CompactRISC™

Assembler
Reference Manual

Part Number: 424521772-003

August 1998

REVISION RECORD

VERSION	RELEASE DATE	SUMMARY OF CHANGES
0.6	August 1995	First beta release.
0.7	January 1997	Minor changes and corrections.
1.0	August 1996	CR16A Product Version. CR32A Beta Version.
1.1	February 1997	Minor modifications and corrections.
2.a	September 1997	Alpha release for CR16B.
2.0	January 1998	Beta release.
2.1	August 1998	Product release.

PREFACE

The CompactRISC Development Toolset supports development of software for National Semiconductor's CompactRISC microprocessor family. This manual describes the CompactRISC Assembler which is a part of the CompactRISC Toolset.

The assembler program takes a CompactRISC assembly language program and creates a relocatable object file which is then used as an input to the CompactRISC Linker.

This manual describes both the CompactRISC Assembler and the CompactRISC assembly language.

The information contained in this manual is for reference only and is subject to change without notice.

CONTENTS

FIGURES

TABLES

Chapter 1
OVERVIEW

1.1 INTRODUCTION

CompactRISC Assembler is a software development tool that assembles CompactRISC Assembly Language source programs and generates relocatable object modules. Relocatable object modules may be linked to create executable load modules which may be run on CompactRISC family microprocessor-based systems that support the Common Object File Format (COFF) as implemented by National Semiconductor. The CompactRISC language tools provide linkage and library maintenance programs.

This manual describes the CompactRISC Assembler in detail. It is organized as follows:

Chapter 1 *Overview* (this chapter), introduces the CompactRISC Assembler, summarizes its features, and describes the registers.

Chapter 2 *Invoking the Assembler*, describes the CompactRISC Assembler, assembly options, output files, and error messages.

Chapter 3 *Elements of the CompactRISC Assembly Language*, describes the format of the CompactRISC Assembly Language statements, constants, values, symbols, and expressions.

Chapter 4 *CompactRISC Assembler Programs*, describes program segments, linkage, and relocation.

Chapter 5 *Instruction Operands*, describes the syntax of the CompactRISC Assembly Language instruction operands.

Chapter 6 *CompactRISC Assembler Directives*, defines the syntax and function of the CompactRISC Assembler directives.

Chapter 7 *Macro and Conditional Assembly*, describes the new macro-assembler.

Appendix A *Directive Summary*, summarizes the CompactRISC Assembler directive syntax and function.

Appendix B *Reserved Symbols*, lists the CompactRISC Assembler reserved symbols.

Appendix C *Glossary*, provides a glossary of CompactRISC terms.

1.2 OVERVIEW OF ASSEMBLER FEATURES

The CompactRISC Assembler provides a number of features for efficient assembly language programming.

Input and Output Files. The CompactRISC Assembler generates an object code file, an optional listing file, an optional cross-reference listing, and an optional symbol table dump from an assembler source file. The object code file consists of assembled statements suitable for execution after the appropriate linking process. The listing file consists of the source file statements, and the assembled code, if the source file assembles successfully; otherwise, the listing file consists of error messages and source file statements that caused the error. Input and output files, listing file format, cross-reference listing, symbol table dump, and error messages are described in Chapter 2.

Architecture Support. The CompactRISC Assembler supports the complete instruction set, including the integer, boolean, string, array, processor control, and processor service instructions.

The CompactRISC Assembler supports all the addressing modes supported by the CompactRISC architecture.

Data Types. The CompactRISC Assembler recognizes a variety of operand data types including integers (byte, word, double-word), and single- and double-precision floating-point numbers. The CompactRISC Assembler supports all the data types supported by the CompactRISC architecture.

Assembler Directives. The CompactRISC Assembler provides directives to create symbolic labels, generate data, allocate storage, control program listings, control linkage, control line number table, control program segments, define module table entry, define symbol table entry, define macros, and define file name.

Macro-preprocessor. The CompactRISC Assembler has a built-in macro preprocessor. Macro processing is performed as the first pass of the assembly process. The powerful macro preprocessor simplifies assembly programming.

1.3 DEFINITION OF TERMS

The following terms are used throughout this document:

- Software Module
  A software module is a portion of a program that may be separately compiled or assembled and linked together with other software modules into an executable program image.

- Relative Value
  A relative value is a symbol or expression that specifies an address within one of the Common Object File Format (COFF) sections or the corresponding assembly program segment. Because such addresses are not bound to actual memory locations until link time, their values are relative to the base or starting address of the module. Relative values are called relocatable addresses.

- Absolute Value
  An absolute value is a symbol or expression that specifies a numeric address. An absolute value or absolute address is unaffected by linkage.

Chapter 2
INVOKING THE ASSEMBLER

2.1 INTRODUCTION

The CompactRISC Assembler generates object code from CompactRISC assembly language source files and optionally produces a listing file, a symbol table file, a cross-reference file, and debugging information. Each assembly source file produces one object file, that may consist of a text (code) section and an initialized data section. This object file can be later linked with other object files using the CompactRISC Linker to generate an executable object file.

This chapter describes the input and output files used by the CompactRISC Assembler, assembler invocation, listing file, symbol table listing, the cross-reference table, assembly errors, and the CompactRISC Assembler limitations.

2.2 INPUT AND OUTPUT FILES USED/GENERATED BY THE ASSEMBLER

The files used as input and those generated as output by the assembler are shown in Figure and described below.

Source file Input. The source file is a text file containing the source program to be assembled.

Object file Output. The object file contains the relocatable object code and data produced by the assembler, as well as optional debugging information. When no filename for the object file is given, the default name is the name of the source file with the **`.s`** suffix, if any, stripped off and a **`.o`** suffix appended. For example, if the source file is named **`build.s`**, the name of the object file is **`build.o`**. The object file is suitable for use as input to the linker or librarian.

Listing file Output. The listing file, created with the **`-L`** option, contains the program listing produced by the assembler. The default listing file is the standard output (**`stdout`**). If a filename is specified with the listing option, this file is used as the listing output.

Symbol table file Output. The symbol table file, created with the **-y** option, contains a dump of the symbol table. For each symbol it gives its name, value, section and if it is external. The default symbol table file is the standard output (**stdout**). If a filename is specified with the symbol table option, this file is used as the listing output.

Cross-reference file Output. The cross-reference file, created with the **-x** option, contains, for each symbol, the lines on which it is used together with the line on which it is defined. The default cross-reference file is the standard output (**stdout**). If a filename is specified with the cross-reference option, this file is used as the listing output.

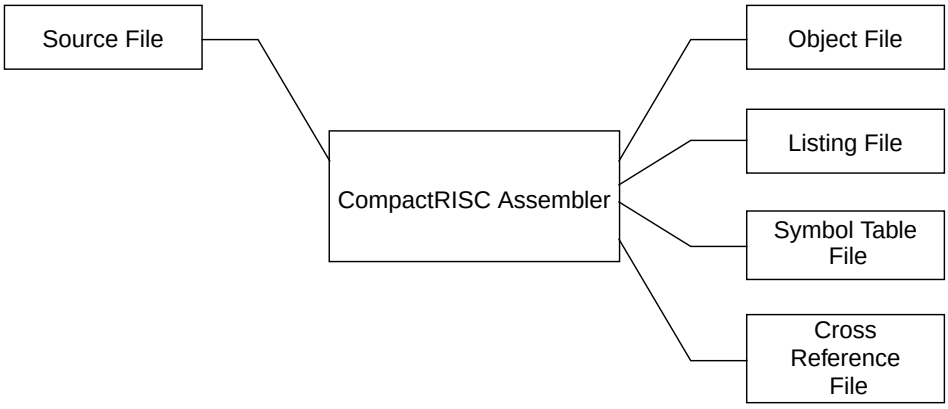

Figure 2-1. Input and Output Files for the CompactRISC Assembler

The CompactRISC Assembler creates a number of temporary files during the assembly process. These files are created in a directory specified by the value of the **TMPDIR** environment variable. If **TMPDIR** is not set, the current directory is used for temporary files.

After the assembly process, the CompactRISC Assembler deletes these temporary files.

2.3 ASSEMBLER INVOCATION

You invoke the CompactRISC Assembler from a command prompt by entering the **crasm** command, followed, if required, by options, and a source filename. In addition it is possible to specify all, or part, of the arguments from an argument file. An arguments file is denoted by an at-sign (@) followed by a file name. The assembler command line syntax is:

crasm [{***options*** | ***sourcefile*** | ***@argfile***] **...**

The source filename and the options may appear in any order with the exception of the **-c** option which must come before the **-D, -U,** or **-I** options. Only one source filename is permitted. See Table 2-1 for a list of the options, their syntax, and their definitions.

Examples

```
1. crasm
2. crasm myfile.s
3. crasm -L myfile.s
4. crasm -L -o myfiledebug.o myfile.s
5. crasm -L myfile.s > myfile.lis
6. crasm -Lmyfile.lis myfile.s
```

Example 1 does not specify any filename or option. The assembler waits for input from **stdin** and creates an object file **.o**

Example 2 assembles the source file **myfile.s** and generates an object file with the default name **myfile.o.** No listing file is produced.

Example 3 generates both an object file, **myfile.o,** and a listing file from the source file **myfile.s.** The listing file is output to **stdout**.

Example 4 generates the object file **myfiledebug.o** from the source file **myfile.s**. Because the **-L** option is specified, a listing is produced on **stdout**.

Examples 5 and 6 generate a listing file from the source file **myfile.s** and output it to **myfile.lis**. The object file generated is **myfile.o**.

Table 2-1. Options Syntax

Option	Definition
-ds | -dm | -dl	Sets the default displacement size to small, medium, or large. The default is large (l). If there are only two possible sizes in the architecture, medium is the same as large.
-c	Runs the C compiler pre-processor **(cpp)** on the input to the assembler.
-r	Incorporates the data segment into the text segment. Off by default.
-s	Saves compiler-generated labels in the symbol table of the object file.
-V	Writes the version number of the assembler to stderr.
-v	Uses memory for intermediate storage rather than a temporary disk file.
-L[***filename***]	If ***filename*** is given, produces the listing in that file. If not, the listing is sent to the standard output.

Table 2-1. Options Syntax (Continued)

Option	Definition
-n	Disables displacement size optimization.
-o *objfile*	Names the output object file as ***objfile***. By default, the output filename is formed by removing the **.s** suffix, if it is present, from the input filename and adding a **.o** suffix.
-mlarge	(CR16B only) Generate code for CR16B large programming model.
-msmall	(CR16B only) Generate code for CR16B small programming model (default).
-mcr16a	Generate code for CR16A.
-t	Causes the assembler to show all the utilities it calls. This option is useful for tracing all processes executed by the assembler.
@*filename*	Reads arguments from file ***filename***.
-w	Suppresses assembly warning messages.
-y[***filename***]	Produces a symbol table listing in ***filename***. If ***filename*** is omitted, it is sent to the standard output.
-x[***filename***]	Produces a cross-reference file in ***filename***. If ***filename*** is omitted, it is sent to the standard output.
-D*name* or -D*name=def*	Defines ***name*** to **cpp**, as if by "**#define**". If no definition is given, ***name*** is defined as 1. The **-c** option must precede this option.
-U*name*	Removes initial definition of a cpp predefined symbol ***name***.
-I*dir*	First searches "***#include***" files, that do not begin with **/**, in the ***filename*** directory, then the directory named in this option, then the directories on a standard list. The **-c** option must precede this option.
-g	Produces additional line number information for symbolic debugging.
-MO	Invokes only macro-processing phase.
-MP[*filename*]	Prints the macro processor output.
-ML*filename*	Includes macro library file.
-MI*dir*	Specifies an include search directory for the macro processor.
-MD*name* -MD*name=def*	Defines a name to the macro processor, as if by macro assignment statement.

Table 2-1. Options Syntax (Continued)

Option	Definition
-z	Dumps errors and warnings into ***filename*.err** file, where ***filename*** is the base name of the input file. For example: **crasm -z test.s** generates the error file **test.err**.
-zn*filename*	Dumps errors and warnings into ***filename*** file. For example: **crasm -zntest.x new_test.s** generates the error file **test.x**. Note, there must be no space between **zn** and ***filename***.

2.3.1 Assembler Symbolic Debugging

When you invoke the assembler with the **-g** option, it generates a line number entry in the object file for every source line of the input assembly file where a breakpoint can be inserted. The information from the line number entries allows you to reference the line numbers when using a software debugger.

Code segments are grouped by the assembler to form dummy procedures. Dummy procedures start at the first statement and end at the last statement of the assembly source file. The name of the dummy procedure is of the form ***.Xbasename_number***, where ***basename*** is the source file name without the **.s** suffix; and ***number*** is the file segment number.

Every assembler label with a storage allocation directive (e.g., **.double**, **.blkd**) is given a type based on the storage allocation. Types are assigned as follows:

Storage Allocation Directives	Corresponding Type
.byte, **.blkb**	***unsigned char***
.word, **.blkw**	***short int***
.double, **.blkd**	***int***
.float, **.blkf**	***float***
.long, **.blkl**	***double***
.ascii	***char***

When the **.ascii** directive is used or when a repetitive factor is specified for any other storage allocation directive, the associated label is considered an array of the corresponding type.

If the input source file contains **.ln** directives (see Section 6.10.1), the assembler assumes that symbolics were generated by the compiler and no symbolic debugging information is prepared; instead, information from the **.ln** directive is used to generate the line number entry.

2.4 ASSEMBLER OUTPUT LISTINGS

Figure 2-2 shows a sample assembly language program. The annotated listing produced when the program is assembled is shown in Figure 2-3.

