-A5 and later load and store ranges for @sectoff_s9

Instruction	Range	Total
ldb or **stb**	-256 to 255	512 bytes
ldw or **stw**	-256 to 511	768 bytes
ld or **st**	-256 to 1020	1280 bytes

The assembler automatically applies address scaling to load and store instructions to facilitate use of the extended ranges.

NOTE To make full use of available space, place one-byte variables first in memory, followed by two byte variables, and then four-byte variables.

Chapter 4 — Assembler macros

In This Chapter

Overview

A *macro* is a named block of assembly-language statements that the assembler inserts automatically into the assembly source code at any point where you put a special statement known as a *macro call*. You define a given macro only once, but you can call it as often as you want. A macro can have formal parameters. You can pass a different value to a macro parameter with each call to the macro.

Defining a macro

A *macro definition* contains the actual code for the macro operation. It consists of three parts: the macro heading, the macro body, and the macro terminator.

Macro heading

You introduce a macro heading with the **.macro** directive, followed by the name of the macro; then any parameters, separated from the macro name and from each other by commas, as in the following example:

```
.macro min_max, num1, num2
```

If you give a macro the name of an assembler instruction or directive, that instruction or directive is redefined to be the macro. You can return the name to its original use by using the **.purgem** directive with the macro name.

> **NOTE** Macro names are not case-sensitive.

For information about the **.macro** and **.purgem** directives, see Assembler directive reference on page 49.

Macro body

The macro body begins with the first assembly-language statement following the **.macro** directive.

The name of any formal parameter you specified with the **.macro** directive can appear in any field in the macro body. If the name of the parameter is embedded in alphanumeric text or in a character string, it must be escaped from the surrounding text with an ampersand preceded by a backslash (\&).

For example, if your macro has a parameter *parm1*, the assembler recognizes *parm1* in each of the following contexts:

```
labl\&parm1: .word   blk\&parm1\&fp
             .print "parm1 = \&parm1\&"
```

At assembly time, the assembler first inserts the macro body in place of the macro call, then replaces each recognized reference to a parameter with the actual value passed to the parameter.

> **NOTE** Macro parameters that occur in comments are not expanded.

Symbol $macro

The symbol `$macro` in a macro definition is replaced at assembly time by the name of the macro currently being expanded. For example, the following macro definition:

```
.macro fred
   .ascii $macro
.endm
```

expands to this at assembly time:

```
.ascii "fred"
```

By default, `$macro` expands as a string. To interpret `$macro` as a symbol instead of as a string, precede the expansion with an escape sequence (\&), as follows:

```
.macro fred
   .long \&$macro
.endm
```

The preceding macro expands to this at assembly time:

```
.long fred
```

Using $macro

Symbol `$macro` is useful primarily when making macro constructors. For example, this macro will not work as expected, due to macro expansion rules:

```
.macro make_macro,arg
   .macro arg
      .ascii  "\&arg"
   .endm
.endm
```

This macro creates a macro constructor, `make_macro`. The following call to `make_macro`:

```
make_macro  fred
```

creates a new macro `fred`, which you would expect to expand to this:

```
.ascii  "fred"
```

However, this is the actual result:

```
.ascii  "arg"
```

Here is the same example, using symbol `$macro`:

```
.macro make_macro,arg
   .macro arg
      .ascii  "\&$macro"
   .endm
.endm
```

In this case, the macro created by this call to `make_macro`:

```
make_macro  fred
```

creates a new macro `fred`, which correctly expands to this:

```
.ascii  "fred"
```

Symbol $narg

The symbol `$narg` used in a macro definition is replaced at assembly time by the number of actual parameters used in a macro call; for example:

```
; Macro INST handles instructions with 0 to 4
; arguments.
.macro INST, mnemonic, arg1, arg2, arg3, arg4
   .if $narg==1
      mnemonic
   .elseif $narg==2
      mnemonic arg1
   .elseif $narg==3
      mnemonic arg1 arg2
   .elseif $narg==4
      mnemonic arg1 arg2 arg3
   .elseif $narg==5
      mnemonic arg1 arg2 arg3 arg4
   .endif
.endm
```

Symbol $suffix

The symbol `$suffix` represents a conditional suffix attached to a macro name. When the macro is invoked with a suffix, `$suffix` is set to equal that suffix. For example, if you invoke macro `INST` as follows, the value of `$suffix` is set to `.ne`:

```
INST.ne a,b
```

You can use `$suffix` to cause any instruction in the macro to execute conditionally by using the format *instruction*`\&$suffix`. When the macro is expanded, `$suffix` is replaced with the conditional suffix, and the instruction executes only if the condition is met. For example, if a macro is defined as follows, the **flag** instruction is expanded to **flag.ne**, and executes only if the `.ne` condition is true:

```
flag\&$suffix h_bit
```

> **NOTE** Invoking a macro with a suffix does not make the expansion of the macro itself conditional.

Macro terminator

The directive **.endm** terminates the macro definition. See Assembler directive reference on page 49 for a discussion of this directive.

Redefining macros

You can redefine one or more macros at any point in the program, but first you must purge any earlier definitions with the **.purgem** directive (see Assembler directive reference on page 49 for a discussion of the **.purgem** directive).

If you redefine a macro without first purging the earlier definition, the assembler emits an error message and aborts.

Calling a macro

To call a macro, insert the macro name and arguments in the opcode field of an assembly statement. Macro names are used like assembler directives, except that they are not preceded by a period:

```
min_max x, y
```

NOTE A macro invocation may include a suffix, for example `min_max.ne.f`. See Defining a macro on page 38 for details of suffixes and the `$suffix` symbol.

Arguments

Arguments in a macro call (also called actual parameters) are separated from one another by commas, and from the macro name by whitespace.

When a macro is expanded, each argument is passed as a character string into the

If an argument contains a comma, you must enclose the argument in angle brackets (<>) or parentheses. If an argument contains a semicolon, you must enclose the argument in angle brackets (<>).

If you omit an argument from the macro call, that argument defaults to a null string when the macro is expanded.

Here is an example of defining and calling a macro with three formal parameters:

```
.macro    mac, a, b, c
   add    a, b, c
.endm
mac       r1, r2, 3+5
mac       r1, r2, <x+3 ; reference external>
```

Macro parameter substitution

The following rules govern macro parameter substitution:

- Formal parameter names in the macro body are replaced by actual parameter strings. That is, declare a macro as follows:

  ```
  .macro min_max, parm1, parm2
  ```

  Then call the macro with the following actual parameters:

  ```
  min_max x, y
  ```

  All instances of *parm1* in the macro body are replaced with *x*, and all instances of *parm2* are replaced with *y*.

- No parameter substitution occurs in comments.
- Formal parameter names concatenated to or embedded in other text must be separated from the adjoining text with an escape sequence (\&); otherwise, the assembler does not recognize them as parameters.
- At macro expansion, all single '\&' sequences are treated as concatenation characters. Once the parameters they delimit are processed, the '\&' sequence vanishes.
- Two \& sequences become a single '\&' sequence, which is then treated as a break character for purposes of substitution.
- Within quoted strings, substitution occurs only if the parameter name is preceded and followed by '\&' sequences.

Nesting and suppressing macros

A macro definition is *nested* if it occurs entirely inside the body of another macro definition. A nested macro is defined when a call is made to the surrounding macro.

NOTE You can call other macros from within the body of a macro, but you cannot recursively call the macro itself.

You can use a nested macro to redefine the surrounding macro, by preceding the new definition with a **.purgem** directive. This is most often done in a conditional situation, where you do not want the macro to operate if certain conditions are present.

You can suppress macro expansion at any point in the macro body with the **.exitm** directive. Any code occurring after this directive is not included in the macro expansion. See Assembler directive reference on page 49 for a discussion of the **.exitm** directive.

Chapter 5 — Assembler directives

In This Chapter

Overview

The ELF assembler supports the directives listed alphabetically in Assembler directive reference on page 49. With assembler directives, also called *pseudo operations*, you can control program organization and manipulate data.

Assembler directives by operation

The tables in this section list assembler directives according to the type of operation they perform:

Directives in each category are listed alphabetically.

Table 11 Directives for conditional assembly

Directive	What it does
.else	Indicate alternative code to be assembled if corresponding .if* condition is false.
.elsec	Indicate alternative code to be assembled if corresponding .if* condition is false.
.elseif	Indicate code to be assembled if conditional expression is true and corresponding .if* condition is false.
.endc	Terminate conditional block.
.endif	Terminate conditional block.
.if	Indicate code to be assembled if conditional expression is true.
.ifdef	Indicate code to be assembled if an identifier is defined.
.ife	Indicate code to be assembled if conditional expression is true.
.ifeq	Indicate code to be assembled if conditional expression is true.
.ifeqs	Indicate code to be assembled if two strings are equal.
.ifn	Indicate code to be assembled if conditional expression is false.
.ifndef	Indicate code to be assembled if an identifier is not defined.
.ifne	Indicate code to be assembled if conditional expression is false.
.ifnes	Indicate code to be assembled if two strings are not equal.
.ifnotdef	Indicate code to be assembled if an identifier is not defined.

Table 12 Directives for data-storage declaration

Directive	What it does
.2byte	Store initialized 16-bit value(s) (half words) in current section.
.3byte	Store initialized 24-bit value(s) in current section.
.4byte	Store initialized 32-bit value(s) (full words) in current section.
.align	Advance current location counter to specified boundary.
.ascii	Place string(s) without terminating null character in current section.
.asciz	Place string(s) with terminating null character in current section.
.block	Generate a block of initialized or uninitialized bytes.
.byte	Store initialized eight-bit value(s) (bytes) in current section.
.double	Store double-precision floating-point constant(s) in current section.
.endian	Change byte order of generated code.
.even	Advance current location counter to an even two-byte boundary.
.float	Store single-precision floating-point constant(s) in current section.
.half	Store initialized 16-bit value(s) (half words) in current section.
.long	Store initialized 32-bit value(s) (full words) in current section.
.short	Store initialized 16-bit value(s) (half words) in current section.
.skip	Generate a block of initialized or uninitialized bytes.
.space	Generate a block of initialized or uninitialized bytes.
.string	Place string(s) with terminating null character in current section.
.word	Store initialized 32-bit value(s) (full words) in current section.

Table 13 Directives for defining extensions

Directive	What it does
.extAuxRegister	Define an extension auxiliary register.
.extCondCode	Define an extension condition code.

Table 13 Directives for defining extensions

Directive	What it does
.extCoreRegister	Define an extension core register.
.extInstruction	Define an extension instruction.

Table 14 Directives for listing control

Directive	What it does
.blank	Insert blank lines in source-code listing.
.eject	Advance listing to top of page.
.lflags	Set listing flags.
.list	Enable source-code listing.
.nolist	Disable source-code listing.
.page	Advance listing to top of page.
.sbttl	Specify subtitle for source-code listing.
.title	Specify main title for source-code listing.

Table 15 Directives for macro definition

Directive	What it does
.define	Define a macro variable.
.endm	Terminate macro definition.
.exitm	Terminate macro expansion.
.ldefine	Declare a local macro variable.
.macro	Declare macro name and parameters.
.purgem	Discard current macro definition.
.undef	Undefine one or more macro variables.

Table 16 Directives for manipulating DWARF 2 CFA information
ARCtangent-A5 and later

Directive	What it does
.cfa_bf	Begin function
.cfa_def_cfa	Define location of CFA
.cfa_def_cfa_register	Define CFA register
.cfa_def_cfa_offset	Define CFA offset
.cfa_ef	End function
.cfa_info	Emit toolset-specific CFA instruction
.cfa_offset	Set register offset rule
.cfa_pop	Adjust offset of stack-pointer CFA for stack pops
.cfa_push	Adjust offset of stack-pointer CFA for stack pushes
.cfa_register	Assign rule and save previous value
.cfa_reg_offset	Set register offset rule
.cfa_remember_state	Save rules for all registers
.cfa_restore	Restore registers to function-entry
.cfa_restore_state	Restore saved register rules

Table 17 Directives for repeat blocks

Directive	What it does
.endr	Terminate repeat block.
.endw	Terminate repeat block initiated by .while directive.

Directive	What it does
.irep	For each item listed, assemble a repeat block and replace identifier with item.
.irepc	For each character in a string, assemble a repeat block and replace identifier with character.
.rep	Assemble a repeat block the specified number of times.
.while	Assemble a repeat block while expression evaluates to true.

Table 18 Directives for section specification

Directive	What it does
.bss	Change current section to `.bss`.
.comm,.common	Define common block (uninitialized block of storage).
.data	Change current section to default `.data` section.
.lcomm,.lcommon	Define local uninitialized block of storage.
.org	Set ELF section address.
.popsect	Pop section stack; restore most recently pushed section.
.previous	Resume prior section.
.pushsect	Push current section onto section stack; switch to new section.
.rdata	Change current section to default read-only data section.
.rodata	Change current section to default read-only data section.
.sectflag	Set the `SHF_*` flags field of the specified section.
.section	Define control section and type.
.sectionflag	Set the `SHF_*` flags field of the specified section.
.sectionlink	Set the link field of one section to point to another section.
.sectlink	Set the link field of one section to point to another section.
.seg	Define control section and type.
.segflag	Set the `SHF_*` flags field of the specified section.
.seglink	Set the link field of one section to point to another section.
.text	Change current section to default `.text` section.

Table 19 Directives for symbol declaration and binding

Directive	What it does
.eflags	Bitwise **OR** a value with the `e_flags` field of the ELF header.
.entry	Set the `ENTRY` ELF assembler binding.
.equ	Assign a value to an identifier.
.extern	Designate a symbol as external.
.global,.globl	Export symbol(s).
.reloc	Specify relocation of the next instruction in the current section.
.set	Assign a value to an identifier.
.weak	Specify weak ELF assembler binding.

Table 20 Miscellaneous directives

Directive	What it does
.assert	Print an error message if assertion fails.
.end	Terminate assembly.
.err	Print an error message.
.file	Specify a source-file name.
.ident	Place string(s) into comment section of the object file.
.incbin	Include specified file in binary form.
.include	Include specified source file.
.line	Identify line number.

Table 20 Miscellaneous directives

Directive	What it does
.offwarn	Turn off selected warnings
.onwarn	Turn on selected warnings
.option	Specify an assembly option
.print	Print string to standard output.
.size	Specify size of a symbol in bytes.
.type	Specify a type.
.version	Place string(s) in comment section of the object file.
.warn	Print a warning message.

Assembler directive reference

The rest of this chapter describes individual assembler directives. The directives are listed alphabetically.

The names of assembler directives are not case-sensitive. The assembler recognizes directive names with or without the dot (.) prefix, unless the name of the directive is defined as a macro

NOTE Some directives listed in this section are identified as preprocessor directives and are processed by the macro preprocessor before assembly.

.2byte *expression*[, *expression*, ...] — Store initialized 16-bit value(s) (half words) in current section

Stores initialized half-word values (16-bit two's-complement) in the current section.

Assembles *expression* arguments into consecutive half words. For example, this directive initializes five consecutive half words; the first half-word has the value of expression *val_1*, the second has the value of *val_2*, and so on:

`.2byte` *val_1*, *val_2*, *val_3*, *val_4*, *val_5*

.2byte allows misalignment; the assembler does not warn.

This directive is not valid for the `.bss` section.

Synonym for .half and .short.

.3byte *expression*[, *expression*, ...] — Store initialized 24-bit value(s) in current section

Stores initialized values (24-bit two's-complement) in the current section.

Assembles *expression* argument(s) into consecutive 24-bit blocks. For example, the following directive initializes four consecutive 24-bit blocks; the first block has the value of expression *val_1*, the second has the value of *val_2*, and so on:

`.3byte` *val_1*, *val_2*, *val_3*, *val_4*

.3byte allows misalignment; the assembler does not warn.

This directive is not valid for the `.bss` section.

.4byte *expression*[, *expression*, ...] — Store initialized 32-bit value(s) (full words) in current section

Stores initialized values (32-bit two's-complement) in the current section.

Assembles *expression* argument(s) into consecutive 32-bit blocks. For example, the following directive initializes four consecutive 32-bit blocks; the first block has the value of expression *val_1*, the second has the value of *val_2*, and so on:

`.4byte` *val_1*, *val_2*, *val_3*, *val_4*

.4byte allows misalignment; the assembler does not warn.

This directive is not valid for the `.bss` section.

Synonym for **.long** and **.word**.

.align *boundary* — Advance location counter to specified boundary

Advances the location counter to the next boundary specified by *boundary*, which can be any positive integer power of two:

1	Byte alignment (8 bits)
2	Half-word alignment (16 bits)
4	Word alignment (32 bits)
8	8-byte alignment
16	16-byte alignment
...	etc.

Use the **.align** directive after a **.ascii**, **.block**, **.byte**, **.half**, or **.word** directive to force proper alignment of the next data item.

.align *2* is the same as **.even**.

.ascii *string*[, *string*, ...] — Place string(s) without terminating null character in current section

Stores *string*(s) without a terminating null (\0) character in the current section. Each string must be enclosed in double quotes.

The **.ascii** directive leaves the location counter positioned after the last character in the string, whether or not it is on the current byte boundary. This means that if the next directive is **.block**, **.double**, **.float**, **.half**, or **.word**, the assembler emits a warning that the data item will be misaligned. It is a good idea to follow a **.ascii** directive with a **.align** directive to force proper alignment of the next data item to be stored.

This directive is not valid for the `.bss` section.

.asciz *string*[, *string*, ...] — Place string(s) with terminating null character in current section

Places *string*(s) in the current section followed by a terminating null (\0) character.

This directive is not valid for the `.bss` section.

Synonym for **.string**.

.assert *expression* — Print an error message if assertion fails
Evaluates *expression*; if *expression* evaluates to 0 (false), generates an error message.

.blank *expression* — Insert blank lines in source-code listing
Tells the assembler to insert the number of blank lines specified by *expression* in the source listing. *expression* must evaluate to an absolute integer value.

Directive **.blank** works only if the assembler was invoked with option **-l**.

.block *number* — Generate a block of initialized or uninitialized bytes
Skips *number* bytes in the current section.

- If the section is a `.bss` data section, the bytes are left uninitialized.
- If the section is a text section, **nop** instructions are placed in the bytes.
- In all other sections, zeros are placed in the bytes.