Example

```
#PIKE encoding
.globl _j
.data
_j:
          .word         1
.text
.globl    _max
_max:
          loadw         _i,r1
          loadw         _j,r0
          cmpw          r1,r0
          ble           .L2
          movw          r1,r0
.L2:
          cmpw          r2,r0
          ble           .L4
          storw         r2,_i
          jump          ra
.L4:
          storw         r0,_i
.L5:
          jump          ra
.globl _i
.bss  _i,2,2
```

Figure 2-2. Sample Assembly Program

Example

```
                                                  (1)       (2)
CompactRISC Assembler (CR16A) Version 0.6 (rev 3) 6/21/95  Page:1

          (3)
##### File "ex1.s" #####

(4)     (5)  (6)             (7)

1                .globl                       _j
2                .data3
     D00000000   _j:
4    D00000000 0100                    .word    1
5                .text
6                .globl       _max
7    T00000000   _max:
8    T00000000f9a32400loadw  _i,r1
9    T00000004f9a12000loadw  _j,r0
10   T00000008e360    cmpw   r1,r0
11   T0000000a044e    ble    .L2
12   T0000000c8361    movw   r1,r0
13   T0000000e    .L2:
14   T0000000ee560    cmpw   r2,r0
15   T00000010084e    ble    .L4
16   T00000012 f9e52400storw  r2,_i
17   T000000165d5c    jump   ra
18   T00000018    .L4:
19   T00000018f9e12400storw  r0,_i
20   T0000001c    .L5:
21   T0000001c5d5c    jump   ra
22               .globl _i
23               .bss_i,2,2
```

Callouts 1 - 7:

1 Version number of a tool.

2 Listed file page number.

3 Source file name. Reflects included files.

4 Source file line number.

5 Address of the current line. Preceded by the letter representing the section of address.

6 Code or value of source line.

7 User source line itself.

Figure 2-3. CompactRISC Assembler Listing File (Annotated)

A sample program with one error is shown in Figure 2-4. When the program is assembled, the error is flagged as shown in Figure 2-5. Assembly errors are discussed in Section 2.5.

Example

```
#PIKE
_main::
addw $-4,
storw r4,0(sp)
addw $-4,sp
```

Figure 2-4. A Sample Program Containing Errors

Example

```
CompactRISC Assembler (CR16A) Version 0.6 (revision 3)6/21/95  Page:
1

##### File "ex2.s" #####

1               _main::
2               addw $-4,

"ex2.s", line 2: Operand 2: general-purpose register expected

3               storw r4,0(sp)
4               addw $-4,sp
5

ERRORS DETECTED : 1.
```

Figure 2-5. CompactRISC Assembler Listing File With Error Message

2.4.1 Assembler Symbol Table Listing

The symbol table listing is entitled “Symbol Table Dump.” It is preceded by a formfeed, and is output to the file specified on the invocation line. If no output file is specified, the symbol table is output to **`stdout.`** Figure 2-6 shows a sample source file, and Figure 2-7 shows a sample symbol table listing.

Example

```
#SR
.set x,10
br foo
movw $f00,r0
foo:
.globl blap
movw $blap,r0
```

Figure 2-6. Sample CompactRISC Assembler Symbol Table Source File

Example

```
CompactRISC Assembler (CR32A) Version 0.6 (revision 3)

Symbol Table Dump
Symbol          Value               Section
blap            0X0                 undefined, external
f00             0X0                 undefined, external
foo             0Xa                 .text
```

Figure 2-7. Sample CompactRISC Assembler Symbol Table Listing

The symbols are listed in the order in which they are encountered. The first column of the output is the name of the symbol, the second column is the value (in hexadecimal) of the symbol, and the last column is the name of the section to which it belongs.

2.4.2 Cross-Reference Table Listing

The cross-reference listing is entitled “Cross-Reference Table”. It is preceded by a formfeed, and is output to the file specified on the invocation line. If no output file is specified, the cross reference is output to **stdout.** Figure 2-8 shows a sample source file, and Figure 2-9 shows a sample cross-reference table listing.

Example

```
            .set x, 10
            bsr foo
            movd foo, r0
foo:
            .globl blap
            movd blap, r0
```

Figure 2-8. Sample CompactRISC Assembler Cross-Reference Source File

Example

```
CompactRISC Assembler (CR16A) Version 0.6 (revision
3)6/21/95  Page: 1

Cross Reference Table
blap        5+          6
foo         2           3           4^
x           1-
```

Figure 2-9. Sample Assembler Cross-Reference Table Listing

Symbols are listed in alphabetical order. The numbers listed beside the line numbers are the source lines where the symbol appears. A **^** beside a line number indicates that the symbol is declared on that line. A **+** beside a line number indicates that the symbol is imported/exported (declared with a **.globl** directive) on that line. A **-** beside a line number indicates that the symbol is set (or reset with a **.set** directive) on that line.

2.5 ASSEMBLER ERRORS

When the assembler finds an error, it provides an error message through standard error file **(stderr**). If you selected the **-L** option, the assembler includes the error message in the listing file following the line containing the error. Most errors inhibit the assembler from generating any further object code (refer to Figure 2-5). If you selected the **-z** or **-zn*filename*** option, the assembler dumps errors into a file, in addition to the standard error file.

2.6 ASSEMBLER LIMITATIONS

This section contains a list of limitations of the CompactRISC Assembler.

Expression Expressions are calculated as 4-byte integers. High order bytes/bits are filled with zeros.

Line The length of the input line is limited to 64K characters.

Range of values The range of values for displacements is architecture dependent. See the datasheet for the relevant microprocessor.

The range of values for floating-point constants is:

single precision: 1.17549436 x 10**−38 to 3.40282346 x 10**38 and
−1.17549436 x 10**-38 to −3.40282346 x 10**38

double precision: 2.2250738585072014 x 10**−308 to
1.7976931348623157 x 10**308 and
−2.2250738585072014 x 10**-308 to
−1.7976931348623157 x 10**308

The range of values for integer constants is:

byte constants: −128 to 255

word constants: −32768 to 65535

double-word constants: −2147483648 to 2147483647
i.e.: −2**31 to (2**31−1)

Section The length of a section name as specified with the **.section** directive must be up to eight characters.

The number of sections, within one assembly source file, is limited to 32. The first three sections are reserved for: **.text**, **.data** and **.bss**.

If there are **.ident** directives there is a **.comment** section. Therefore, you can define only 28 sections in the assembly source level.

String The string length is limited to 256 characters.

Symbol name The length of a symbol name in the Cross-reference Table (**-x** option) is truncated to 14 characters.

The length of a symbol name in the Symbol Table (**-y** option) is truncated to 14 characters.

Chapter 3
ELEMENTS OF THE ASSEMBLY LANGUAGE

3.1 INTRODUCTION

This chapter describes the elements of the CompactRISC Assembly Language. The following topics are discussed:

- Character set
- Statements
- Constants
- Symbols, symbol types, and values
- Location counter
- Expressions

3.2 CHARACTER SET

The CompactRISC Assembly Language character set consists of the following subset of the standard ASCII character set:

- Upper- and lower-case letters **A** through **Z** of the English alphabet.
- Digits 0 through 9.
- Blanks (ASCII 32), Tabs (9), Vertical Tabs (11), and Form Feeds (12).
- The following printable characters:

Character	Name	Character	Name
'	Single Quote/Apostrophe	+	Plus Sign
(	Left Parenthesis	/	Slash
)	Right Parenthesis	:	Colon
.	Period	;	Semi-Colon
_	Underscore	@	Ampersand Sign
,	Comma	[	Left Square Bracket
-	Minus Sign/Hyphen	]	Right Square Bracket

Character	Name	Character	Name
*	Asterisk	"	Double-Quote
\	Back Slash	%	Percent
~	Tilde	#	Pound Sign
^	Caret	|	Vertical Bar
&	Ampersand	<	Left Angle Bracket
$	Dollar Sign	>	Right Angle Bracket
?	Question Mark		

Carriage Return and Line Feed serve as line terminators; therefore, they cannot be entered directly into source code statements. They can be entered as their ASCII value. Any other ASCII character may appear only within quoted strings.

The CompactRISC Assembler is case sensitive, i.e., the assembler distinguishes between upper- and lower-case letters. Reserved symbols must be typed in lower-case. User symbols are interpreted exactly as they are typed.

3.3 ASSEMBLER STATEMENTS

The CompactRISC Assembly Language consists of lines of text that contain one or more statements separated by semicolons and an optional comment. A statement is an optional label followed, optionally, by a mnemonic plus its operands. Statements are composed of user-defined symbols (names and labels representing variable quantities or memory locations), reserved symbols, constant values, and delimiters.

CompactRISC Assembly Language statements are of two kinds: assembly language instructions and assembler directives. The assembly language instructions are translated directly into machine instructions so that their meanings are carried out at execution time. The Assembler directives, on the other hand, are commands to the assembler itself to carry out some action during program translation, e.g., allocating a block of memory.

Lines of CompactRISC Assembly Language code have the following form:

"([***label*** : [:]] " [***mnemonic[operands]***] [;])... [# ***comment***] .

label is an optional label. The label must be a valid symbol name and must be followed by one or two colons. See the syntax descriptions of CompactRISC assembly language directives in Chapter 6 for those directives that do not allow labels.

mnemonic is an instruction mnemonic or assembler directive. It must end with a space, tab, end-of-line, or semicolon.

operands are the operands of the instruction or of the assembler directive. The number of operands depends on the instruction or directive type. Each operand must be separated from the next operand by a comma. Spaces between operands are ignored. If the statement contains no instruction or directive, the operands must also be omitted.

comment is the optional comment. A comment must be preceded by a pound sign (#) or double slash (//).
If you use a pound sign, and the **-c** option is given, comments should not begin in column 1.

A line of CompactRISC Assembly Language code must conform to the following rules:

1. Multiple statements (i.e., label, mnemonic, and operands) must be separated by a semicolon (;).

2. If a line is terminated with a backslash (\), the next line is considered as a continuation. This is the only way to break one statement into more than one line.

3. The code line may begin in any column.

4. A line of code may be up to 64K characters in length (including EOL, the end-of-line character). However, in the listing, lines longer than 132 characters (including NL, the new-line character) are truncated.

5. A code line may consist of zero or more statements, i.e., label, mnemonic, and operands, separated by semicolons, and optionally followed by a comment.

Examples

```
1          br        START      # a branch instruction and its
                                  one operand
2          movw      r2, r3     # a move word instruction and
                                  two operands
3  END:                        # a label only
4  START:movb   r0, r1          # a label, instruction, and
                                  operands
5                              # a comment only
```

3.4 STRING AND NUMBER SYNTAX

There are four basic types of constants in CompactRISC Assembly Language statements: integer values, floating-point values, character constants, and strings. The syntax for each type of constant is defined in Section 3.4.1 through 3.4.4.

3.4.1 Integer Syntax

Integer syntax has the following form:

[***sign***] [***base***] ***digits***

sign specifies the sign. By default, the sign is positive. A negative sign may be specified with the minus sign (−).

base specifies the base. It may be one of the following:
Binary − B' or b'
Octal − O', o', Q', q' or 0 (leading digit zero)
Decimal − D' or d'
Hexadecimal − H', h', X', x', 0x (digit zero), or 0X (digit zero)
Default is decimal.

digits specifies the integer. Digits must be compatible with the specified base. Binary − 0 to 1
Octal − 0 to 7
Decimal − 0 to 9
Hexadecimal − 0 to 9 and A to F or a to f

Integer constants may have the following range of values, depending on the context in which the constant is specified: −128 to 255 for byte constants, −32768 to 65535 for word constants, and −2147483648 to 2147483647 (-2^{31} to $2^{31} - 1$) for double-word constants.

Decimal constants are sign-extended to double-words. Hexadecimal, octal and binary constants are zero-extended to double-words.

Examples

Binary	Octal	Decimal	Hexadecimal
B'11110001	O'077	D'1492	H'12ff
-B'11	-Q'5077	-999	-X'302F
b'11	123	1457	0xAB03

3.4.2 Floating-Point Number Syntax

Floating-point values may be specified in one of two forms: as a decimal number in scientific notation, or as a hexadecimal value. The CompactRISC Assembler expects floating-point numbers specified as hexadecimal values to be correctly encoded in internal floating-point format. Therefore, hexadecimal notation is most useful to the writers of compilers or optimizers.

Decimal Floating-Point Syntax

Decimal floating-point syntax has the following form:

[***decimal prefix*** **]** ***decimal value***

decimal prefix

specifies whether the constant is short or long floating-point format. It may be one of the following:

{0f | 0F} – short format floating-point value (float).
{0l | 0L} – long format floating-point value (long).

decimal value

specifies a floating-point value in scientific notation.

A decimal floating-point constant has two parts, an optional prefix that specifies short format (32 bits) or long format (64 bits) and a decimal value expressed in scientific notation.

The ***decimal value*** format is:

[***sign*** **]** ***digits*** **[** **.** **[** ***digits*** **]]** **[** **{** ***E | e*** **}** **[** ***sign*** **]** ***digits*** **]**

Mantissa Exponent

sign specifies the sign. A negative sign may be specified (–); by default, the sign is positive.

digits specify the value. Only decimal digits are permitted (0 to 9). At least one digit must precede the decimal point.

. is the decimal point.

E | e is the exponent flag. It is required when specifying an exponent.

The decimal value must be in the appropriate range for the prefix size specified or in the format that is required by the instruction. See note below.

Examples

Valid	Invalid	Comments
`3.14152`	`.0125`	`# digit before decimal point required`
`971.`	`-0.00FF`	`# decimal digits only`
`0f0.1E-14`	`0.125E999`	`# exponent exceeds limit`

Note

Assembler recognizes two types of floating-point constants: single-precision (float) and double-precision (long). Single-precision numbers occupy four bytes.
The most positive single-precision value is 3.40282346 x 10^{38}; the least positive value is 1.17549436 x 10^{-38}.
The most negative value is the negative of the most positive value. Double-precision numbers occupy eight bytes.
The most positive double-precision number is 1.7976931348623157 x 10^{308}; the least positive value is 2.2250738585072014 x 10^{-308}. The negative range is the negative of the positive value.