This directive is valid for the `.bss` section.

Synonym for **.skip** and **.space**.

.bss — Change current section to .bss
Changes the current section to `.bss`, the default BSS section.

.byte *expression*[, *expression*, ...] — Store initialized eight-bit value(s) (bytes) in current section
Stores initialized bytes in the current section. For example, the following directive initializes three consecutive bytes; the first byte has the value of expression *val_1*, the second has the value of *val_2*, and so on:

`.byte` *val_1*, *val_2*, *val_3*

This directive is not valid for the `.bss` section.

.cfa* — Canonical frame address directives
For ARCtangent-A5 and later
For a listing of the optional DWARF 2 CFA directives supported, see CFA directive reference on page 74.

.comm *name*, *expression*[, *align*]
.common *name*, *expression*[, *align*] — Define a common block (uninitialized block of storage)
Defines an uninitialized block of storage, called a *common block*, in the `.data` section. This block can be common to more than one module.

name	block name; references the block of storage
expression	block size, in bytes; must be a positive integer
align	specifies the alignment, which must be a positive power of 2; see **.align**

This directive is valid for the `.bss` section.

.data — Change current section to default .data section

Changes the current section to `.data`, which is the default data section.

.define *name*[, {*string* | *integer*}] — Define a global macro variable

Preprocessor directive **.define** defines a macro variable (*name*) for use during macro processing. The value of *name* can be either a character string or an integer constant. If you do not specify a string or integer, the value of *name* defaults to `1` (one). *name* is global to the whole program.

During macro processing, the preprocessor replaces occurrences of the macro variable *name* in the source code with its defined value.

Using **.define** does not add *name* to the symbol table; to add *name* to the symbol table, use **.set** or an explicit assignment (s = e).

To redefine *name*, first undefine it with an **.undef** directive, then use it in another **.define** directive. (The assembler emits a warning if you redefine an already defined macro variable.)

Compare **.define** to the **.ldefine** directive, which defines a macro variable local to the current macro expansion.

NOTE If you use **.define** to redefine a variable that was previously defined with **.ldefine**, **.define** redefines the variable within the most recently declared scope.

.define name redefines name even if name was not defined in the same scope as the **.define** statement. **.ldefine** only looks in the current scope; if the variable is not found, **.ldefine** defines it.

For more information, see Assembler macros on page 37.

.double *floating_constant*[, *floating_constant* ...] — Store double-precision floating-point constant(s) in current section

Stores one or more double-precision floating-point constants in the current section. *floating_constant* is converted to floating-point if necessary.

.double allows misalignment; the assembler does not warn.

This directive is not valid for the `.bss` section.

.eflags *expression* — Bitwise OR a value with the e_flags field of the ELF header

The **.eflags** directive directs the assembler to evaluate *expression*, and combine the result, using a logical OR operation, with the `e_flags` field of the ELF header. The `e_flags` field holds processor-specific flags associated with the object file.

.eject — Advance listing to top of page

Tells the assembler to move to the top of the next page in the source listing form. For example, you can use **.eject** to start the listing of each subroutine on a new page.

Directive **.eject** works only if the assembler was invoked with option **-l**.

Synonym for **.page**.

.elif — *conditional_expression* — indicate code to be assembled if conditional expression is true and corresponding .if* condition is false

Synonym for **.elseif**.

.else — Indicate code to be assembled if corresponding .if* condition is false

Preprocessor directive

Indicates code to be assembled if the conditional expression of the corresponding **.if*** directive evaluates to false (zero). In that case, the assembler assembles the code following the **.else** directive instead of the code following the **.if*** directive.

Synonym for **.elsec**.

.elsec — Indicate code to be assembled if corresponding .if* condition is false

Preprocessor directive

Synonym for **.else**.

.elseif *conditional_expression* — Indicate code to be assembled if conditional expression is true and corresponding .if* condition is false

Preprocessor directive

Indicates that all code between the **.elseif** directive and the corresponding **.else**, **.elseif**, or **.endif** directive is to be assembled if both of the following conditions are met:

- The conditional expression of the corresponding **.if*** directive evaluates to false (zero).
- *conditional_expression* evaluates to true (non-zero).

Otherwise, the next **.else**, **.elseif**, or **.endif** directive is processed.

.end — Terminate assembly

Signals the assembler to terminate assembly and ignore all code after the **.end** directive. The assembler warns if anything but whitespace follows the **.end** directive.

.endc — Terminate conditional block

Preprocessor directive

Marks the end of a conditional block. If you nest **.if*** directives, an **.endc** (or **.endif**) directive is paired with the most recent **.if*** directive.

Synonym for **.endif**.

.endian {*big*|*little*} — Change byte order of generated code

Enables you to change the byte order of generated code by specifying either big-endian or little-endian mode.

.endif — Terminate conditional block

Preprocessor directive

Synonym for **.endc**.

.endm — Terminate macro definition

Preprocessor directive

Marks the end of a macro definition begun with the previous **.macro** directive. An **.endm** directive is paired with the most recent **.macro** directive. Therefore, if another **.macro** directive occurs before the

.endm directive, the macro initiated by the second **.macro** directive is nested inside the first macro; it must be terminated by an **.endm** directive of its own. See Nesting and suppressing macros on page 42 for a discussion of nested macros.

For more information, see Assembler macros on page 37.

.endr — Terminate repeat block
Preprocessor directive
Terminates a repeat block initiated by the **.rep**, **.irep**, or **.irepc** directive.

.endw — Terminate repeat block
Preprocessor directive
Terminates a repeat block initiated by the **.while** directive.

.entry *name*[, *name* ...] — Set the ENTRY ELF binding
Sets the `ENTRY` ELF binding for a symbol.

.equ *name*, *expression* — Assign a value to an identifier
Assigns the value of *expression* to *name*. *expression* is an absolute or relocatable value.

This is the same as using an assignment operator:

name `=` *expression*

A given identifier should appear in only one **.equ** statement, because the **.equ** assignment is constant for the whole program.

If you want a dynamic rather than a constant assignment, use directive **.set**.

.err ["*string*"] — Print an error message
Causes the assembler to print an error message to `stderr`, or to `stdout` if you have specified option **-Eo**. Also increments the error count.

You can optionally specify a string to include in the error message. The string must be enclosed in double quotes.

.even — Advance location counter to an even two-byte boundary
Advances the location counter to the next even two-byte boundary.

.even is the same as **.align** *2*.

.exitm — Terminate macro expansion
Preprocessor directive
Causes macro expansion to stop and all code between the **.exitm** directive and the **.endm** directive of the macro to be ignored. The **.exitm** directive is generally used with a **.if*** directive to test for a particular condition and abort macro expansion if the condition occurs.

The **.exitm** directive terminates expansion of only the macro in which it appears. If macros are nested, **.exitm** returns code generation to the previous nesting level.

For more information, see Assembler macros on page 37.

.extAuxRegister *name*, *loc*, *mode* — Define an extension auxiliary register
Defines a new auxiliary register in your program. For information, see the following:

- Using extension directives on page 67
- The listing for directive **.extAuxRegister** in Extension directive reference on page 68

.extCondCode *suffix*, *value* — Define an extension condition code
Defines a new condition code value to be used as a suffix on instructions in source code. For information, see the following:

- Using extension directives on page 67
- The listing for directive **.extCondCode** in Extension directive reference on page 68

.extCoreRegister *name*, *rnum*, *mode*, *shortcut* — Define an extension core register
Defines a new core register to be available in your program. For information, see the following:

- Using extension directives on page 67
- The listing for directive **.extCoreRegister** in Extension directive reference on page 68

.extern *name* — Designate a symbol as external
Identifies a symbol defined in an external module. Because undefined symbols are assumed to be external symbols, you do not need to use this directive.

.extInstruction *name*, *opcode*, *sub_opcode*, *suffix_classes*, *syntax_class* — Define an extension instruction
Adds a new instruction to the set of valid assembler instructions. For information, see the following:

- Extension directive reference on page 68
- The listing for directive **.extInstruction** in Extension directive reference on page 68

.file *name* — Specify a source-file name
Identifies the name of a source file.

.float *floating_constant*[, *floating_constant*, ...] — Store single-precision floating-point constant(s) in current section
Stores one or more single-precision floating-point constants in the current section. *floating_constant* is converted to floating-point if required.

.float allows misalignment; the assembler does not warn.

This directive is not valid for the `.bss` section.

.global *name*[, *name* ...]
.globl *name*[, *name* ...] — Export symbol(s)
Exports one or more *name* symbols.

.half *expression*[, *expression*, ...] — Store initialized 16-bit value(s) (half words) in current section
Synonym for **.2byte** and **.short**.

.ident *string*[, *string*, ...] — Place string(s) in comment section of the object file

Places one or more strings in the comment section of the object file.

Synonym for **.version**.

.if *conditional_expression* — Indicate code to be assembled if conditional expression is true

Preprocessor directive

Indicates that all code between the **.if** directive and the corresponding **.else**, **.elseif**, or **.endif** directive is to be assembled if *conditional_expression* evaluates to true (non-zero). Otherwise, the next **.else**, **.elseif**, or **.endif** directive is processed.

Synonym for **.ife**, **.ifeq**.

.ifdef *name* — Indicate code to be assembled if an identifier is defined

Preprocessor directive

Indicates that all code between the **.ifdef** directive and the corresponding **.else**, **.elseif**, or **.endif** directive is to be assembled if *name* is defined. If *name* is not defined, the next **.else**, **.elseif**, or **.endif** directive is processed. *name* can be either an assembler variable or a macro variable defined with the **.define** directive.

.ife *conditional_expression* — Indicate code to be assembled if conditional expression is true

Preprocessor directive

Synonym for **.if**, **.ifeq**.

.ifeq *conditional_expression* — Indicate code to be assembled if conditional expression is true

Preprocessor directive

Synonym for **.if**, **.ife**.

.ifeqs "*string1*", "*string2*" — Indicate code to be assembled if two strings are equal

Preprocessor directive

Indicates that all code between the **.ifeqs** directive and the corresponding **.else**, **.elseif**, or **.endif** directive is to be assembled if *string1* is equal to *string2*. If the strings are not equal, the next **.else**, **.elseif**, or **.endif** directive is processed. The character strings must be enclosed in double quotes.

.ifn *conditional_expression* — Indicate code to be assembled if conditional expression is false

Preprocessor directive

Indicates that all code between the **.ifn** directive and the corresponding **.else**, **.elseif**, or **.endif** directive is to be assembled if *conditional_expression* evaluates to false (zero). Otherwise, the next **.else**, **.elseif**, or **.endif** directive is processed.

Synonym for **.ifne**.

.ifndef *name* — Indicate code to be assembled if an identifier is not defined

Preprocessor directive

Indicates that all code between the **.ifndef** directive and the corresponding **.else**, **.elseif**, or **.endif** directive is to be assembled if *name* is not defined. If *name* is defined, the next **.else**, **.elseif**, or **.endif** directive is processed. *name* can be either a regular assembler variable or a macro variable defined with the **.define** directive.

Synonym for **.ifnotdef**.

.ifne *conditional_expression* — Indicate code to be assembled if conditional expression is false

Preprocessor directive

Synonym for **.ifn**.

.ifnes "*string1*", "*string2*" — Indicate code to be assembled if two strings are not equal

Preprocessor directive

Indicates that all code between the **.ifnes** directive and the corresponding **.else**, **.elseif**, or **.endif** directive is to be assembled if *string1* is not equal to *string2*. If the strings are equal, the next **.else**, **.elseif**, or **.endif** directive is processed. The character strings must be enclosed in double quotes. Use this directive inside macros to test macro parameters.

.ifnotdef *name* — Indicate code to be assembled if an identifier is not defined

Preprocessor directive

Synonym for **.ifndef**.

.incbin "[*pathname*]*file_name*" — Include specified file in binary form

Preprocessor directive

Includes *file_name* in binary form in the generated object file. The assembler searches for *file_name* first in the current directory, then in any directories listed with command-line option **-I**.

Directive **.incbin** differs from **.include** in that any file included with **.incbin** must meet one of the following criteria:

- It must have an explicit location (that is, its path name must be fully qualified).
- It must be relative to the current directory.
- It must have its path specified with command-line option **-I**.

Files included with **.incbin** are included as-is. There is no endian swapping or special alignment, so take the following precautions:

- If you want a file included with **.incbin** to be in a particular section, you must specify the section with the **.section** directive before the **.incbin** directive.
- If you want the contents of the file included with **.incbin** to be aligned in the object file, you must precede the **.incbin** directive with a **.align** directive.
- If you need alignment after the **.incbin** section, you must add a **.align** directive after the **.incbin** directive. For example, the following code might produce unexpected results if the size of the file `abc.bin` is not a multiple of four bytes:

  ```
  b        label1
  .incbin  "abc.bin"
  add      r1,r2,r3
  ```

  To correct the problem, add **.align** *4* after the **.incbin** directive.

.include "[*pathname*]*file_name*" — Include specified source file

Preprocessor directive

Instructs the assembler to include the specified source file in the input source-code stream at assembly time. The file name and the pathname, if any, must be enclosed in double quotes.

If you assemble with command-line option **-I**, the assembler looks for **.include** files with non-absolute path names first in the current directory, then in the *pathname* directory. If the assembler cannot find the specified source file, it emits an error message and aborts.

The assembler passes the *pathname* and *file_name* specifications to the host operating system without any conversion from lowercase to uppercase.

NOTE Included source files can contain **.include** directives of their own, as can macros.

file_name should be in the form described in String constants on page 31. We recommend using forward slashes as path separators, because they are supported on all platforms. However, you can use backslashes, as long as you use two backslashes to represent each single backslash in the path name.

.irep *identifier*, *item*[, *item* ...] — For each item listed, assemble a repeat block and replace identifier with item

Preprocessor directive

Tells the assembler to assemble instructions up to the next **.endr** directive once for each *item* listed. On each pass, the corresponding *item* replaces *identifier* in the instruction sequence.

Separate any occurrence of *identifier* from adjacent text with a '\&' (backslash-ampersand) sequence; for example:

```
            .irep  init_val, 10, 20, 30
l\&init_val: .2byte  init_val
            .endr
```

The assembler converts this instruction sequence to the following:

```
l10:        .2byte   10
l20:        .2byte   20
l30:        .2byte   30
```

If *item* contains a comma or semicolon, enclose *item* in angle brackets, <*like;this*>.

.irepc *identifier*, "*string*" — For each character in a string, assemble a repeat block and replace identifier with character

Preprocessor directive

Tells the assembler to assemble instructions up to the next **.endr** directive once for each character in *string*. *string* must be enclosed in double quotes. On each pass, the corresponding character replaces *identifier* in the instruction sequence.

Separate any occurrence of *identifier* from adjacent text with a '\&' (backslash ampersand) sequence; for example:

```
            .irepc ch, "XYZ"
l_\&ch:     .byte '\&ch\&'
            .endr
```

The assembler converts this instruction sequence to the following:

```
l_X:        .byte 'X'
l_Y:        .byte 'Y'
l_Z:        .byte 'Z'
```

If *string* is empty (" "), no instruction sequences are assembled.

.keep [*symbol*[, *symbol*, ...]] — Place local symbol(s) in the ELF symbol table

Directs the assembler to place local symbols in the ELF symbol table.

If you specify directive **.keep** without arguments, the assembler places all local symbols in the ELF symbol table. If you specify *symbol*, the assembler places in the symbol table only the local symbol(s) you specify.

Compare to command-line option **-L**, which directs the assembler to place all local symbols in the ELF symbol table.

.lcomm *name*, *expression*[, *align*]

.lcommon *name*, *expression*[, *align*] — Define local uninitialized block of storage

Defines a local uninitialized block of storage in the `.bss` section. The result defines *name* as a `bss` symbol.

name	block name; references the storage, cannot be predefined
expression	block size; must be a positive integer
align	specifies the alignment; must be a positive integer and a power of 2; see **.align**

This directive is valid for the `.bss` section.

.ldefine *name*[, {*string* | *integer*}] — Define a local macro variable

Preprocessor directive

Defines a local macro variable, *name*, for use during macro processing. The value of *name* can be either a character string or an integer constant. If you do not specify a string or integer, the value of *name* defaults to `1` (one). If *name* is already defined in the current scope, **.ldefine** redefines it.

During macro processing, the preprocessor replaces occurrences of *name* in the source code with its defined value.

Compare to directive **.define**, which defines a macro variable global to the whole program.

NOTE If you use **.define** to redefine a variable that was previously defined with **.ldefine**, **.define** redefines the variable within the most recently declared scope.

.define name redefines name even if name was not defined in the same scope as the **.define** statement. **.ldefine** only looks in the current scope; if the variable is not found, **.ldefine** defines it.

For more information, see Assembler macros on page 37.

.lflags *flag*[*flag* ...] — Set listing flags

Sets listing flags that control the listing of the source file. See the listing for option **-f** for the flags and their default settings. **.lflags** performs the same function as command-line option **-f**, except that you can apply the directive to portions of a module, whereas option **-f** affects the entire module.

NOTE To use the **.lflags** directive, you must also specify option **-l** on the command line.

Setting and toggling listing flags

To turn On a flag, list it after the **.lflags** directive. To turn Off a flag, list it after the **.lflags** directive and put an 'n' before the flag. To set the *l* and *p* flags, insert an integer value after them in the flag list.

For example, the following **.lflags** directive:

```
.lflags cgnmp66
```

tells the assembler to do the following:

- Turn On flags *c* and *g*.
- Turn Off flag *m*.
- Set the value of flag *p* to 66.

.line *number* — Identify line number

Identifies a line number.

The assembler accepts this directive but ignores it.

.list — Enable source-code listing

Tells the assembler to output a source-code assembly listing. Every program begins with an implicit **.list** directive, but the assembler generates the listing only if you also specify assembler command-line option **-l**.

Alternate the **.list** directive with the **.nolist** directive to list selected portions of a program.

.long *expression*[, *expression*, ...] — Store initialized 32-bit value(s) (full words) in current section

Synonym for **.4byte** and **.word**.

.macro *name*, *param*[, *param* ...] — Declare macro name and parameters

Preprocessor directive

Declares the name and formal parameters of a macro and marks the beginning of the macro definition. Code following the **.macro** directive, down to the corresponding **.endm** directive, constitutes the body of the macro *name*.