Hexadecimal Floating-Point Syntax

Hexadecimal floating-point syntax is of the following form:

hexadecimal prefix
hexadecimal digits

hexadecimal prefix
is one of the following:
{f' | F' | 0y | 0Y} – short format.
{l' | L' | 0z | 0Z} – long format.

f' | F' | 0y | 0Y
specifies an encoded short (32-bit) floating-point value. Must be followed by eight hexadecimal digits, if not, the assembler might generate unpredictable results.

l' | L' | 0z | 0Z
specifies an encoded long (64-bit) floating-point value. Must be followed by sixteen hexadecimal digits, if not, the assembler might generate unpredictable results.

hex digits specify the value. Only hexadecimal digits are permitted (0 to F or f). The encoded value is an exact bit representation of the resultant 32- or 64-bit value.

Examples

Valid	Invalid	Comments
`f'E01267AC`	`-F'A7261CD5`	`#no sign permitted`
`L'12A945BD4266ECF0`	`L'E596C.4BF5DB46A26`	`#no decimal point permitted`

3.4.3 Character Constant Syntax

Character constants have the following form:

{*ASCII char* | *escape sequence*}

ASCII char is any single ASCII encoded character.

escape sequence
is one of the special escape sequences, described below.

A character constant is a single ASCII character enclosed by single quotes, as in **'A'**. If the desired character is a special character, for example, the single quote itself, or if the character is not a printable character, then an escape sequence may be used to represent the character. The following rules apply to escape sequences:

- Except as noted in Table 3-1, any character preceded by the escape character backslash (\) represents that character.
- A backslash followed by one to three octal digits represents the character whose ASCII encoding is the octal value.
- Certain special characters are represented by the escape sequences specified in the escape sequence table below.

If the character constant is itself a single quote, the quote must be escaped, that is, preceded by the escape character backslash (\). Thus, the character constant single quote is (\'). Similarly, if the character constant is a backslash it must be escaped. The character constant backslash is (\\).

Other non-printable or special characters may be generated by the escape sequences in Table 3-1

Character constants may be used in expressions. The value of the constant is its ASCII encoding. If the character constant is used as an immediate operand or in an expression, it is zero-extended to the appropriate number of bytes.

Table 3-1. Escape Sequences

Escape	Value
\n	newline
\t	horizontal tab
\b	backspace
\r	carriage return
\f	form feed
\\	backslash

Table 3-1. Escape Sequences (Continued)

Escape	Value
\'	single quote
\0	ASCII character 0, or null, the C string terminator
ddd	an arbitrary byte-sized bit pattern, where ***ddd*** is one to three octal digits, i.e., the character constant "\0" represents the character with value zero.

3.4.4 String Syntax

String syntax has the following form:

"**(**{***ASCII char*** | ***escape sequence***}**)** . . ."

ASCII char is an ASCII encoded character

escape sequence
is a character sequence used to represent special or non-printable ASCII encoded characters. Refer to Table 3-1.

A string is a sequence of ASCII encoded characters enclosed by double-quotes. The same rules and escape sequence definitions specified in the description of character constants may be used in string constants. Special consideration must be given if a double-quote mark is part of the string. Strings enclosed in double-quote marks which also contain double-quotes are allowed. However, each quote which is a part of the string must be escaped, that is, it must be preceded by the escape character backslash (\ ). It is not necessary to escape the single-quote character in a string constant.

Strings may not be used in expressions.

Examples

Strings Coded In Source Statements	Generated String
`"This is a string"`	`This is a string`
`"Five O'Clock"`	`Five O'Clock`
`"\"A\" for Ampere"`	`"A" for Ampere`

3.5 SYMBOLS

A symbol is a name that refers to a memory location. Each symbol has a type and a value. The type of a symbol is either the segment in which the symbol is defined, ***external*** if the symbol is not defined in the assembly file, or ***absolute*** if the symbol is a numeric address. The value of a symbol is the address of the memory location. A symbol may have the attribute global. A symbol with the global attribute may be referenced from any software module in the program. By default, all symbols referenced but not defined are considered global.

Reserved names

Some symbol names are reserved, i.e., the instruction mnemonics, directive mnemonics, names for the registers, address mode indicators, options, scaled index qualifiers, the delimiters, and operators. You may not redefine the reserved symbols. Appendix B contains a list of the reserved symbols in the CompactRISC Assembly Language. The rest of this section and all of Section 3.6 deal with user-defined symbols.

3.5.1 Symbol Names

The name of a user-defined symbol is composed of one or more letters, digits and the characters underscore (_) and period (.). Except for temporary labels, the first character of the name may not be a digit. The name's length is limited to 64 characters.

Symbol names which include the character period (.) are assumed to be internal names generated by the CompactRISC language tools, e.g., compiler labels, Common Object File Format (COFF) section names, and reserved names. Assembly programmers should not use names which include the character period (.).

Case sensitive

The assembler is case sensitive, that is, it differentiates between upper- and lower-case letters in a user-defined symbol name. Thus, for example, the names ALPHA and Alpha are not identical and can be defined as separate symbols.

Examples

Valid	Invalid	Comment
`SYMBOL`	`$YMBOL`	`# ''$'' dollar-sign character illegal`
`_ALPHA`	`2ALPHA`	`# first character cannot be number`
`REG2`	`r1`	`# r1 is reserved symbol`

3.5.2 Symbol Types

The type of a symbol specifies the segment of the object file in which it occurs. All labels defined within a segment have the type of that segment. For example, all symbols defined in the **.text** segment (i.e., following the **.text** directive) are of type text. The address of symbols associated with object file segments must be updated at link time, when the linker associates the object file segment with memory locations.

Undefined symbols

Undefined symbols are of type external. The value of an undefined symbol is resolved by the linker. Numeric addresses are of type absolute. The value of absolute symbols is unaffected by linkage.

A symbol's type determines the default addressing mode the assembler uses when the symbol is referenced. The following table lists symbol types, the associated object file segment and the default addressing mode for references to the symbol.

Type	Segment	Default Addressing Mode
Text	Text or code segment	PC Relative
Data	Initialized data segment	Absolute
Bss	Uninitialized data segment	Absolute
External	-	Absolute
Absolute	-	Absolute
<user-defined>	<defined by attributes>	Absolute

The type of a symbol limits the places where the symbol may be used as an operand and the way its value may be manipulated in expressions. Expressions also have one of the above types. The type of an expression is determined by the types of the symbols it contains.

Following are descriptions of each of the symbol types:

Text symbols

- All symbols defined in the .text segment, i.e., labels following a **.text** directive, are of type text. All symbols or expressions of type text represent addresses within the text segment of the program's object code. The text segment contains program code and read-only data.

Data symbols

- All symbols defined in the .data segment, i.e., labels following a **.data** directive, are of type data. All symbols or expressions of type data represent addresses within the initialized data segment of the program's object code.

Bss symbols

- All symbols defined in the uninitialized data (**.bss**) segment are of type bss. Symbols defined by the **.bss** directive are of type bss, as are labels defined after a **.udata** directive. All symbols or expressions of type bss represent addresses within the uninitialized data segment of the program's object code.

External symbols

- All undefined symbols are of type external. Symbols defined using the **.comm** directive are also of type external.

Absolute symbols

- All symbols assigned numeric values are of type absolute. Absolute symbols specify an absolute numeric address. They are not relative to any segment of the object file. Symbols of type absolute may only be defined using the **.set** directive.

User-defined symbols

- All symbols defined in a section, following the .section definition, are of the type of the section.

3.5.3 Global Symbols

Global symbols are used by multiple software files. The symbol must be defined exactly once. The defining module exports the symbol, that is, makes the symbol available for import by one or more additional software files. Global symbols must be declared for export by the defining module with the **.globl** directive. Undefined symbols intended to be imported from other software modules should also be declared with the **.globl** directive, although this is not required.

Except for temporary labels, every user-defined symbol must be defined exactly once. A symbol definition assigns a value and type to a symbol name. There are several formats for defining symbols. The formats form four groups:

- Labels.
- Symbols defined by the **.set** directive.
- Uninitialized symbols defined by the **.bss** directive.
- Common symbols defined by the **.comm** directive.

External, or undefined, user symbols may be declared for import with the **.globl** directive. Such a declaration does not define the symbol. Any symbol that is referenced in an assembler statement but not defined within the assembly is assigned type external.

Labels

The formats permitted for label definitions are:

symbol name :
or

symbol name ::
or

symbol name : assembly statement
or

symbol name :: assembly statement

assembly statement
may be any assembly statement except those directives that do not accept labels. See Chapter 6 for detailed description of the syntax of all the CompactRISC assembly language directives.

In each case, the current value and the type of the location counter is assigned to the symbol, see Section 3.7. The second construction (using "*::*") also sets the global attribute on the symbol, see Section 6.6.

Temporary Labels

temporary label:

temporary label
consists of a digit from 1 to 9.

A temporary label consists of a digit from 1 to 9, followed by a colon. Reference to the label is via the symbols ***nf*** and ***nb***, where ***n*** specifies temporary label ***n***, where ***f*** means forward, and ***b*** means backwards. All referenced temporary labels must be defined somewhere within the program. Temporary labels may not be exported. There is no limit on the number of times that a temporary label may be redefined. The following symbols are reserved:
1f 2f 3f 4f 5f 6f 7f 8f 9f 1b 2b 3b 4b 5b 6b 7b 8b 9b
Temporary labels are most useful in conjunction with macros.

Example

```
1                                       #SR encoding
2     T00000000                         9:
3     T00000000       84008400          .space 10
                      84008400
                      8400
4     T0000000a       aabe0a00          br 7f
5     T0000000e       aabef2ff          br 9b
6     T00000012       aa2e              br 9f
7     T00000014                         7:
```

```
8    T00000014                    9:
9    T00000014                    7:
10   T00000014      aa0e          br 7b
```

In this program, the branch on line 3 refers to label 7 on line 6, the branch on line 4 refers to label 9 on line 1, the branch on line 5 refers to label 9 on line 7, and the branch on line 9 refers to label 7 on line 8.

Defining Symbols with the .set Directive

The format for symbol definition using the **.set** directive is:

.set *symbol name, expression*

The statement assigns the value and type of ***expression*** to the symbol. The expression may not be of type external (undefined), nor a forward reference. For more information, see Section 6.2.1.

Defining Uninitialized Symbols with the .bss Directive

The format for the definition of uninitialized symbols using the **.bss** directive is:

.bss ***symbol name, expression1, expression2***

This form is used only for uninitialized data (bss) symbols. The symbol is assigned type bss and the value of the current bss location counter after it is aligned to a multiple of ***expression2***. See Section 6.7.4 for a description of the **.bss** directive.

Defining Common Symbols

The format for the definition of uninitialized, common symbols using the **.comm** directive is:

.comm *symbol name, expression*

The type of common symbols is external. If no software module defines a global symbol by this name, the linker allocates an uninitialized storage area whose size is the largest ***expression*** specified by any **.comm** directive for this ***symbol***. See Section 6.6.2 for a description of the **.comm** directive.

3.6 LOCATION COUNTER

The CompactRISC Assembler manages a location counter that keeps track of the current relocatable memory address. The current location counter is set to the type of the segment that is being assembled and the value of the next available address within the segment. The current location counter is initialized to the TEXT segment, address 0 at the start of assembly.

The assembler re-initializes the current location counter to a new value (i.e., a new type and offset) each time a segment control directive is encountered. The segment control directives determine the segment into which the following code should be assembled. On encountering a segment control directive, the assembler saves the next available address in the previous segment before entering the new segment, so that it is able to restore the previous address if the previous segment is reopened. The assembler maintains a saved location counter for each object file segment (text, data, bss, user-defined sections and dsects) as well as each user-defined segment.

When a statement is processed, the assembler increments or decrements the location counter by the number of bytes of object code generated or by the amount of data storage allocated.

The location counter symbol, (.) period, is a special token which may be used in expressions or instruction operands to specify the location counter's current value (before it has been incremented). The symbol may appear alone or as a term in an arithmetic expression (addition or subtraction only).

Examples

```
1.    .set          A, .

2.    bne           .-8
```

In example 1, (**.**) specifies the current address. The symbol **A** is assigned the current location counter address.
In example 2, the expression **.-8** specifies the current address minus 8.

3.7 EXPRESSIONS

An expression is a combination of terms and operators which evaluate to a single value and type. Valid expressions include addresses and integer expressions. Floating-point expressions are not valid.

Terms in expressions may be constants or symbols, including the location counter symbol (.), see Sections 3.4, 3.5 and 3.6. The type of the term determines the way in which the term may be combined with other terms and operators. Section 3.7.2 defines the effect the type of a term has on the result of an expression.

In architectures where some pc bits are implied (i.e., CR16) relative terms may have a special attribute, in addition to the section. There may be terms with code-label attributes. See Sections 6.3.3, 5.7, and 6.6.3.

Operators

Operators in expressions are the special symbols which define arithmetic and logical operations. An operator has the following characteristics:

- An operator has a level of precedence which affects the order in which the CompactRISC Assembler evaluates an expression containing the operator.
- An operator defines the type of the term(s) that may be used with the operator and the location of the term(s) relative to the operator.

Table 3-2 lists all CompactRISC Assembly Language operators in order of precedence.

Table 3-3 defines the type and order of the terms that may be used with the operators.

Table 3-2. Operator Precedence

Precedence	Operator	Name	Operation
Unary Operator			
1	-	Unary minus	Two's complement.
1	~	Unary complement	One's complement.
Binary Operator			
2	*	Multiply	Multiply 1st term by 2nd.
2	/	Divide	Divide 1st term by 2nd.*
2	%	Modulus	Remainder from 1st term divided by 2nd.**
2	<<	Shift left	Shift 1st term by 2nd; emptied bits are zero-filled.
2	>>	Shift right	Shift 1st term by 2nd; emptied bits are zero-filled.
2	~	Logical OR / complement	Bit-wise OR of 1st term and one's complement of 2nd term.
3	&	Logical AND	Bit-wise AND of 1st and 2nd terms.
3	|	Logical OR	Bit-wise OR of 1st and 2nd terms.
3	^	Logical XOR	Bit-wise XOR of 1st and 2nd terms.
4	+	Add	Add 1st and 2nd terms.
4	-	Subtract	Subtract 2nd term from 1st term.