> **NOTE** Macro names are not case-sensitive.

The names of the formal parameters are recognized only within the macro definition. These names can be used for other purposes outside the macro.

The actual parameters in the call to the macro are matched to the formal parameters in the macro declaration, starting with the left-most of each. There can be fewer actual parameters than formal parameters. Formal parameters that lack a corresponding actual parameter default to the null string.

For more information, see Assembler macros on page 37.

.nolist — Disable source-code listing

Disables assembly source-code listing, except for lines flagged with errors.

Directive **.nolist** works only if the assembler was invoked with option **-l**.

.offwarn=*warning_no*[,*warning_no*...] — Turn off selected warnings

Disables selected warnings. The argument to this directive is a comma-separated list of warning numbers, which you can obtain from the warnings. If a warning does not include a warning number, the warning cannot be disabled.

To reenable warnings, use **.onwarn**.

.onwarn=*warning_no*[,*warning_no*...] — Turn on selected warnings

Reenables selected warnings. The argument to this directive is a comma-separated list of warning numbers.

.option *option* [,*option* ...] — Specify an assembly option

Enables you to specify an assembly option. These are the valid arguments:

a4	Specify the ARCtangent-A4 processor series (same as *isa_arc*)
a5	Specify the ARCtangent-A5 processor series
a6 or *arc600*	Specify the ARC 600 processor series
a7 or *arc700*	Specify the ARC 700 processor series
core1	Support ARCtangent-A5 and later core version 1
core2	Support ARCtangent-A5 core version 2
core3	Support ARCtangent-A5 core version 3
core5	Support core version 5 (prior to ARCtangent-A4)
core6	Support core version 6 (prior to ARCtangent-A4)
core7	Support core version 7 (prior to ARCtangent-A4)
core8	Support core version 8 (ARCtangent-A4)
esc	Enable backslash (C-style) escape character.
extregs	Define all extension core registers with the default names `%r32`, `%r33`, etc.
isa_ac	Assemble for ARCompact ISA (ARCtangent-A5 and later)
isa_ac32	Assemble for ARCompact ISA (ARCtangent-A5 and later) using only 32-bit encodings
isa_arc	Assemble for ARCtangent-A4 ISA
maths	Evaluate expressions using signed math
mathu	Evaluate expressions using unsigned math
noesc	Disable backslash (C-style) escape character

percent_reg	Specify that register names must begin with a percent sign
%reg	Specify that register names must begin with a percent sign
Xmult32	Recognize 16-bit ARCtangent-A5 multiply extension instructions
Xbarrel_shifter	Recognize 16-bit ARCtangent-A5 barrel-shifter extension instructions

See Command-line options on page 11 for more detailed information about individual options.

NOTE Options set with directive **.option** take precedence over options set at the command line.

.org *address* — Set ELF section address

Preassigns link-time *address* to a section.

NOTE Directive **.org** can be used only once per section.

.page — Advance listing to top of page

Synonym for **.eject**.

.popsect — Pop section stack; restore most recently pushed section

Pops the section stack, making the current section be the section most recently pushed on the section stack.

.previous — Resume prior section

Resumes the section that was active prior to the current section.

.print ["*string*"] — Print string to standard output

Causes the assembler to print a string to standard output. The string must be enclosed in double quotes. *string* is optional; if you do not specify a string, directive **.print** outputs a newline.Use the **.print** directive to output an error message, with a **.if*** directive to test for a user-defined error condition and the **.err** directive to flag the error.

.purgem *name*[, *name* ...] — Discard current macro definition

Preprocessor directive

Discards current macro definition of all macros listed as arguments. Macros are expanded for all calls up to the **.purgem** directive.

For more information, see Assembler macros on page 37.

.pushsect *name*[, *class*][, *entsize*][, *min_align*] — Define and push section onto section stack; switch to new section

Directive **.pushsect** is equivalent to directive **.section**, except that **.pushsect** pushes the current section on the stack before switching the current section to section *name*. To discontinue assembly to section *name* and restore the previous section, use **.popsect**.

class is one of the following attributes:

bss	The section contains writable data initialized to 0 (zero).
comdat	The section contains common code or data.
data	The section contains writable data.
directive	The section contains additional linker command-line arguments.
lit	The section contains read-only data.
note	The section contains special information for the operating system.
rodata	The section contains read-only data.
text	The section contains executable instructions.

class can also be a string of characters, used singly or in combination, with the following meanings:

a	The section occupies memory during execution.
w	The section is writable.
x	The section is executable.

The *entsize* argument is an integer indicating the size in bytes of entries in the section. Specify *entsize* if the section contains a table of entries of fixed size. The default is 1 (one). *entsize* sets the `sh_entsize` field of the section in the ELF section header.

The *min_align* argument specifies the minimum alignment required for the section. For executable sections the default is 4 (four); for all others the default is one (1).

.rdata — Change current section to default read-only data section

Changes the current section to `.rodata`, the default read-only data section.

Synonym for **.rodata**.

.reloc *symbol*, *reltype* — Specify relocation for the next word in the current section

Specifies the relocation type of the next word to be assembled in the current section. **.reloc** adds a relocation entry to the relocation table, using the identifier *symbol*. The relocation type is designated by *reltype*, an integer value.

.removecore {*reglist*} — Remove one or more registers

Removes one or more registers specified in argument *reglist*, where *reglist* is a comma-separated list of registers or register ranges, for example:

```
.removecore {r18,r20-r23}
```

The registers listed are removed from the assembler and are assumed to be removed from the processor core.

CAUTION	Objects that contain **.removecore** directives cannot be linked with objects that use the removed registers.

.rep *n* — Assemble a repeat block the specified number of times

Preprocessor directive

Tells the assembler to duplicate an instruction sequence ending with the next **.endr** directive the number of times represented by *n*. *n* must evaluate to an absolute integer value.

.rodata — Change current section to default read-only data section

Synonym for **.rdata**.

.sbttl "*subtitle*" — Specify subtitle for source-code listing

Specifies a subtitle for the source-code listing. The subtitle string must be enclosed in double quotes. The subtitle appears below the main title at the top of each page of the source listing. When you specify a new subtitle, it appears on the page immediately following the page that contains the **.sbttl** directive.

Directive **.sbttl** works only if the assembler was invoked with option **-l**.

.sectflag *section_name*, *flag* — Set the SHF_* flags field of the specified section

Sets the SHF_* flags field of the specified section *section_name*. The recognized values of *flag* are as follows:

begin	sets the SHF_BEGIN flag
end	sets the SHF_END flag

section_name and *flag* can be either identifiers or quoted strings.

Synonym for **.sectionflag** and **.segflag**.

.section *name*[, *class*][, *entsize*][, *min_align*] — Define control section

Defines a control section *name* of type *class*, and arranges for subsequent code to be placed within the control section.

After the section has been defined, you can reactivate it at a later time by respecifying the **.section** directive. The attributes are unnecessary when you reactivate the section.

class is one of the following attributes:

bss	The section contains writable data initialized to 0 (zero).
comdat	The section contains common code or data.
data	The section contains writable data.
directive	The section contains additional linker command-line arguments.
lit	The section contains read-only data.

`note`	The section contains special information for the operating system.
`rodata`	The section contains read-only data.
`text`	The section contains executable instructions.

class can also be a string of characters, used singly or in combination, with the following meanings:

`a`	The section occupies memory during execution.
`w`	The section is writable.
`x`	The section is executable.

The *entsize* argument is an integer indicating the size in bytes of entries in the section. Specify *entsize* if the section contains a table of entries of fixed size. The default is 1 (one). *entsize* sets the `sh_entsize` field of the section in the ELF section header.

The *min_align* argument specifies the minimum alignment required for the section. For executable sections the default is 4 (four); for all others the default is one (1).

Synonym for **.seg**.

.sectionflag *section_name*, *flag* — Set the SHF_* flags field of the specified section

Synonym for **.sectflag** and **.segflag**.

.sectionlink *section_a*, *section_b* — Set the link field of one section to point to another section

Sets the link field of *section_a* to point to *section_b*. Both sections must have been previously defined. The section-linking feature is necessary for COMDAT support. (See the *MetaWare C/C++ Language Reference* for information about COMDAT.)

Synonym for **.sectlink** and **.seglink**.

.sectlink *section_a*, *section_b* — Set the link field of one section to point to another section

Synonym for **.sectionlink** and **.seglink**.

.seg *name*[, *class*][, *entsize*] — Define control section, type, and size of entry

Synonym for **.section**.

.segflag *section_name*, *flag* — Set the SHF_* flags field of the specified section

Synonym for **.sectflag** and **.sectionflag**.

.seglink *section_a*, *section_b* — Set the link field of one section to point to another section

Synonym for **.sectionlink** and **.sectlink**.

.set *name*, *expression* — Assign a value to an identifier

Assigns the value of *expression* to *name* and adds *name* to the symbol table. *expression* is an absolute or relocatable value.

.set is a dynamic assignment operator. That is, you can use it to set the value of an identifier repeatedly, instead of just once for the entire assembly. This means that an identifier's value, if it is set with **.set**, can be different at different points in the execution.

If you want a constant rather than a dynamic assignment, use directive **.equ**.

.short *expression*[, *expression*, ...] — Store initialized 16-bit value(s) (half words) in current section

Synonym for **.2byte** and **.half**.

.size *name*, *expression* — Specify size of a symbol in bytes

Sets the size (in bytes) for symbol *name* to *expression*. This value is passed to the linker.

.skip *number* — Generate a block of initialized or uninitialized bytes

Synonym for **.block** and **.space**.

.space *number* — Generate a block of initialized or uninitialized bytes

Synonym for **.block** and **.skip**.

.string *string*[, *string*, ...] — Place string(s) with terminating null character in current section

Synonym for **.asciz**.

.text — Change current section to default .text section

Changes the current section to the default `.text` section, which consists of executable code.

.title "*main_title*" — Specify main title for source-code listing

Specifies a main title for the source-code listing. The title string must be enclosed in double quotes. The title appears at the top of each page of the source listing.

The default title defined by the assembler is blank.

For the title you specify to appear on the first page of the source listing, you must make the **.title** directive the first statement in the program, before all other lines, including comments.

Directive **.title** works only if the assembler was invoked with option **-l**.

.type *name*, *type* — Specify a type

Associates *type* with *name*. Type information is passed to the linker.

type can be one of the following:

```
@function    "function"
@import      "import"
@no_type     "no_type"
@object      "object"
```

For example:

```
.type my_function, @function
```

.undef *name*[, *name* ...] — Undefine one or more macro variables

Preprocessor directive

Undefines one or more macro variables.

If you undefine an identifier that has not been previously defined, the assembler does not emit a warning.

For more information about macros, see Assembler macros on page 37.

.version *string*[, *string*, ...] — Place string(s) in comment section of the object file

Synonym for **.ident**.

.warn ["*string*"] — Print a warning message

Causes the assembler to print a warning message to `stderr` (or to `stdout` if you have specified option **-Eo**). Also increments the warning count.

You can optionally specify a string to include in the warning message. The string must be enclosed in double quotes.

.weak *name*[, *name*, ...]— Specify weak ELF binding

Sets the binding of *name* to *weak*.

.while *expression* — Assemble a repeat block while expression evaluates to true

Preprocessor directive

Tells the assembler to duplicate an instruction sequence while *expression* evaluates to true. Directive **.endw** terminates the instruction sequence.

.word *expression*[, *expression*, ...] — Store initialized 32-bit value(s) (full words) in current section

Synonym for **.4byte** and **.long**.

Using extension directives

You can extend the assembler by adding new instructions, registers, and condition codes. This section describes how to extend the assembler. Extension directive reference on page 68 describes the directives provided by the assembler to allow use of these extensions in your assembler source code.

Consult the *Programmer's Reference* and *Extensions Interface Reference* for your specific processor for details about extending the processor functionality.

> **NOTE** Extensions should be defined and used only when hardware (or a simulator) supporting those extensions is present.

We recommend that you place in a separate include file any extensions you define. The file *install_dir*`/bin/arcexlib.s` is the assembler definition of the Extensions Library and DSP extensions, and is used when compiling and assembling code with the **-X** family of **mcc** driver options. (For information about these options, see Command-line option reference on page 12.)

Once you have defined your extension directives, you can use them in three ways:

- Add the extension definition file to your assembler command line before any files that need the extensions. For example, the following command assembles both files, concatenated together:

ARCtangent-A4

```
asarc myexts.s mysource.s
```

ARCtangent-A5 and later

```
asac  myexts.s mysource.s
```

The assembler processes files in the order they are specified on the command line.

- Include the extension definition file by adding a **.include** directive to your source file (`mysource.s`) before any of the extension directives are used. For example, the following command defines the extension directives in-line where the **.include** directive is specified:

  ```
  .include "myexts.s"
  ```

- Use **mcc** driver option **-Hextfile** to specify the extension definition file on the command line:

  ```
  mcc -c -Hextfile=myexts.s mysource.s
  ```

To use the standard extensions defined in `arcexlib.s`, we recommend you use the **-X mcc** driver options for specifying the extensions; for example:

```
mcc -c -Xxmac_16 -Xswap mysource.s
```

You can use option **-v** to see the exact command passed to the assembler.

For more information about the standard extensions and related driver options, see the *MetaWare C/C++ Programmer's Guide*.

NOTE When you use the extension directives, the debugger disassembly and register displays recognize the extensions when you debug your executable.

Defining extensions

NOTE The sample include file, `arcexlib.s`, defines the extensions used in the *Extensions Library* document as well as DSP extensions. The extensions in `arcexlib.s` are conditionally defined through several preprocessor definitions. See the file for details.

To define extensions, you use the following directives:

- **.extAuxRegister**
- **.extCondCode**
- **.extCoreRegister**
- **.extInstruction**

These directives add extensions to the auxiliary registers, condition codes, core registers, and instructions that are available for writing an assembly program.

When the assembler analyzes instructions, registers, and condition codes, it searches user-defined extensions before it searches the base definitions provided by the assembler. This allows extension instructions, registers, and condition codes to override the base definitions, and makes the order of definitions in your program important. Extensions that are defined later can override or overload previously defined extensions and core elements.

Extension directive reference

This section lists extension directives alphabetically.

.extAuxRegister *name*, *loc*, *mode* — Define an extension auxiliary register

Defines a new auxiliary register, *name*, in your program. *name* is case insensitive. Register names are limited to 16 characters in length.

If *name* already exists, either as a base register or a previously defined extension register, the existing definition is overridden, and the new definition is used from the point of definition forward.

loc is the memory location mapped to the auxiliary register.

mode is a flag that indicates the read/write status of the register. The value of *mode* is one of the following:

`r`	read-only
`w`	write-only
`r|w`	read/write

For example, the following statement defines an auxiliary register `MULHI`, with memory location 0x12, as a write-only register:

Example

```
.extAuxRegister MULHI, 0x12, w
```

.extCondCode *suffix*, *value* — Define an extension condition code

Defines a new condition code value, *suffix*, to be used as a suffix on instructions in source code.

CAUTION Do not redefine suffixes that are not condition code suffixes, such as `a` and `di`, because this disables their previous meaning.

suffix is not case sensitive, and can be any length.

If *suffix* already exists, either in the base-case definition or as a previously defined extension condition code, that existing definition is overridden, and the new definition is used from the point of definition.

value is the condition code value for the suffix, in the range 0 (zero) to 31.

NOTE Instructions such as **BLO** are actually **B***cond_code*, meaning that any new condition codes you define can be used as part of them. For example, if you define a new condition code named PLAH, **BPLAH** becomes a valid instruction name.

Example

```
.extCondCode isOnFire, 22
```

You can then use the suffix `isOnFire` as follows:

```
add.isOnFire r1,r2,r3
BisOnFire target
```

.extCoreRegister *name*, *rnum*, *mode*, *shortcut* — Define an extension core register

Defines a new core register to be available in your program. You can also use directive **.extCoreRegister** to add new names, or aliases, for the base case registers.

name is a mnemonic name for the register, such as `mlo`. *name* is case insensitive. Register names are limited to 16 characters in length.

If *name* already exists, either as a base register or a previously defined extension register, that existing definition is overridden, and the new definition is used from the point of definition forward.

num is the register number, in the range 0 (zero) to 60.

mode is a flag that indicates the read/write status of the register. The value of *mode* is one of the following:

r	read-only
w	write-only
r|w	read/write

shortcut is a flag that specifies whether the register can be "shortcutted" in the processor's pipeline. A register that is used as a destination in one instruction and a source in the next instruction must be shortcut-enabled. If it is not, the assembler issues a warning.

The value of *shortcut* is one of the following:

can_shortcut	Shortcutting is allowed.
cannot_shortcut	Shortcutting is not allowed.

See the *Programmer's Reference* for information about how registers can be shortcutted through the pipeline.

Example

```
.extCoreRegister mlo, 57, r|w, can_shortcut
```

.extInstruction *name*, *opcode*, *sub_opcode*, *suffix_classes*, *syntax_class* — Define an extension instruction

Adds a new instruction to the set of valid instructions. This new instruction is not a macro; rather, it defines a new instruction and rules for encoding that instruction.

> **NOTE** Later-defined instructions take precedence over previously defined instructions (including base instructions). Which instruction is actually used depends on two things: the name of the instruction, and the syntax of the instruction. For more information about how the assembler selects instructions, see Examples on page 72.

name is the name of the new instruction. *name* is case insensitive.

Argument *opcode* is the major opcode for encoding the instruction and occupies bits 27:31 of a 32-bit instruction (bits 11:15 of a 16-bit ARCompact instruction). For ARCtangent-A4, *opcode* must be in the range 0x10 to 0x1F, or 0x03. For the ARCompact ISA (ARCtangent-A5 and later), *opcode* must be in the range 0x05 to 0x07.

Argument *sub_opcode* is the minor opcode for encoding the instruction. For the ARCtangent-A4 ISA, *sub_opcode* must be in the range 0x09 through 0x3F. For the ARCompact ISA, how the assembler uses *sub_opcode* for encoding the instruction depends on the syntax class. When the syntax class is `SYNTAX_3OP`, the sub-opcode may be anywhere in the range 0x0 to 0x3F except 0x2F. When the syntax

class is SYNTAX_2OP, the sub-opcode must be in the range 0x0 to 0x3E. When the syntax class is SYNTAX_NOP or SYNTAX_1OP, the sub-opcode must be in the range 0x0 to 0x3F.