* Rounds toward 0, *e.g.*, -7/3 = -2 and 7/3 = 2

** *e.g.*, -7%3 = -1 and 7%3 = 1.

Table 3-3. Types and Operators

Unary Operators			
Operator		Term1	Operation
-		abs	Type abs.
~		abs	Type abs.
Binary Operators			
Term1 Type	Operator	Term2 Type	Result Type
abs	*	abs	Type abs.
abs	/	abs	Type abs.
abs	%	abs	Type abs.
abs	<<	abs	Type abs.
abs	>>	abs	Type abs.
abs	~	abs	Type abs.
abs	&	abs	Type abs.
abs	|	abs	Type abs.
abs	^	abs	Type abs.
abs	+	abs	Type abs.
abs	-	abs	Type abs.
rel	+	abs	Type rel.*
rel	-	abs	Type rel.*
rel	-	rel	Type abs.**
ext	+	abs	Type ext.
ext	-	abs	Type ext.

Note

abs — Any term of type absolute.

rel — Any term of relative type, i.e., ***text***, ***data***, etc.

ext — Any term of type external, undefined.

* The type of the result matches the type of the relative term in the expression.

** Term1 and Term2 must be the same type, the result is type absolute.

3.7.1 Rules for Expressions

The rules for forming and evaluating expressions are as follows:

- All unary operators must precede a single term and cannot be used to separate two terms.
- All binary operators must separate two terms. For example, the expression **8*4** is legal, but **8**4** is not.
- Compound expressions are valid. An expression may be constructed from other expressions using unary and binary operators. For example, the two individual expressions **A+1** and **B+2** may be combined with a multiply operator and parentheses to form the single expression **(A+1)*(B+2)**. Note that the parentheses override the default precedence rules.
- Evaluation of an expression is governed by three factors:
  - Parentheses - expressions enclosed in parentheses are always evaluated first. For example, the expression **8/4/2** evaluates to 1, but the expression **8/(4/2)** evaluates to 4.
  - Precedence Groups - an operation of a higher precedence group is evaluated before an operation of a lower precedence whenever parentheses do not otherwise determine the evaluation order. For example, the expression **8+4/2** is evaluated as 10, but the expression **8/4+2** is evaluated as 4.
  - Left to Right Evaluation - expressions are evaluated from left to right whenever parentheses and precedence groups do not determine evaluation order. For example, the expression **8*4/2** is evaluated as 16, but the expression **8/4*2** is evaluated as 4.

3.7.2 Types in Expressions

The type of the result of an expression depends on the type of the terms and the operations performed. The rules for types in expressions are as follows:

- **Expressions with terms having absolute type**
  Terms with absolute type may be added, subtracted, multiplied, etc. All operators are allowed. The result is always an absolute type.

Examples

```
1.   21 * 5          # result is 105
2.   21 / 5          # result is 4
3.   21 % 5          # result is 1
4.   21 & 5          # result is 5
5.   21 << 5         # result is 672
```

```
6.    21 >> 5          # result is 0
7.    21 + 5           # result is 26
8.    21 - 5           # result is 16
9.    21 | 5           # result is 21
10.   21 ^ 5           # result is 16
```

- **Expressions combining terms having relative and absolute types**
  The only valid operations between terms with relative types and terms with absolute type are addition and subtraction. The operations take place between the values of the first and the second terms and the result is assigned the type of the relative term.

Addition

Addition is commutative. An absolute term may be added to a relative term or a relative term may be added to an absolute term, the result is the same in either case.

Subtraction

Subtraction is not commutative. An absolute term may be subtracted from a relative term. A relative term may not be subtracted from an absolute term.

Example

```
1               .set      ZERO, 0
2               .set      TEN, 10
3               .set      COUNT, 30
4
5               .udata
6     Size:     .blkd
7     Start:    .space    (COUNT * 4)
8     End:      .blkd
9
10              .text
11              movb      $ZERO, Start + ZERO
12              movb      $TEN, TEN + Start
13              movd      (End - TEN), r0
```

In the preceding example several symbols and expressions are used. The symbols ZERO, TEN, and COUNT are of type absolute. The symbols Size, Start, and End are of type bss, refer to Section 3.5.2.

The expression “(COUNT * 4)” in line 7 combines two absolute terms, the result is absolute.

The expression “Start + ZERO” in line 11 adds a relative type to an absolute type. The result is type bss.

The expression “TEN + Start” in line 12 adds an absolute type to a relative type. The result is type bss.

The expression “End - TEN” in line 13 subtracts an absolute type from a relative type. The result is type bss.

- **Expressions combining terms having relative types**
  Terms with relative or absolute type may be subtracted from terms with the same type. No other operator is allowed. The result is always an absolute type.

Example

```
1               .set      COUNT, 30
2
3               .udata
4    Size:      .blkd
5    Start:     .space    (COUNT * 4)
6    End:       .blkd
7
8               .text
9               movw      $(End - Start)/4, Size
```

The expression “End - Start” in line 9 subtracts a relative term from another relative term. Since both symbols are of the same type (bss), this is a legal expression. The result is of type absolute, i.e., the absolute number of bytes between the two labels. The result of the subtraction is then divided by 4, both terms are type absolute and the result is type absolute.

Note that “(End - Start)/4” is not a legal expression without parentheses. Division is of higher precedence than subtraction, but a relative term may not be divided.

- **Expressions with terms having external and absolute type.**
  Terms with absolute type may be added to or subtracted from terms with external type. No other operations are allowed. The result always has external type. A term of type absolute may be subtracted from a term of type external, but a term of type external may not be subtracted from a term of type absolute. The first term of the subtraction must be the term of type external.

Example

```
1          .set          ZERO, 0
2          .set          TEN, 10
3          .set          COUNT, 30
4
5          .globl        Start
6          .globl        End
7
8          .text
9          movb          $ZERO, Start + ZERO
10         movb          $TEN, TEN + Start
11         movw          (End - TEN), r0
```

The expression “Start + ZERO” in line 9 adds an absolute type to an external (undefined) type. The result is type external.

The expression “TEN + Start” in line 10 adds an external type to an absolute type. The result is type external

The expression “End - TEN” in line 11 subtracts an absolute type from an external type. The result is type external.

- **Expressions with character constants**.
  Character constants may appear as terms in expressions. When a character constant is used this way, it is converted to an integer constant. Integer constants are stored in four bytes; the assembler fills the higher order bytes with zero.

Examples

```
1.   .set      UPCASE,   'A' - 'a'      # result is -32
2.   .set      LOWCASE,  'a' - 'A'      # result is 32
```

3.7.3 Encoding of Expressions

Expressions are encoded according to their size, and the addressing mode.

Some expressions cannot be resolved by the assembler since they contain a reference to an external symbol. If the assembler can not resolve the expression, it assumes the maximum size for it, unless explicitly specified otherwise. You may determine the size of the encoding using the modifier ***exp_len***.

Example

```
br extsym+100:m
```

As an alternative you can force all the unresolved expressions in a file to a maximum size using the command line option **-d*exp_len***. Possible values for ***exp_len*** are **s, m** or **l**. The exact meaning of the value depends on the particular architecture and the addressing mode. Generally speaking **s**, **m**, and **l** stand for small, medium and large respectively. A smaller encoding size is generally suitable for expressions that are resolved to smaller numbers. However, due to some encoding peculiarities designed to save code space, this is not always the case.

The directive ***.code_label*** affects the encoding of symbols that appear in immediate operands. In this case there is an implied zero least significant bit for the symbol address (since it’s a CompactRISC instruction address is always even) and so only bits 1-16 of the symbol are encoded and bit 0 is not. Refer to Section 6.6.3.

Chapter 4
ASSEMBLER PROGRAMS

4.1 INTRODUCTION

This chapter describes the structure of CompactRISC Assembly Language programs and how the CompactRISC Assembler assigns memory addresses to symbols, instructions, and data. In particular, it describes:

- Program Structure
- Program Segments
- User-Defined Dummy and Comment Segments
- Linkage and Relocation Modes

4.2 ASSEMBLER PROGRAM STRUCTURE

The structure of a CompactRISC Assembly Language program reflects the structure of the object file, and the layout of the program image in memory. The structure allows instructions and data to be grouped into logical segments that occupy contiguous memory. Each segment is "atomic", i.e., segments may be combined into larger units, but not broken into smaller units.

Program segments

Every object file contains at least three program segments: text, data and bss. These segments correspond to the **`.text`**, **`.data`**, and **`.bss`** sections of Common Object File Format (COFF). The text segment contains program instructions and constant data, the data segment contains writable, initialized data, and the bss segment contains uninitialized data. No object file space is allocated for the bss segment.

You may create user-defined segments with the assembler directives **`.dsect`** and **`.section`**, or comment segments with the assembler directive **`.ident`**. The CompactRISC Assembler maintains a location counter for each object file segment.

In architectures which do not allow non-aligned references, i.e., an entity referenced as a word/double-word must be on a word/double-word boundary (SR) it is recommended to separate the data into sections according to their alignment. Thus you can create two levels of separation: BSS, read-only-data, and initialized data, where each is divided into three according to its alignment to 4, to 2 and to 1. This ensures alignment at link time.

4.3 PROGRAM SEGMENTS

Every assembly program consists of one or more program segments. A program segment is a block of sequential statements which are placed in contiguous memory and treated as a unit with common properties, e.g., access protection. Every program contains the following types of segments:

- Text or Program Code Segment
- Initialized Data Segment
- Uninitialized Data Segment (bss)

A program segment begins and ends with one of the segment control directives (Section 6.7) and contains any number of statements.

Example

```
.text           # specifies the start of a program code segment
statement-1     # assembler statements
statement-2
.
.
.
statement-n
.data           # specifies start of a data segment
                # a segment terminates with another segment
                # control directive or EOF
```

Text segment

The text segment contains instructions and constant data. After every statement, the text segment location counter is incremented by the number of bytes generated for that statement. The location counter of the text segment may not be decremented.

The text segment is written to the **.text** section of the object file. Each text address maps to a location in the **.text** section of the object file.

All symbols defined in the text segment are of type text. References to locations in the text segment are addressed in Program Counter Relative addressing mode.

Related directive

.text (Section 6.7.2).

Initialized data segment

The initialized data segment contains writable, initialized data. After every statement, the data segment location counter is incremented by the number of bytes generated for that statement. The location counter of the data segment may not be decremented.

The data segment is written to the **.data** section of the object file. Each data address maps to a location in the **.data** section of the object file.

All symbols defined in the **.data** section are of type data. References to locations in the data segment are addressed in Absolute addressing mode.

Related directive

.data (Section 6.7.3).

Uninitialized data (bss) segments

The uninitialized data or bss segment consists of storage allocated for uninitialized data. After every statement following the **.udata** section control directive, the bss segment location counter is incremented by the number of bytes allocated by that statement. The bss location counter is also updated by the **.bss** directive. No code or data may be generated in the bss segment. The location counter of the bss segment may not be decremented.

Each bss address maps to a location in the **.bss** section of the object file, although the **.bss** section of the COFF file contains no actual data. Storage space is allocated and zeroed at load time.

All symbols defined in the **.bss** section are of type bss. References to locations in the bss segment are addressed in Absolute addressing mode.

Related directives

.udata (Section 6.7.5), **.bss** (Section 6.7.4).

4.4 USER-DEFINED, DUMMY AND COMMENT SEGMENTS

This section describes user-defined, dummy and comment segments.

User-defined segments

User-defined segments are generated with the **.section** directive. These segments occupy real space in the object file and, depending on the attributes selected, may appear in the linked file. Symbols declared in these segments are addressed via the absolute addressing mode.

Related directives

.section (Section 6.7.6).

Dummy segments

The dummy segments are generated with the **.dsect** directive. These segments do not allocate storage, nor do they contain generated code or data. If the dummy segment is of a relative type, it overlays some portion of that type of segment. For example, a user-defined dummy segment might be used to overlay one or more structured data types on a pool of storage. Dummy segments of type absolute may be used to generate symbolic positive or negative offsets from the stack register for function arguments or local variables.

Every statement following a **.dsect** directive increments or decrements the location counter for the dummy segment by the number of bytes specified by that statement.

Related directive

.dsect (Section 6.7.1).

Comment Segments

Comment segments are generated with the **.ident** directive and corresponds to the **.comment** section of the COFF file.

Chapter 5
INSTRUCTION OPERANDS

5.1 REGISTER OPERANDS

register

r<n>	is a general-purpose register
sp	is the stack pointer register (and is also a general-purpose register)
ra	is the register holding the return address of a function (and is also a general-purpose register)
era	(CR16B large programming model only) is a register that holds the most significant bits of the return address (and is also a general-purpose register). It replaces r13 in other programming models.

A register operand specifies a general-purpose register. The register contains the operand for the instruction.

Examples

```
1.   movw          r0,r1
2.   jump          ra
```

The first example copies a word from **r0** to **r1**.

The second example jumps to the address which resides in register **ra**.

5.2 REGISTER PAIR OPERAND (CR16B LARGE MODEL ONLY)

(register<n+1>, register<n>)

A register pair operand is any pair of consecutive general purpose-registers (in descending order). The operand address is formed by concatenating the values of the two registers. Note that the most significant bits of the address are always in the register with the higher index.

Examples

```
1.    movd          $label,(r6,r5)
2.    jump          (era,ra)
```

The first example loads an address of a label into the register pair **r5-r6**.

The second example jumps to an address which resides in the register pair **ra-era**. This is the common way to return from a function in the CR16B large programming model.

5.3 PROCESSOR REGISTER OPERANDS

procreg

Specifies a dedicated register. ***procreg*** must be one of the following register names[1]:

pc	-	Program Counter.
isp	-	Interrupt Stack Pointer
intbase	-	Interrupt Base Register
psr	-	Processor Status Register.
cfg	-	Configuration Register.
dsr	-	Debug Status Register.
dcr	-	Debug Condition Register.
car	-	Compare Address Register.

The functions of the dedicated registers are described in the datasheet of the relevant microprocessor.

Examples

```
1    lpr       r1,psr
2    spr       cfg,r0
```

The first example loads the value in **r1** into the Processor Status Register, **psr**.

The second example stores the value of the Configuration Register, **cfg**, into **r0**.

5.4 REGISTER RELATIVE OPERANDS

expression{:exp_len}(register)

expression is an expression which evaluates to an absolute value, or to a relative value type (see Section 3.6).
During assembly, or link process, the expression is evaluated to produce an intermediate value.

---

1. Only part of this register list is available on all CompactRISC architecture derivatives. A register name is recognized by the assembler only if it is supported by the target architecture.

:*exp_len* is an optional field that determines the length of the expression field in the instruction's encoding.
The colon is required.
exp_len can be one of the following (see Section 3.7.3):
s - specifies small.
m - specifies medium.
l - specifies large.

(register) is one of the general purpose registers, **r<n>**, or the stack pointer, **sp**, or the return address register **r17**.
Parentheses are required.

A Register Relative operand specifies an operand at a memory address. The address is the sum of the evaluated expression and the contents of the register.

The architecture supports a few different sizes of intermediate values, to decrease code size. Intermediate value can be of small, medium, or large size. (See the datasheet of the relevant microprocessor.) When ***exp_len*** is not specifically mentioned in the instruction, the assembler attempts to use the smallest size that can still hold the expression value. For expressions that are not defined at assembly time, large displacement is used. You may overwrite this default by explicitly specifying the ***exp_len*** option of the operand. If the evaluated expression does not fit in the desired length, an error is issued.

Example

```
loadb           5(r0),r1
```

This example takes the contents of **r0**, adds 5, and uses the result as the address of the operand to be loaded to **r1**.

5.5 PROGRAM COUNTER RELATIVE OPERANDS

expression{*:exp_len*}

expression is a legal expression of any type which is the target of a branch instruction.

:exp_len is an optional field that determines the length of the expression field in the instruction.
The colon is required.
exp_len can be one of the following (see Section 3.7.3):
s - specifies small.
m - specifies medium.
l - specifies large.

A Program Counter relative operand specifies an operand at a memory address which is the destination of a branch instruction. The assembler converts the address to an offset from the current value of the location counter.
To minimize the code size of branch instructions, the evaluated expression field can be small, medium or large size (See the datasheet of the relevant microprocessor.) When ***exp_len*** is not specifically mentioned in the instruction, the assembler attempts to use the smallest size that can still hold the expression value. For expressions that are not defined at assembly time, large displacement is used. You may overwrite this default by specifying the ***exp_len*** option of the operand. If the evaluated expression does not fit in the desired length, an error is issued.

The most common way of specifying Program Counter relative operands is by writing a "label" at the branch target, and specifying this "label" as the operand to the branch instruction. Alternatively, write an expression of the form "*+absolute_value"; in this case the assembler uses only the absolute value specified when encoding the instruction.

Example

```
br              L1:m
```

The assembler generates a branch instruction with a medium size displacement whose content is the difference between the address of the branch statement and the address of **L1**.

5.6 FAR RELATIVE OPERANDS

expression{(register, register)

expression is an expression which evaluates to an absolute value, or to a relative value type (see Section 3.6).
During assembly, or the link process, the expression is evaluated to produce an intermediate value.

(**register**, **register**)
two consecutive general purpose registers of the type:
(**Rb+1**, **Rb**).

The far relative mode is an extension to the register relative mode designed for 16-bit architectures like the CR16A and CR16B. The CR16A and CR16B have 18-bit and 21-bit address spaces, respectively. You can not use the register relative addressing mode to refer to an address which is greater than, or equal to, 65536 (2^{16}), since the base register, which is a general-purpose register, is only 16-bit wide. In this case, you can use the far relative addressing mode. In far relative addressing mode, the base address is stored in two consecutive general-purpose registers.

Example `loadw 50(r2,r1),r3`

takes the address that is formed by concatenating the contents of **r2** and **r1**, adds 50, and uses the result as the address of the operand to be loaded to **r3**.

5.7 IMMEDIATE OPERANDS

$[qualifier] expression{:exp_len}

qualifier **hi** or **low** in architectures where the address space is larger than can be expressed in a word.
hi *expression* yields the higher word and **low *expression*** yields the lower word.

expression is one of the following:
- character constant
- a legal expression of any type (See Section 3.7)

:exp_len is an optional field that determines the length of the expression field in the instruction's encoding.
The colon is required.
exp_len can be one of the following (see Section 3.7.3):
s - specifies small.
m - specifies medium.
l - specifies large (CR32 only).

Immediate operands are encoded into the Immediate addressing mode; thus the operands' value is stored in the instruction stream.

If the expression is a relative type or an external, undefined type, the assembler generates a relocation entry for the operand. The linker uses the relocation entry to update the operand address at link time.

If the expression is of type code-label (see Section 6.6.3) then in architectures where the pc has implied bits, the expression is adjusted accordingly. In CR16, for example, it is divided by two. In CR32A there are no implicit bits.

To minimize the code size of immediate operands, the evaluated expression field can be small, medium or large size. When ***exp_len*** is not specifically mentioned in the instruction, the assembler attempts to use the smallest size that can still hold the expression value. For expressions that are not defined at assembly time, the maximum available displacement is used. You may overwrite this default by specifying the ***exp_len*** option of the operand. If the evaluated expression does not fit in the desired length, an error is issued.

The range of immediate operands is limited by the length specifier of the instruction (see the datasheet of the relevant microprocessor).
No floating point expressions are allowed.

Examples

```
movb            $5,r0
```

loads the value 5 to the least significant byte of **r0**.

```
movd            $_stremp,r0
```

loads the address of the function **stremp** into **r0**.

```
# CR16A
.code_label     _func
movw            $_func, r0
```

loads the address of **_func**, shifted right by 1, into **r0**.

```
# CR16A
movw            $hi  table, r1
movw            $low table, r0
```

loads the higher word of **table** into **r1**, and the lower word into **r0**.

```
# CR16B
.code_label     _func
movd            $_func, (r1,r0)
```

loads the address of **_func**, shifted right by 1, into register pair **r0-r1**.

5.8 ABSOLUTE OPERANDS

expression*{:**exp_len**}*

expression is a legal expression of any type (see Section 3.7).

:exp_len is an optional field that determines the length of the expression field in the instruction.
The colon is required.
exp_len can be one of the following (see Section 3.7.3):
s - specifies small.
m - specifies medium.
l - specifies large (CR32 only).

As absolute operand specifies the absolute memory address of an operand.

To minimize the code size of instructions that use absolute operands, the evaluated expression field can be small, medium or large size. When ***exp_len*** is not specifically mentioned in the instruction, the assembler attempts to use the smallest size that can still hold the expression value. For expressions that are not defined at assembly time, the maximum available displacement is used. You may overwrite this default by specifying the ***exp_len*** option of the operand. If the evaluated expression does not fit in the desired length, an error is issued.

Example `loadw _a, r0`

puts the contents of **_a** in **r0**.

5.9 STATIC-BASE RELATIVE OPERANDS

`^expression`

expression is an absolute expression.

A static-base relative operand is a special case of register-relative operand. The operand is encoded relative to a predefined based address and this base address is assumed to be the contents of **r13** (**r12** in the CR16B large programming model). This kind of operand may be useful as an alternative to absolute operand e.g., if you want to refer to all symbols using a register-relative addressing mode rather than absolute addressing mode. A relocation entry of the form SBREL is generated to enable the linker to modify the displacement. The linker encodes the displacement relative to the predefined base address given by the symbol **_STATIC_BASE_START**. It is the programmer's responsibility to assign this address to **r13** (**r12** in CR16B large programming model) e.g., at program initialization.

This operand type is currently supported only in the CR32A architecture.

Example `storw r1, ^_a`

stores the contents of **r1** in **_a** and uses register-relative addressing mode with **r13** (**r12** in the CR16B large programming model) as the base address register.

5.10 EXCEPTION OPERANDS

exception_name
exception_number

exception name
is one of the valid exceptions defined for the CompactRISC architecture. For a list of valid exception names, refer to the appropriate CompactRISC core architecture specification.

exception_number
is the number of the exception entry in the trap vector.

Exception name/number operands are legal only as operands of the EXCP instruction. Each exception that is defined in the architecture has a name associated with it, and a number which the assembler puts in the instruction in place of this name. The assembler treats the exception names as reserved words; they may not be used as labels or as operands to instructions other then EXCP.

Example

1. `excp bpt`
2. `excp 8`

The **excp** instruction activates a trap according to the number of the interrupt vector associated with the exception name specified.

In example 1, **bpt** stands for break-point trap.

Example 2 is similar to example 1, except that the specific number is in the instruction instead of the name.

Chapter 6
ASSEMBLER DIRECTIVES

6.1 INTRODUCTION

Directives are commands to the assembler which allow the programmer to control the assembler in its generation of object code and production of listings.

The CompactRISC Assembler directives are divided into functional groups as follows:

Directive	Function	Section
Symbol Creation	Assigns a name, type, and value to a symbol.	6.2
Data Generation	Initializes a block of memory with constant values.	6.3
Storage Allocation	Reserves a block of memory for data storage.	6.4
Listing Control	Controls format of program listings.	6.5
Linkage Control	Exports and imports data and procedures.	6.6
Segment Control	Defines physical or logical image segments.	6.7
Filename	Names the source file.	6.8
Symbol Table	Specifies symbol table entry data.	6.9
Line Number Table	Specifies a line number table entry.	6.10
Macro Support	Provides macro and conditional assembly support.	6.11

The remainder of this chapter discusses these directives in detail.

6.2 SYMBOL CREATION DIRECTIVE

The symbol creation directive causes the assembler to compute the value of an expression and assign that value to a symbol name.

Directive	Function
`.set`	creates a symbol name

6.2.1 .set

.set ***symbol, expression***

.set is the directive name.

symbol is a symbol name as defined in Section 3.5.

expression is a constant or an expression. It may evaluate to any type.

The **.set** directive causes the CompactRISC Assembler to compute the value of ***expression*** and assign this value to the symbol name. ***expression*** may evaluate to any type except undefined, refer to Section 3.7. The ***expression*** may not be of type external (undefined), nor a forward reference.

For each symbol defined with the **.set** directive, the CompactRISC Assembler enters the symbol name and value in its internal symbol table. The symbol may then be used in expressions in subsequent portions of the assembly.

Example

```
1    .set  SYMBA,         5
2    .set  SYMBB,         LABELA + SYMBA
3    .set  SYMBC,'        A'
```

Line 1 defines the symbol SYMBA and assigns it the value 5.

Line 2 defines the symbol SYMBB and assigns it the value of LABELA+SYMBA. If SYMBA has the value 5, then SYMBB is assigned the value of LABELA+5 and the type of LABELA.

Line 3 defines the symbol SYMBC and assigns it the value of the 'A' expression. Note that only single character constants may be used in expressions (refer to Section 3.7.2).

6.3 DATA GENERATION DIRECTIVES

The data generation directives place constant data in the instruction stream during assembly-time. The data generation directives are:

Directive	Function
.ascii	assigns ASCII encoded textual data
.byte	assigns byte-long data
.word	assigns word-long data
.double	assigns double word-long data

Directive	Function
`.float`	assigns single-precision floating-point number
`.long`	assigns double-precision floating-point number
`.field`	assigns bit field

Each of the above directives places one or more bytes of data in the object code of the program currently assembling. Data generation directives may be specified only in Program Code segments where data is written to the object file (*i.e.*, when the location counter is in the text segment, the data segment, or a user-defined segment).

All the numeric data generation directives, *i.e.*, all directives listed except **`.field`** and **`.ascii`**, have the following form:

[label]directive ({[[repetition-factor]] expression | string}),,,

The ***directive*** stores the ***expression*** value in the instruction stream. If a ***repetition-factor*** is specified, the ***directive*** stores the ***expression*** value in consecutive locations as specified by the ***repetition-factor***. A ***label*** is optional.

The **`.byte`**, **`.word`**, **`.double`**, **`.float,`** and **`.long`** directives may specify one or more ***expressions***. Multiple ***expressions*** must be separated by commas. Each ***expression*** is evaluated and stored in the number of bytes specified by the directive. An ***expression*** must evaluate to an absolute value within the range specified by the directive, but ***expressions*** for the **`.long`** and **`.float`** directives should evaluate to a long value. (The assembler evaluates all floating-point expressions as long floating-point numbers. If necessary, the result is then converted to a single-precision floating-point value.) If no expression is specified, the CompactRISC Assembler issues an error message and terminates code generation.

A ***repetition-factor*** may be any expression which evaluates to a positive absolute value. The ***repetition-factor*** expression may use symbolic values, but no forward symbol references are allowed.

The **`.byte`**, **`.word`**, and **`.double`** directives may be used for both signed and unsigned numbers.

6.3.1 .ascii

`[`*`label`*`] .ascii "`*`string`*`"`

label is an optional label.

.ascii is the directive name.

"string" specifies a string constant. The string must not contain an embedded new-line. You may use the escape sequence "\n" to enter a new-line into a string constant.

The **.ascii** directive generates textual data. The CompactRISC Assembler places the text in the instruction stream at the current address specified by the location counter. The assembler stores the ASCII value of each character in the ***string*** in one byte, placing the first character of the string at the lowest byte address and the last character of the string at the highest byte address. Unprintable ASCII characters may be included via the escapes defined in Section 3.4.3. No special string terminator is implied or inserted by the assembler.

Example

```
1                       .data
2   D000000004572726f.ascii"Error: unknown command.\n"
          723a2075
          6e6b6e6f
          776e2063
          6f6d6d61
          6e642e0a
3   D0000001855736167.ascii"Usage: list [-tdrx]"
          653a206c
          69737420
          5b2d7464
          72785d20
```

Line 2 places the ASCII character string "Error: unknown command." followed by a new-line character (\n) in consecutive bytes beginning at address 0 of the data segment.

Line 3 places the character string "Usage: list [–tdrx]" in consecutive bytes starting at address 00000018 in the data segment.

6.3.2 .byte

[***label***].byte({[[***repetition-factor***]] ***expression*** | ***string***}),,,

label is an optional label.