NOTE For ARCtangent-A4, if *opcode* is not 0x03, *sub_opcode* must be 0 (zero).

Argument *suffix_classes* is a bit vector that tells the assembler which types of suffixes to allow with the instruction. To specify a value for *suffix_classes*, bitwise **OR** the values that apply from the following list:

SUFFIX_NONE	No suffixes are allowed on this instruction.
SUFFIX_COND	A condition-code suffix is allowed on this instruction.
SUFFIX_FLAG	A flag suffix is allowed on this instruction.
SUFFIX_COND | *SUFFIX_FLAG*	Both condition-code and flag suffixes are allowed on this instruction.

Argument *syntax_class* is a set of possible operand formats for a given instruction. Each instruction you define must conform to one of the following syntax classes:

SYNTAX_NOP	ARCtangent-A5 and later: the instruction contains no operands.
SYNTAX_1OP	ARCtangent-A5 and later: the instruction contains one source operand.
SYNTAX_2OP	The instruction contains two operands: one source operand and one destination operand.
SYNTAX_3OP	The instruction contains three operands: two source operands and one destination operand.

Any operand can be either a register or immediate value.

NOTE For ARCtangent-A4, if opcode is not 0x03, and *syntax_class* is SYNTAX_2OP, the instruction is encoded as a three operand instruction; the second operand is repeated.

Syntax-class modifiers

- OP1_DEST_IGNORED (ARCtangent-A5 and later)

  Use OP1_DEST_IGNORED with OP1_MUST_BE_IMM and OP1_IMM_IMPLIED when the instruction ignores the destination operand. This allows the assembler to encode the instruction using a register destination field even if the destination is syntactically specified with an immediate value, if the assembler determines that it is advantageous to do so.

- OP1_IMM_IMPLIED

  OP1_IMM_IMPLIED modifies syntax class SYNTAX_2OP; it specifies that there is an implied immediate destination operand which does not appear in the syntax. For example, if the source code contains an instruction like this:

  ```
  inst r1,r2
  ```

it really means that the first argument is an “implied immediate” (that is, the result is discarded).

This is the same as though the source code were:

```
inst 0,r1,r2
```

You use `OP1_IMM_IMPLIED` by bitwise **OR**ing it with `SYNTAX_2OP`.

- `OP1_MUST_BE_IMM`

  `OP1_MUST_BE_IMM` modifies syntax class `SYNTAX_3OP`; it specifies that the first operand of a three-operand instruction must be an immediate (that is, the result must be discarded).

  You use `OP1_MUST_BE_IMM` by bitwise **OR**ing it with `SYNTAX_3OP`.

Examples

This section provides examples of use of the directive **.extInstruction**.

Example 1 Overloading of a base instruction

ARCtangent-A4

```
.extInstruction asl,0x10,0x00,
SUFFIX_COND|SUFFIX_FLAG,SYNTAX_3OP
```

ARCtangent-A5 and later

```
.extInstruction asl,0x05,0x00,
SUFFIX_COND|SUFFIX_FLAG,SYNTAX_3OP
```

This example overloads base instruction **ASL**. For comparison, if the base instruction were written as an extension instruction, it would look like this:

ARCtangent-A4

```
.extInstruction asl,0x08,0x00,
SUFFIX_COND|SUFFIX_FLAG,SYNTAX_2OP
```

ARCtangent-A5 and later

```
.extInstruction asl,0x04,0x00,
SUFFIX_COND|SUFFIX_FLAG,SYNTAX_2OP
```

How the assembler determines which version of the **ASL** instruction to encode depends on the actual syntax of the instruction used. The assembler checks all defined instructions with the same name (from last-defined to first-defined) to find the first instruction that accepts the syntax used in the source code being assembled.

In this example, because the `SYNTAX_3OP` version (the extension version) is defined after the base instruction, it takes precedence during the search and is chosen if its syntax matches the syntax of the source instruction being assembled.

Suppose the source code being assembled contains the following statement:

```
asl r1,r2,r3
```

The assembler first checks the extension instruction. The register-register-register syntax falls under syntax class `SYNTAX_3OP`, so the assembler recognizes the extension instruction as the correct instruction to use.

If the source code had contained the following statement instead, the search would have continued past the extension version of the **ASL** instruction:

```
asl r1,r2
```

This is because register-register is not a member of the SYNTAX_3OP syntax class. The base-case **ASL** instruction would then have been checked and accepted, because register-register is a member of the SYNTAX_2OP syntax class.

NOTE Both forms can be used in the same source program:

```
asl r1,r2
asl r1,r2,r3
```

Example 2 Implementing a multiplier extension

OP1_IMM_IMPLIED and OP1_MUST_BE_IMM are provided to allow implementation of extensions such as the **MULU64** and **MUL64** instructions. The **MUL64** instruction has the following formats:

```
mul64<.cc><imm,>b,c
mul64<.cc><imm,>b,imm
mul64<.cc><imm,>imm,c
```

The <imm,> means that if there are three operands, the first must be an immediate. If there are two operands, the instruction is treated like a three-operand instruction with an implied immediate value as the first operand.

The **MUL64** instruction is defined as follows using extension directives:

- Three-operand format

  ```
  .extInstruction mul64,0x14,0x00,SUFFIX_COND,
  SYNTAX_3OP|OP1_MUST_BE_IMM
  ```

  This definition specifies that if the assembler sees a **mul64** instruction with three operands, it verifies that the first operand is an immediate, and generates an error message if it is not.

- Two-operand format

  ```
  .extInstruction mul64,0x14,0x00,SUFFIX_COND,
  SYNTAX_2OP|OP1_IMM_IMPLIED
  ```

This definition specifies that if the assembler sees a **mul64** instruction with two operands, it changes it into a three-operand **mul64** instruction, and then processes it as though it were a three-operand **mul64** instruction, adding a new first operand of 0 (zero), which means "discard the result."

ARCtangent-A5 and later

Extension instructions may also use the OP1_DEST_IGNORED syntax-class modifier for increased efficiency. Given an instruction written as follows:

```
mul64   0, r1, 1000
```

the assembler can safely encode it as

```
mul64   r1, r1, 1000
```

because the assembler knows the instruction ignores the destination. The second version is a four-byte instruction, while the first version is an eight-byte instruction.

Using CFA directives (ARCtangent-A5 and later)

The canonical frame address (CFA) mechanism is a subset of the DWARF 2 standard. CFA information allows debuggers to trace the call stack. The MetaWare C/C++ compiler for ARCtangent-A5 emits CFA information, and the Run-Time Libraries contain CFA directives. Writing your own assembly code does not require using CFA directives, but you may do so; the directives **.cfa_bf** and **.cfa_ef** are usually sufficient to enable call-stack tracing for a hand-written assembly function.

Canonical frame address mechanism

The canonical frame address (CFA) is like an anchor in the stack frame that never moves during execution of a function. The CFA is computed by adding the value of the CFA register (`CFA_REG`) to the CFA offset (`CFA_OFF`). The CFA offset of a non-volatile register is the amount that must be added to the CFA to determine where the saved register is stored.

CFA directives provide a means of setting and manipulating the `CFA_REG` and `CFA_OFF`.

CFA directive reference

As of this writing, the assembler supports the CFA directives listed in this section. For an example of usage of CFA directives, see Examples on page 76.

.cfa_bf *func* [,*linkreg*] — Begin function

Directive **.cfa_bf** indicates the beginning of a function and must be the first CFA directive in every function. Argument *func* is the name of the function; the optional *linkreg* indicates an alternate return register (that is, if the function returns to a register other than `blink`). The corresponding directive **.cfa_ef** must be the last CFA directive in every function.

.cfa_def_cfa *reg*, *off* — Define location of CFA

Directive **.cfa_def_cfa** defines the location of the CFA as the value contained in register *reg* plus the offset *off*. By default, directive **.cfa_bf** defines the location of the CFA as the `sp` register with an offset of 0 (zero):

```
.cfa_def_cfa   sp, 0
```

.cfa_def_cfa_register *reg* — Define CFA register

Directive **.cfa_def_cfa_register** defines *reg* as the register (`CFA_REG`) to be used for the CFA. It does not affect the offset portion of the CFA definition.

.cfa_def_cfa_offset *off* — Define CFA offset

Directive **.cfa_def_cfa_offset** defines *off* as the offset (`CFA_OFF`) for calculating the location of the CFA. It does not affect the register portion of the CFA definition.

.cfa_ef — End function

Directive **.cfa_ef** indicates the end of a function and must be placed at the end of every function in which **.cfa_bf** was emitted.

.cfa_info ULEB128 — Emit toolset-specific CFA instruction

Directive **.cfa_info** causes the assembler to emit a toolset-specific CFA instruction with the single argument **ULEB128** to provide information to the MetaWare debugger.

> **NOTE** This listing is provided for information only; you should not need to use directive **.cfa_info**. It is used in the MetaWare C Run-Time Libraries in rare cases where unusual hand-coded assembly requires telling the debugger additional information about what it can assume.