.byte is the directive name.

[*repetition-factor*]
(optional) specifies the number of occurrences of the specified data byte. It must be an expression which evaluates to a positive absolute value. If the ***repetition-factor*** is specified, it must be enclosed in "[ ]" brackets.

expression specifies the data byte value. This value must be in the range of -128 to 255.

string specifies a string constant. The assembler issues a warning if the string contains an embedded new-line. Therefore, it is preferable to use the "\n" escape sequence.

The **.byte** directive generates one or more byte constants. The CompactRISC Assembler places the constants in the instruction stream at the current address specified by the location counter. If multiple constants are specified **(**e.g., ***repetition-factor*** is greater than one or more than one ***expression*** is given), the constants are stored in consecutive bytes beginning at the current address.

If a ***string*** is specified, the assembler places the ***string***, starting with the first character in the string, in one or more bytes beginning at the current address. The assembler stores the ASCII value of each character in the ***string*** in one byte. Character constants appearing as terms in the ***expression*** are converted to integers (see Section 3.7.2).

Example

```
1    T00000000      81        .byte     129
2    T00000001      03030303  .byte     [5] 3
                    03
3    T00000006      414243    .byte     "ABC"
4    T00000009      034142    .byte     3,"AB"
5    T0000000c      2202      .byte     'f'/3,'f'/'3'
6    T0000000e      81        .byte     -127
```

Line 1 places 81 in a byte at address 000000 of the text segment.

Line 2 places 3 (repeated 5 times) in five consecutive bytes starting at address 000001 in the text segment.

Line 3 places the ASCII values of "ABC" in three consecutive bytes starting at address 000006 in the text segment.

Line 4 places 3 in the byte at address 000009 in the text segment followed by the ASCII values of "AB" in two consecutive bytes.

Line 5 places the value of the expressions 'f'/3 and 'f'/'3' in consecutive bytes beginning at address 00000C in the text segment. The value of 'f'/3 (0x22) is first, followed by the value of 'f'/'3' (0x02).

Line 6 places 81 in a byte at address 00000E in the text segment.

6.3.3 .word

[*label*] *.word* ({[[*repetition-factor*]] *expression* | *string*}),,,

label is an optional label.

.word is the directive name.

[***repetition-factor***]
(optional) specifies the number of occurrences of the specified data word. It must be an expression which evaluates to a positive absolute value. If the ***repetition-factor*** is specified, it must be enclosed in "[ ]" brackets.

expression specifies the data word value. It must evaluate to an absolute value within the range of −32768 to 65535.

string specifies a string constant. If the string is not composed of an even multiple of two characters, it is null padded by the appropriate amount.

The **.word** directive generates one or more word length constants. The assembler places the constants in the instruction stream at the current address specified by the location counter. The assembler stores the least-significant byte at the lower address and the most-significant byte at the higher address.

If the expression is a relative type or an external, undefined type, the assembler generates a relocation entry for the operand. The linker uses the relocation entry to update the operand address at link time.

If the expression is of type code-label (see Section 6.6.3) then in architectures where the pc has implied bits, the expression is adjusted accordingly. In CR16, for example, it is divided by two. In CR32A there are no implicit bits.

If multiple constants are specified (e.g., ***repetition-factor*** is greater than one, or more than one ***expression*** is given), the constants are stored in consecutive words beginning at the current address.

When a string is specified as an operand of the **.word** directive, it is output as a byte string beginning at the lowest address and padded at the high address to an even multiple of two bytes if necessary.

Example

```
1    T0000000       0180      .word 32769
2    T0000002       34123412  .word [2] 0x1234
3    T0000006       41004142  .word 'A', "AB"
4    T000000a       0100      .word 0x41424344/0x41424344
5    T000000c       0180      .word -32767
6                              .code_label Func1
7    T000000e       0009      .word Func1
```

```
8    T000010        0012      .word Func2
9                             Func1:
10                            Func2:
```

Line 1 places the constant 32769 in a word at the address 000000 in the text segment.

Line 2 places the constant 0x1234 (repeated twice) in two consecutive words.

Line 3 places the word values of the character constant 'A' and the string "AB" (evaluated as integers) in two consecutive words.

Line 4 places the value of the expression 0x41424344/0x41424344 in a word at the address 00000A in the text segment.

Line 5 places 0x8001 (−32767) in a word at address 00000C in the text segment.

Line 6 places the address of function Func1, shifted by 2, at address 00..0e.

Line 7 places the address of function Func2 in a word at address 0010.

6.3.4 .double

[*label*] .double ({{[*repetition-factor*]] *expression* | *string*}),,

label is an optional label.

.double is the directive name.

[***repetition-factor*** (optional) specifies the number of occurrences of the specified double-word. It must be an expression which evaluates to a positive absolute value. If the ***repetition-factor*** is specified, it must be enclosed in "[ ]" brackets.

expression specifies the double-word value. It must evaluate to an absolute value within the range of -2^{31} to $2^{31} - 1$.

string specifies a string constant. If the string is not composed of an even multiple of four characters, it is null padded by the appropriate amount.

The **.double** directive generates one or more double-word constants. The assembler places the constants in the instruction stream at the current address specified by the location counter. The assembler places the bytes in ascending order, beginning with the least-significant byte at the lowest address.

If multiple constants are specified (e.g., ***repetition-factor*** is greater than one, or more than one ***expression*** is given), the constants are stored in consecutive double-words, beginning at the current address. When a string is specified as an operand of the **.double** directive, it is output as a byte string, beginning at the lowest address and padded at the high address to an even multiple of four bytes if necessary.

Example

```
1    T0000000     ffff0000.double 0x0000FFFF, 0xFFFF0000
                  0000ffff
2    T0000008     03000000.double [2] 3
     03000000
3    T0000010     41424300.double ''ABC''
4    T0000014     01000000.double 0x41424344/0x41424344
5    T0000018     70ffffff.double -144, 257
                  01010000
```

Line 1 places the constants 0x0000ffff and 0xffff0000 in two consecutive double-words.

Line 2 places the constant 3 (repeated twice) in two consecutive double-words.

Line 3 places the value of the string “ABC” in a double-word.

Line 4 places the value of the expression 0x41424344/0x41424344 in a double-word at address 00000014 in the text segment.

Line 5 places the value of the signed constants −144 and 257 in consecutive double-words.

6.3.5 .float

[*label*] .float ([[*repetition-factor*]] *expression*),,,

label is an optional label.

.float is the directive name.

[***repetition-factor***]
(optional) specifies the number of occurrences of the specified floating-point number. It must be an expression which evaluates to a positive absolute value. If the ***repetition-factor*** is specified, it must be enclosed in brackets. “ [ ]” .

expression specifies a single-precision floating-point constant (refer to Section 3.4.2). Strings are not permitted.

The **.float** directive generates one or more single-precision floating-point constants. The assembler places the constants in the instruction stream at the current address specified by the location counter. The assembler stores a single-precision floating-point constant in a double-word (32 bits).

If multiple constants are specified (e.g., ***repetition-factor*** is greater than one, or more than one ***expression*** is given), the constants are stored in consecutive double-words beginning at the current address.

Example

```
1    T0000000       4cdc3654       .float 3.14152E+12
2    T0000004       1ff47c3f       .float [2]0.9881
                    1ff47c3f
```

Line 1 places the floating-point constant 3.14152E+12 in a double-word at the current address.

Line 2 places the floating-point constant 0.9881 (repeated twice) into two consecutive double-words.

6.3.6 .long

[***label***] .long ([[***repetition-factor***]]***expression***),,,

label is an optional label.

.**long** is the directive name.

[***repetition-factor***]
(optional) specifies the number of occurrences of the specified floating-point number. It must be an expression which evaluates to a positive absolute value. If the ***repetition-factor*** is specified, it must be enclosed in "[ ]" brackets.

expression specifies a double-precision floating-point constant (refer to Section 3.4.2). Strings are not permitted.

The **.long** directive generates one, or more double-precision floating-point constants. The assembler places the constants in the instruction stream at the current address specified by the location counter. The assembler stores a double-precision floating-point constant in a quad-word (64 bits).

If multiple constants are specified (e.g., ***repetition-factor*** is greater than one, or more than one ***expression*** is given), the constants are stored in consecutive quad-words beginning at the current address.

Example

```
1    T0000000       00002078       .long 3.14152E+12
                    89db8642
2    T0000008       3695efe2       .long 6.12E-23, [3] 0.9881
```

```
1e7f523b
e6ae25e4
839eef3f
e6ae25e4
839eef3f
e6ae25e4
839eef3f
```

Line 1 places the floating-point constant 3.14152E+12 in a quad-word at the current address.

Line 2 places the floating-point constants 6.12E−23 and 0.9881 (repeated three times) in four consecutive quad-words.

6.3.7 .field

[*label*] .field ([*subfield-size*]*subfield-value*),,,

label is an optional label.

.field is the directive name.

[***subfield-size***]
(required) specifies the length in bits of the field being generated. It may be any expression which evaluates to a positive absolute value. No forward referencing of symbols is permitted. The ***subfield-size*** must be enclosed in "[ ]" brackets.

subfield-value(
required) specifies a field value. It may be any expression which evaluates to a non-negative absolute value. It must be within the range specified by the field size (e.g., 0 to 15 for a 4-bit field, 0 to 31 for a 5-bit field).

The **.field** directive generates one or more bit fields. The assembler places the field(s) in the instruction stream at the current address specified by the location counter. The directive provides no default values; thus, both ***subfield-size*** and ***subfield-value*** must be specified.

If the directive specifies more than one ***subfield-size/subfield-value*** pair, the values are placed in contiguous fields. If a field or a combination of fields do not extend to a byte boundary, the assembler zero-fills the remaining bits.

If multiple constants are specified, the ***subfield-size/subfield-value*** pairs must be separated by commas. See lines 2 and 3 in the following example.

Example

```
1    T0000000       08   .field [4] 8
2    T0000001       3f   .field [4] 15, [4] 3
3    T0000002       2143 .field [4] 1, [4] 2, [4] 3, [4] 4
```

Line 1 places 8 in a 4-bit field at address 0000000 in the text segment and zero-fills the four high-order bits.

Line 2 places 15 and 3 in two consecutive 4-bit fields at address 0000001 in the text segment.

Line 3 places 1, 2, 3, and 4 in four consecutive 4-bit fields. The fields occupy two bytes beginning at address 0000002 in the text segment.

6.4 STORAGE ALLOCATION DIRECTIVES

There are six storage allocation directives:

Directive	Function
`.blkb`	allocates byte storage
`.blkw`	allocates word storage
`.blkd`	allocates double-word storage
`.blkf`	allocates double-word(s) for floating-point storage
`.blkl`	allocates quad-word(s) for long floating-point storage
`.space`	allocates a block of storage

All storage allocation directives except **`.space`** have the following form:

`[label] directive [expression]`

The optional ***`expression`*** specifies the number of bytes, words, double-words, or quad-words to be allocated. It must evaluate to a non-negative absolute value. If the ***`expression`*** evaluates to zero, no storage is allocated. If no ***`expression`*** is specified, the default value is one. The ***`expression`*** may use symbolic values, but no forward symbol references are allowed.

When storage allocation directives occur in the text segment, the allocated bytes, words, double-words, or quad-words allocated are initialized to the **nop** instruction and appear in the program listing as generated code. When storage allocation directives occur in the data segment, the allocated bytes, words, double-words, or quad-words are initialized to zero and appear in the program listing as generated code. For all other segment types, the allocated space is uninitialized.

Sections 6.4.1 through 6.4.6 define the syntax of these directives.

6.4.1 .blkb

[***label***] .blkb[***expression***]

label is an optional label.

.blkb is the directive name.

expression specifies the number of bytes to be allocated. It must be an unsigned integer constant or an expression which evaluates to a non-negative absolute value. The default value is one.

The **.blkb** directive allocates zero or more consecutive bytes of memory for data storage. The bytes begin at the current location counter address.

Example

```
1                                           .data
2     D00000000      00                     .blkb 1
3     D00000001      00000000       AA:     .blkb 15
                     00000000
                     00000000
                     000000
4     D00000010      00000000               .blkb (.-AA)/3
                     00
5     D00000015      00                     .blkb
```

Line 2 allocates a single byte for data storage. The byte is located at address 00000000 in the data segment.

Line 3 allocates 15 consecutive bytes for data storage, beginning at address 00000001 in the data segment. The label AA is assigned the address of the first byte.

Line 4 allocates the number of bytes specified by the "(.—AA)/3" expression.
The expression evaluates to 5, i.e., (16 (data relative) — 1 (data relative)) = 15 (absolute), 15/3 = 5. Therefore, 5 bytes are allocated, beginning at address 00000010 data segment relative.

Line 5 allocates a single byte for storage.

6.4.2 .blkw

[***label***] .blkw [expression]

label is an optional label.

.blkw is the directive name.

expression specifies the number of words to be allocated. It must be an unsigned integer constant or an expression which evaluates to a non-negative absolute value.

The **.blkw** directive allocates zero or more consecutive words of memory for data storage. The words begin at the current location counter address.

Example

```
1                                                       .text
2     T00000000       a2a2                              .blkw 1
3     T00000002       a2a2a2a2        AA:               .blkw 15
                      a2a2a2a2
                      a2a2a2a2
                      a2a2a2a2
                      a2a2a2a2
                      a2a2a2a2
                      a2a2
4     T00000020       a2a2a2a2                          .blkw (.—AA)/3
                      a2a2a2a2
                      a2a2a2a2
                      a2a2a2a2
                      a2a2a2a2
5     T00000034       a2a2                              .blkw
```

Line 2 allocates one word for data storage at address 00000000 in the text segment.

Line 3 allocates 15 consecutive words for data storage, beginning at address 00000002 in the text segment. The label AA is assigned the address of the first word.

Line 4 allocates the number of words specified by the "(.—AA)/3" expression. The expression evaluates to 10, *i.e.*, (32 (text relative) — 2 (text segment relative)) = 30 (absolute), 30/3 = 10. Therefore, 10 words are allocated, beginning at address 00000020 in the text segment.

Line 5 allocates one word for storage.

6.4.3 .blkd

[***label***] .blkd [***expression***]

label is an optional label.

.blkd is the directive name.

expression specifies the number of double-words to be allocated. It must be an unsigned integer constant or an expression which evaluates to a non-negative absolute value.

The **.blkd** directive allocates zero or more consecutive double-words of memory for data storage. The double-words begin at the current location counter address.

Example

```
1                       .text
2    text_start:
3                       .dsect lo_text, text_start
4                       .blkd 1
5    AA:                .blkd 15
6                       .blkd (.-AA)/3
7                       .blkd
```

Line 4 allocates one double-word for data storage, overlaid onto address 000000 of the text segment.

Line 5 allocates 15 consecutive double-words for data storage, overlaid onto address 000004 of the text segment. The label AA is assigned the address of the first double-word.

Line 6 allocates the number of double-words specified by the "(.−AA)/3" expression. The expression evaluates to 20, i.e., (64 (text relative) − 4 (text relative)) = 60 (absolute), 60/3 = 20. Therefore, 20 double-words are allocated and overlaid onto address 000040 of the text segment.

Line 7 allocates a single double-word for storage.

6.4.4 .blkf

[***label***] .blkf [***expression***]

label is an optional label.

.blkf is the directive name.

expression specifies the number of double-words to be allocated. It must be an unsigned integer constant or an expression which evaluates to a non-negative absolute value.

The **.blkf** directive allocates zero or more consecutive double-words of memory for storage of single-precision floating-point (32-bit) numbers. The double-words begin at the current location counter address.

Example

```
1                       .udata
2                       .blkf 1
3           AA:         .blkf 15
4                       .blkf
```

Line 2 allocates one double-word for data storage at the address of the bss segment.

Line 3 allocates 15 consecutive double-words for data storage, beginning at the current address of the bss segment. The label AA is assigned the address of the first double-word.

Line 4 allocates one double-word for storage at the address of the bss segment.

6.4.5 .blkl

[*label*] .blkl [*expression*]

label is an optional label.

.blkl is the directive name.

expression specifies the number of quad-words to be allocated. It must be an unsigned integer constant or an expression which evaluates to a non-negative absolute value.

The **.blkl** directive allocates zero or more consecutive quad-words of memory for storage of double-precision floating-point (64-bit) numbers. The quad-words begin at the current location counter address.

Example

```
1                       .udata
2                       .blkl 1
3           AA:         .blkl 15
4                       .blkl
```

Line 2 allocates one quad-word for data storage at address 00000000 of the bss segment.

Line 3 allocates 15 consecutive quad-words for data storage, beginning at address 00000008 of the bss segment. The label AA is assigned the address of the first quad-word.

Line 4 allocates a single quad-word for storage at 00000128 of the bss segment.

6.4.6 .space

[***label***] .space [***expression***]

label is an optional label.

.space is the directive name.

expression specifies the number of bytes to be allocated. It must be an unsigned integer constant or an expression which evaluates to a non-negative absolute value.

The **.space** directive allocates a consecutive block of memory for data storage. The block begins at the current location counter address. The size in bytes of the storage block is specified by ***expression***.

Example

```
1                                                       .data
2     D00000000       00                                .space 1
3     D00000001       00000000        AA:               .space 15
                      00000000
                      00000000
                      000000
4     D00000010       00000000                          .space (.-AA)/3
      00
5     D00000015       00                                .space 1
```

Line 2 allocates one byte for data storage. The byte is located at address 00000000 in the data segment.

Line 3 allocates 15 consecutive bytes for data storage, beginning at address 00000001 in the data segment. The label AA is assigned the address of the first byte.

Line 4 allocates the number of bytes specified by the "(.—AA)/3" expression. The expression evaluates to 5, *i.e.*, (16 (data relative) — 1 (data relative)) = 15 (absolute), 15/3 = 5. Therefore, five bytes are allocated, beginning at address 00000010 data segment relative.

Line 5 allocates a single byte for storage.

6.5 LISTING CONTROL DIRECTIVES

The listing control directives control the format of the CompactRISC Assembler's program listing:

Directive	Function
`.title`	prints title at top of program listing
`.subtitle`	prints subtitle at top of program listing
`.nolist`	suppresses the printing of lines of source program to listing
`.list`	restores printing of lines of source program to listing
`.eject`	continues listing at top of next page
`.width`	sets width of listing page

Sections 6.5.1 through 6.5.6 describe the listing control directives in detail.

6.5.1 .title

[***label***] .title **"*string*"**

label is an optional label.

.title is the directive name.

string specifies the character string to be printed at the top of the listing page. The string (required) may consist of any combination of up to 126 letters, numbers, and text characters and must be enclosed in double-quotes.

The **.title** directive causes the assembler to print the specified ***string*** at the top of each new page of the program listing. The first **.title** directive affects the current listing page as well as all previous pages.

If a program contains more than one **.title** directive, the last **.title** directive to be specified before the page break affects subsequent pages. If a page other than the first page has no **.title** directive, it receives the title of the previous page.

If a program contains no **.title** directive, no title is printed.

No title is printed on the cross-reference page.

Example

```
.title John's Program
```

The preceding example causes the string “John’s Program” to be printed at the top of the current page of the program listing. If it is the only **.title** directive in the program, all pages have the same title.

6.5.2 .subtitle

[***label***] .subtitle **“*string*”**

label is an optional label.

.**subtitle** is the directive name.

string specifies the character string to be printed at the top of the listing page. The string (required) may consist of any combination of up to 126 letters, numbers, and text characters and must be enclosed in double-quotes.

The **.subtitle** directive causes the assembler to print the specified ***string*** at the top of each new page of the program listing. If a **.title** directive is also specified, the subtitle ***string*** appears below the title ***string***.

The first **.subtitle** directive affects the current listing page as well as all previous pages.

If a program contains more than one **.subtitle** directive, the last **.subtitle** directive to be specified before the page break affects subsequent pages. If a page other than the first page has no **.subtitle** directive, it receives the title of the previous page.

If a program contains no **.subtitle** directive, no title is printed.
No title is printed on the cross-reference page.

Example .subtitle “Written 7/7/81”

The preceding example causes the string “Written 7/7/81” to be printed at the top of the current page of the program listing. If it is the only **.subtitle** directive in the program, all pages have the same subtitle.

6.5.3 .nolist

[***label***] .nolist [***qualifier_list***]

label is an optional label.

.**nolist** is the directive name.

qualifier_list
macro listing qualifiers to be set off. Can be any combination of the qualifiers: ***mac_source***, ***mac_expansions*** and ***label***, as described in Section 7.14

The **.nolist** directive suppresses the printing of source program lines. All lines following the **.nolist** directive are assembled but are not printed to the program listing.

The **.nolist** directive does not affect the printing of error messages.

The **.nolist** directive may be inhibited by specifying a **.list** directive (see Section 6.5.4).

Example

```
.nolist
movd            r0, r1
addb            TEMP, r1
subb            r1, r0
.list
```

In the preceding example, the **.nolist** directive suppresses printing of the statement containing the **.nolist** directive and the following three lines of source. Printing is restored by the **.list** directive. Only the statement containing the **.list** directive is printed.

6.5.4 .list

[***label***] .list [***qualifier_list***]

label is an optional label.

.**list** is the directive name.

qualifier_list
macro listing qualifiers to be set off. Can be any combination of the qualifiers: ***mac_source***, ***mac_expansions*** and ***mac_directives***, as described in Section 7.14

The **.list** directive restores the printing of lines of the source program after suppression by a **.nolist** directive. All lines following the **.list** directive are printed to the program listing. The statement containing the **.list** directive is also printed to the program listing.

Example

```
            .nolist
            movd            r0, r1
            addb            TEMP, r1
            subb            r1, r0
            .list
NXT:        cmpb            r1,r0
```

In this example, the **.list** directive restores printing after the previous **.nolist** directive. Only the statement labelled NXT: and the **.list** statements are printed.

6.5.5 .eject

[***label***] .eject

label is an optional label.

.**eject** is the directive name.

The **.eject** directive causes the program listing to continue at the top of the next page. The statement containing the **.eject** directive is printed in the program listing.

Example .eject

This example causes the program listing to continue at the top of the next page. The statement containing the **.eject** directive is printed.

6.5.6 .width

[***label***] .width [***expression***]

label is an optional label.

.**width** is the directive name.

expression specifies page width in characters. It must be an unsigned integer constant or expression which evaluates to an absolute value within the range of 80 to 132.

The **.width** directive sets the width (in characters) of the program listing lines which follow the directive. (The first **.width** directive effects all preceding pages as well.) More than one **.width** directive is allowed, with each directive effective until the next or until the end of the file. If there is no **.width** directive, the width is 132 characters by default. The new-line character is included in the maximum width.

If the ***expression*** value is outside the specified range, an error message is generated.

Example .width MYPAGEWIDTH - 12

The preceding example sets the page width to the value of the expression MYPAGEWIDTH-12. The expression must evaluate to a number within the range 80 to 132.

6.6 LINKAGE CONTROL DIRECTIVES

The linkage control directives provide support for modular programming by allowing symbols and procedures to be exported from, or imported to, separately assembled modules. These directives are:

Directive	Function
.globl	declares external data symbols
.comm	declares external undefined data symbols
.code_label	declares that a symbol is in the code section

The **.globl** directive declares a symbol external, either for import or export, but does not define the symbol. The **.comm** directive is similar, except an associated size is specified. At link time, symbols declared with **.comm** are resolved and allocated in the bss segment. The .code_label directive declares symbols to be in the code section. This is important in architectures whose PC has implied bits.

Sections 6.6.1 through 6.6.3 describe the linkage control directives.

6.6.1 .globl

.global ***symbol***

.global is the directive name.

symbol is the name of a symbol. If more than one **symbol** is specified, the symbols must be separated by commas.

The **.globl** directive declares a symbol to be external, that is, a symbol intended to be used by multiple, separately assembled pieces of the same program. The **.globl** directive guarantees that a symbol table entry is generated in the object file, marked external. The linker uses these entries to resolve external symbol references at link time. Symbols declared with the **.globl** directive may or may not be defined within the current assembly. Defined symbols that are not declared to be external are assumed to be local symbols and may not be used to resolve undefined external references at link time. Undefined symbols are assumed to be external, with or without declaration, but it is good practice to declare all external symbols.

An alternate way to declare external symbols is to replace the colon of the label definition with a double colon (::).

Example

```
        .globl      FIRST,      SECOND
FIRST:
SECOND:
THIRD::
```

This example defines and exports three symbols: FIRST, SECOND, THIRD.

6.6.2 .comm

```
.comm symbol, expression
```

.comm is the directive name.

symbol is the name of a data symbol referenced, but not defined, in the current module.

expression specifies the number of bytes allocated for the symbol. It may be any expression which evaluates to a positive absolute value.

The **.comm** directive imports the specified symbol and assigns it an external undefined type. When the module is linked, the symbol is placed in the **.bss** section. If a symbol is in the **.comm** section of two files, the linker shares their areas.

Example

```
1                                       .comm     SYM1,16
2                                       .comm     SYM2,4
3    T00000000      14a8c000      movb      SYM1,r0
                    0000
4    T00000006      57a8c000      movd      SYM2,r1
                    0000
```

6.6.3 .code_label

```
.code_label symbol
```

.code is the directive name.

symbol is the name of a symbol. If more than one **symbol** is specified, the symbols must be separated by commas.

The .**code_label** directive qualifies a symbol as a label of code. In the CR16 architectures this kind of label needs special treatment in specific cases. Declaring a symbol as .**code_label** means that its value is shifted by 1 to the right whenever it is referenced as an immediate value. When it appears as an operand of a .**word** directive, it must also be shifted. See Sections 5.7 and 6.3.3.

Example

```
# CR16A
.code_label         _func
movw                $_func, r0
```

loads the address of **_func**, shifted right by 1, into **r0**.

Example

```
# CR16B
.code_label         _func
movw                $_func, (r1,r0)
```

loads the address of **_func**, shifted right by 1, into register pair **r1-r0**.

6.7 SEGMENT CONTROL DIRECTIVES

The segment control directives control the current segment type and the value of the assembler's location counter. These directives are:

Directive	Function
.dsect	sets the location counter to a user-defined segment
.text	sets the location counter to the text segment
.data	sets the location counter to the date segment
.bss	assigns space in the bss segment, updates the location counter
.udata	sets the location counter to the bss segment
.section	defines a section with attributes
.org	sets the location counter to specified value
.align	sets the location counter to specified offset
.ident	places the string argument in the **.comment** section of the object file

The segment control directives permit definition of program segments. A segment is a group of sequential statements whose addresses are all relative to the same base. Segments permit data or instructions to be processed as a unit and to be stored in a contiguous block within memory at run-time.

Sections 6.7.1 through 6.7.9 describe the syntax and operation of the segment control directives.

6.7.1 .dsect

.dsect ***symbol expression*** **[, *specifier*]**

.dsect is the directive name.

symbol specifies the name of the dummy section.

expression specifies the value and type of the location counter for the segment. The expression is required the first time a named *dsect* is invoked. Subsequent **.dsect** directives using the same name may omit the expression.

specifier is a plus sign (+) or a minus sign (—). ***Specifier*** indicates whether the location counter should be incremented or decremented.

The **.dsect** directive defines a named, user-defined (or dummy) segment. A dummy segment is used to define symbols which may be used in expressions or as instruction operands to access data. No code or initialized data may be generated in a ***dsect***.

The assembler assigns a location counter to the segment with the value and type specified by the expression. If the type of the expression is relative, for example text or data, the dummy segment may be thought of as an overlay of an existing memory segment. For example, a dummy segment might be used to define differing logical data structures that occupy the same storage space, as in a C union or a Pascal variant record.