.cfa_offset {*reg* | {*reglist*}}, *offset* — Set register offset rule

When directive **.cfa_offset** is used with a single register *reg* as its first argument, it sets that register's offset rule to use the offset *offset*. That is, the previous value of the register can be found in memory at the given offset from the CFA (which itself is determined from a CFA register and CFA offset). This permits debuggers to find the previous value of *reg* at `mem[CFA_REG + CFA_OFF +` *offset*`]`.

When directive **.cfa_offset** is used with multiple registers *reglist* as its first argument, it sets the first register's offset rule to use the offset *offset*, the second register's offset rule to use the offset *offset*+4, and so on.

.cfa_pop {*reg*|{*reglist*}|*n*} — Adjust offset of stack-pointer CFA for stack pops

> **NOTE** This directive assumes that the stack-pointer register is the register portion of the CFA definition.

Directive **.cfa_pop** adjusts the offset portion of the CFA definition when the stack pointer is incremented.

When incrementing the stack pointer simply to restore the stack pointer, use argument *n* to indicate the number of bytes added.

When incrementing the stack pointer for popping a single register, use the value of the register as argument *reg*.

When incrementing the stack pointer as part of a sequence of register-pop operations when restoring non-volatile registers, use argument *reglist*. Using argument *reglist* establishes a restore rule for each register in the list.

.cfa_push {*reg*|{*reglist*}|*n*} — Adjust offset of stack-pointer CFA for stack pushes

> **NOTE** This directive assumes that the stack-pointer register (`sp`) is the register portion of the CFA definition.

Directive **.cfa_push** adjusts the offset portion of the CFA definition when the stack pointer is decremented. In other words, it updates the value of `CFA_OFF` so that adding it to `CFA_REG` still yields the original CFA location.

When decrementing the stack pointer simply to allocate space, use the argument *n* to indicate the number of bytes subtracted.

When decrementing the stack pointer for pushing a single register, use the value of the register as argument *reg*.

When decrementing the stack pointer as part of a sequence of register-push operations when saving non-volatile registers, use argument *reglist*. Using argument *reglist* defines the rule for each register as an offset rule, assuming the registers are pushed in the order that they are listed in *reglist*. The offsets assigned are typically negative values, and the first offset assigned is typically the (negative of the) value of the CFA offset prior to the first push, less 4.

.cfa_register *reg1*, *reg2* — Set register rule

Directive **.cfa_register** makes the rule for *reg1* a register rule and indicates that the previous value of *reg1* is now *reg2*.

.cfa_reg_offset *reg* | {*reglist*}, *offset* — Set register offset rule

This directive is similar to **.cfa_offset** in that it sets offset rules for the register operands. However, the *offset* specified is not the absolute offset of the register location from the CFA. Instead `CFA_REG` is treated as a base pointer, and the offset is the offset from the `CFA_REG` base pointer where the register is stored.

Consider this example, assuming `CFA_OFF` is 32:

```
.cfa_reg_offset r13, 4
```

The result is that register `r13` is stored 4 bytes beyond the `CFA_REG` address.

To reference the `r13` storage location from the CFA, the offset is -28, which is the offset rule this directive generates in this case.

.cfa_remember_state — Save rules for all registers

Directive **.cfa_remember_state** causes the assembler to emit a “remember state” CFA instruction, effectively saving the rules for all registers on an implicit stack.

.cfa_restore *reg* | {*reglist*} — Restore registers to function-entry

Directive **.cfa_restore** indicates that the saved register *reg* (or the list of saved registers *reglist*) has been restored to the register state when the function started executing.

.cfa_restore_state — Restore saved register rules

Directive **.cfa_restore_state** causes the assembler to emit a “restore state” CFA instruction, effectively popping the rules for all registers from the implicit stack (if they were saved there by directive **.cfa_remember_state**).

Examples

If the contents of register `CFA_REG` change frequently (as when the stack-pointer register is used as the `CFA_REG`), `CFA_OFF` must be updated frequently to compute the original CFA.

Supposing the stack-pointer register (sp) is the CFA_REG, the stack pointer and CFA are as shown in Figure 1 CFA on function entry when the application enters a function.

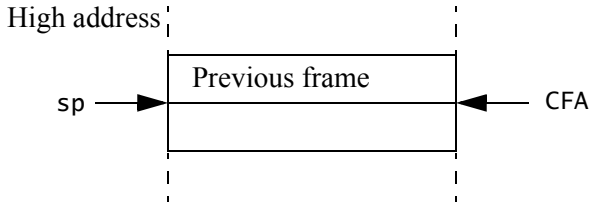

Figure 1 CFA on function entry

At this point, sp and the CFA are the same, so CFA_OFF is zero.

Figure 2 CFA After three register pushes shows the same stack after three registers are pushed onto it.

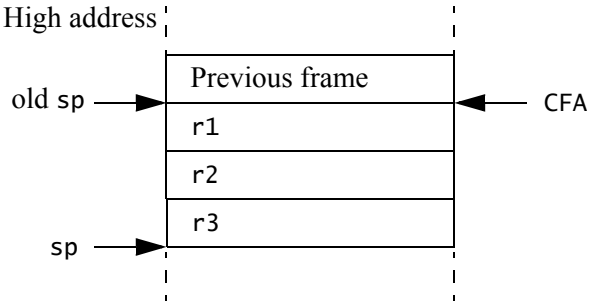

Figure 2 CFA After three register pushes

Because the CFA_REG (sp) has changed, the correct CFA_OFF is no longer zero. Because the CFA is now at sp+12, the compiler emits a directive to indicate that CFA_OFF is now 12, to maintain the requirement that the CFA be the sum of CFA_REG + CFA_OFF.

Either of the following directives indicates that CFA_OFF is now 12:

```
.cfa_def_cfa_offset 12
.cfa_push 12
```

Using **.cfa_def_cfa_offset** assigns CFA_OFF an absolute value (12 in this case).

Using **.cfa_push** adds the parameter (12 in the case) to the current value of CFA_OFF.

The locations of registers r1, r2 and r3 relative to the CFA results in the following offsets:

```
r1   -4

r2   -8

r3   -12
```

The following **.cfa_offset** directives set the individual register locations:

```
.cfa_offset  r1, -4
.cfa_offset  r2, -8
.cfa_offset  r3, -12
```

The following syntax achieves the same result:

```
.cfa_offset  {r3, r2, r1}, -12
```

The **.cfa_push** directive updates CFA_OFF directly and assigns the correct CFA offset to each register:

```
.cfa_push    {r1, r2, r3}
```

Appendix A — Quick Options List

This section presents a hyperlinked list of assembler command-line options for quick reference. For detailed listings, see Command-line options on page 11.

-%reg	Specify that register names must be preceded by a percent sign
-a4	Specify the ARCtangent-A4 processor series
-a5	Specify the ARCtangent-A5 processor series
-ac32	Use 32-bit encodings (ARCtangent-A5 only)
-arc600 | **-a6**	Specify the ARC 600 processor series
-arc700 | **-a7**	Specify the ARC 700 processor series
-be	Assemble using big-endian format
-c	Suppress display of the copyright message
-core*	Specify core version
-D*name*[=*n*]	Define an identifier and assign a value to it
-Eo	Send error messages to standard output
-errors *n*	Set the assembler error limit
-extregs	Define extension core registers
-f*flag*[*flag*…]	Set listing flags
-g	Generate debugging information for assembly source files
-h	Display command-line option help screen
-I*include_dir*[,*include_dir*…]	Specify directory to be searched for **.include** files
-l	Generate an assembly output listing
-L	Place private labels in the output symbol table
-le	Assemble using little-endian format
-makeof=*filename*	Generate makefile-dependency file
-make_rel	Exclude absolute path names from metafile-dependency info
-mathu	Perform expression evaluations using unsigned math.
-noesc	Disable backslash (C-style) escape character
-o *object_file*	Override the default object-file name

-offwarn=*warning_no*[,*warning_no*...]	Turn off selected warnings
-on | **-off**	Turn one or more toggles on or off
-percent_reg	Specify that register names must be preceded by a percent sign
-O {*y*|*n*}	Specify whether assembler version-number information appears in the object file
-require_declarations	Do not permit undefined or undeclared symbols
-U*name*	Undefine an identifier
-v	Print summary of assembler statistics
-w	Suppress warning messages
-wn	Warn about instructions with no effect
-Xadds	Enable ADDS and SUBS instructions
-Xbarrel_shifter	Enable barrel shifter instructions
-Xdpfp	Enable double-precision floating-point extensions
-Xdsp_packa	Enable DSP Pack A extension instructions
-Xmin_max	Enable MIN and MAX instructions (ARCtangent-A4 only)
-Xmul_mac	Enable MULMAC extension instructions
-Xmult32	Enable scoreboarded multiplier instructions
-Xnorm	Enable NORM instruction
-Xscratch_ram	Enable scratch RAM extension
-Xswap	Enable SWAP instruction
-Xvbfdw	Enable support for dual Viterbi instruction
-Xxmac_16	Enable 16-by-16 multiply accumulate extension
-Xxmac_d16	Enable dual 16-by-16 multiply accumulate extension
-Xxmac_24	Enable 24-by-24 multiply accumulate extension
-Xxy	Enable XY memory extension