An optional specifier may be used to indicate whether the location counter for the dummy segment increments or decrements. If the optional specifier is omitted, the value of ***expression*** determines whether the location counter increments or decrements. If the value of the expression is negative, the assembler decrements the location counter. If the value of the expression is positive or zero, the assembler increments the location counter. In either case, labels are assigned the lowest byte address of the following statement. That is, the location counter is post-incremented and pre-decremented.

Example

```
                .dsect      DATE_REC, 0
MONTH:          .blkb
DAY:            .blkb
YEAR            .blkw
```

This example defines three absolute symbols in a dummy segment named **DATE_REC**. The symbols have the absolute values of 0, 1, and 2. The symbols can be used as offsets into any block of memory. In the example below, **r0** contains the address of a block of memory for storing the data. The instructions in the example zero-fill the month, day, and year fields.

```
                .udata
DATE:           .blkb           4
                .text
                movd            $DATE, r0
                movw            $0, r1
                storb           r1, MONTH(r0)
                storb           r1, DAY(r0)
                storw           r1, YEAR(r0)
```

6.7.2 .text

.text

.text is the directive name.

The **.text** directive indicates the beginning of a program text segment or code segment. The assembler assigns the current location counter the next available text segment address. Subsequent storage allocation, data generation, or program statements generate code and constant data that are placed in the **.text** section of the object file. Storage allocated in the text segment is filled with nop instructions. The location counter is incremented after every assignment, storage allocation, or code generation.

Symbols defined in the text segment are of type text. The assembler uses the Program Counter (PC) Relative addressing mode for all symbols or expressions of type text. When the text segment is loaded into memory, it contains a module's instructions and constant data and is, therefore, protected for read-only access.

Example

```
.text
```

In the preceding example, the location counter is set to text segment type. The offset is set to the next available offset. Instructions and data directives that follow the ***.text*** directive generate code in the **.text** section of the object file.

6.7.3 .data

.data

.data is the directive name.

The **.data** directive indicates the beginning of an initialized data segment. An initialized data segment contains writable data or program code and are placed in the **.data** section of the object file. When the data segment is loaded into memory, it is protected for read-write access.

The **.data** directive sets the location counter to the next available data segment address. The location counter is incremented after every data assignment or code generation. Symbols defined in the data segment are of type data. The assembler uses the Absolute addressing mode for all symbols or expressions of type data.

Example `.data`

In the preceding example, the location counter is set to the data segment. The offset is set to the next available data segment address. Subsequent data directives, or instructions, are output to the **.data** section of the object file.

6.7.4 .bss

`.bss` ***symbol***, ***expression1***, ***expression2***

.bss is the directive name.

symbol is a symbol name.

expression1 specifies the symbol size.

expression2 specifies an alignment value for the bss location counter. The alignment value may not be zero.

The **.bss** directive defines a symbol in the bss or the uninitialized data segment. There is no code or data in the object file associated with the bss segment. The **.bss** directive is a shorthand way to align the location counter associated with the bss segment, define a symbol, and allocate the appropriate number of bytes of storage space. It does not change the current location counter to the bss segment. Use **.udata**, Section 6.7.5, to change the current location counter.

The **.bss** directive performs the following actions:

- aligns the bss location counter to a multiple of ***expression2***. The value of the location counter is incremented if necessary.
- defines the specified symbol. The symbol is assigned the current value of the bss segment location counter and type bss. The assembler uses the Absolute addressing mode to reference symbols or expressions of type bss.
- adds the number of bytes specified by ***expression1*** to the bss location counter.

Example `.bss name_str, 25, 4`

In the preceding example, the bss segment location counter is aligned to the next multiple of four bytes, incrementing if necessary. The symbol ***name_str*** is defined and assigned the value of the bss location counter. The bss segment location counter is incremented by 25.

Note If you want the alignment to hold after link time, the **bss** section must be aligned to the lowest common multiplier of all the alignment values it contains.

6.7.5 .udata

.udata

.udata is the directive name.

The **.udata** directive indicates the beginning of a bss or uninitialized data segment. It is used to define symbols and allocate storage space. As with dummy sections, no code or data is generated in the object file. However, storage space is accumulated. The total accumulated size of the segment is recorded in the COFF header and the **.bss** section header of the object file. Memory is allocated for the total size of the bss segment at load time.
.udata sets the location counter to the next available bss segment address. Symbols defined in the bss (udata) segment are of type bss. The assembler uses the Absolute addressing mode for all symbols or expressions of type bss.

Example

```
.udata
```

In the preceding example, the location counter is set to type bss. The offset is set to the next available offset.

6.7.6 .section

.section ***section_name*** **, string** or

.section ***section_name***

section_name is any legal identifier, only eight significant characters

string is a quoted string consisting of any combination of the following letters
b-> STYP_BSS
c-> STYP_COPY
i-> STYP_INFO
d-> STYP_DSECT
x-> STYP_TEXT
n-> STYP_NOLOAD

o-> STYP_OVER
l-> STYP_LIB
w-> STYP_DATA

The **.section** directive allows the assembly programmer to define a section with attributes; refer to the *CompactRISC Toolset - Object Tools Reference Manual* for a description of section attributes. ***Section_name*** is the name of the section, and each character in ***string*** represents an attribute. If ***string*** is not present, the section has no attributes. Symbols declared within a section belong to the particular section. A section is active until the next **.section**, **.text**, **.data**, or **.udata** directive. A maximum of 24 sections are allowed including **.text**, **.data**, **.bss,** and **.comment**. The **.comment** section is optional; therefore there can only be 20 - 21, user-defined sections.

Example

```
.section        .init,"x"
istart:
```

This example declares a section called **.init**, whose section attribute is **STYP_TEXT**. The label "istart" belongs to the **.init** section.

Note The compiler uses this directive to define special sections which allow more efficient allocation of data.

6.7.7 .org

`.org` ***expression***

.org is the directive name.

expression specifies the new value of the location counter. The expression must evaluate to type absolute or the type of the current location counter.

The **.org** directive changes the value of the current location counter within a segment. It sets the location counter to the value specified by ***expression***. The type of the expression should be compatible with that of the current location counter (i.e., it should refer to the location counter or to a label that is defined in the current segment). It can also be an absolute expression (e.g., a constant), however in this case the assembler generates a warning message, and sets the location counter to the value specified by ***expression*** + the starting location of the current segment.

If the current segment is an object file segment, i.e., one of text or data, then the value of the expression must be greater than, or equal to, the current location counter (i.e., backstepping is not permitted). Furthermore, for object file segments, the CompactRISC Assembler fills the bytes between the current and the new location with alignment values as filled for the **.align** directive. The added bytes are included in the program listing.

Example

```
1                       .set      NUM_CHNKS, 10
2                       .set      CHNK_SIZE, 4096
3
4                       .udata
5    B00000000 c_ptr:   .blkd           10
6    B00000028 pool:    .org      pool + (NUM_CHNKS *
                                   (CHNK_SIZE)
7    B0000a028 mark:    .blkd
```

This example uses the **.org** directive to leave a large area of memory available in the bss segment.

6.7.8 .align

.align ***expression1*** [,***expression2***]

.**align** is the directive name.

expression1 specifies the basis of a new location counter value. It must evaluate to a positive absolute value. No forward symbol references are permitted.

expression2 specifies the offset of the new location counter value. It must evaluate to a non-negative absolute value and must be less than the value of ***expression1***. Default value is zero. No forward symbol references are permitted.

The .**align** directive sets the location counter to a new value without changing the current type. The new value is the sum of a multiple of the basis, ***expression1***, and the offset, ***expression2***. The new value is always equal to, or greater than, the current location counter and satisfies the following equation:

new_value MOD ***expression1 = expression2***

The new value is the multiple of the basis that is greater than, or equal to, the current location counter. For example, if ***expression1*** is 6 and the current location counter is 20, then the new value is 24 (*i.e.*, 4*6). The default value of ***expression2*** is zero.

If both ***expression1*** and ***expression2*** are specified, the new value is the sum of the multiple of the basis and the offset. For example, if ***expression1*** is 4, ***expression2*** is 3, and the current location counter is 22, then the new value is 27 (i.e., 6*4+3).

If the **.align** directive is used in the text section, it is filled with 2-byte instructions that are equivalent to NOPs.

All other alignments are filled with combinations of the above.

If the **.align** directive is used in a data segment, the assembler zero-fills all bytes between the current location and the specified address, and includes up to 128 bytes of the zero-filled bytes in the program listing.

Example

```
1                                               .data
2    D00000000      00        FIRST:            .blkb
3    D00000001      000000                      .align 4
4    D00000004      00        SECOND:           .blkb
5    D00000005      00                          .align 4, 2
6    D00000006      00        THIRD:            .blkb
```

The preceding example contains two **.align** directives (lines 3 and 5).

In line 3, the directive sets the location counter to a multiple of 4. The current location counter is D00000001 (data segment), so the new location counter is D00000004 (i.e., 1*4).

In line 5, the directive sets the location counter to a multiple of 4 plus 2. If the current location counter is 5, then the new location counter is 6 (i.e., 1*4 + 2).

Example

```
1                                _mail:
2    T00000000      a2                          nop
3                                LABEL:
4    T00000001      d439                        .align 3
5    T00000003      0a00                        .word 10
6    T00000005      d8a100                      .align 4
7    T00000008      01                          .byte 1
8    T00000009      0a00                        .word 10
9    T0000000b      a2                          .align 2
10   T0000000c      1200                        ret 0
```

The preceding example contains three **.align** directives (lines 4, 6, and 9).

In line 4, the directive sets the location counter to a multiple of 3. The current location counter is T00000001 (text segment), so the new location counter is T00000003 (i.e., 1*3). The 2-byte filler "movb r7,r7" denoted by the opcode d439 (low bytes first) is used.

In line 6, the directive sets the location counter to a multiple of 4. The current location counter is T00000005, so the new location counter is T00000008 (i.e., 2*4). The 3-byte filler "orb $0,r7" denoted by the opcode d8a100 is used.

In line 9, the directive sets the location counter to a multiple of 2. The current location counter is T0000000b, so the new location counter is T0000000c (i.e., 6*2). The single byte filler "nop" denoted by the opcode a2 is used in this case.

6.7.9 .ident

`.ident` ***`string`***

`string` is a quoted string.

The **`.ident`** directive takes its string argument and places it in the **`.comment`** section of the object file. This directive may be used more than once. The **`.comment`** section is given the section attribute of **`STYP_INFO`**. The linker combines all **`.comment`** sections at link time.

Example

```
1                                       .text
2                                       .ident "This is .ident"
3    T00000000      a2a2a2a2            .space 1
                    a2a2a2a2
                    a2a2
4                                       .ident "Another .ident"
```

In this program, the strings "This is .ident" and "Another .ident" are placed in the **`.comment`** section of the object file.

6.8 FILENAME DIRECTIVE

The filename symbol directive specifies the name of the source file:

Directive	Function
`.file`	specifies the source filename

6.8.1 .file

`.file "`***`symbol`*** `"`

`.file` is the directive name.

"***symbol***" specifies source filename for the current assembly. Must be enclosed in double-quotes.

The **.file** directive specifies the name of the source file currently being assembled. The CompactRISC Assembler records the filename in the object file as an auxiliary symbol table entry of the special symbol .file. Only one **.file** directive per source file is allowed. It may appear anywhere in the file. If no **.file** directive is specified, the filename is the input source filename.

If more than one **.file** directive is specified, the first specified filename is taken, and a warning message is issued for the rest of them.

The **.file** directive is used by compilers to associate the name of a high-level language source file with the object file produced by the CompactRISC Assembler.

Example

```
.file "stress.c"
```

This example defines the symbol **stress.c** as the name of the source file associated with the current assembly. When using the debugger, the ***symbol*** must be the same as the filename since the debugger uses this as the name of the source file.

6.9 SYMBOL TABLE ENTRY DEFINITION DIRECTIVES

The symbol table entry definition directives specify symbolic information which the CompactRISC Assembler records in the object file. The directives provide a means to record a variety of information useful to symbolic debuggers. Symbol table entry directives do not affect the execution of an assembly language program.

The basic symbol table entry directives are **.def** and **.endef**. They mark the start and the end of a symbol definition. Between these, various directives may be used to assign attributes to the symbol, for example, its size, value, and type or its location in the source file.

Each **.def** begins to define a new symbol table entry. Therefore, all information to be recorded about a single symbol must be included between the **.def** directive and the matching **.endef** directive.

Symbol table entry definitions may not be nested.

The symbol table entry definition directives are as follows:

Directive	Function
`.def`	begins symbol table entry definition
`.dim`	defines the dimensions of an array
`.line`	specifies a source line number
`.scl`	specifies the symbol's storage classification
`.size`	specifies the symbol's storage size
`.tag`	specifies the tag name associated with a type
`.type`	specifies the symbol's type
`.val`	specifies the symbol's value
`.endef`	terminates the symbol table entry definition

Sections 6.9.1 through 6.9.9 describe the symbol table entry directives in detail. It is important to fully understand the Common Object File Format (COFF) symbol table requirements before attempting to use these directives. For a complete specification of COFF requirements refer to the *CompactRISC Toolset - Object Tools Reference Manual*. For useful constant definitions see the include files:

File	Contents
`syms.h`	Symbol table entry definition, auxiliary entry definition, type and derived type values
`storclass.h`	Storage class values

6.9.1 .def

`.def "`***`symbol`*`"`

`.def` is the directive name.

`symbol` is a symbol name. It consists of a series of characters which may be letters, numbers, period (.), or underscore (_). The first character must not be a number.

The **`.def`** directive causes the CompactRISC Assembler to begin the definition of a Common Object File Format (COFF) symbol table entry for the specified symbol. The CompactRISC Assembler creates the new symbol table entry and enters the symbol name. The assembler does not check the COFF validity of the given values for symbol table entries definition.

Example

```
.def            _n_ptr
                .val            _n_ptr
                .scl            2
                .type           2 | (1 << 4)
.endef
.globl          _n_ptr
.comm           _n_ptr,4
```

This example is a symbolic definition associated with the C declaration:

```
char *n_ptr;
```

The **.def** directive starts the definition. The symbol table entry is assigned the value **_n_ptr**, a storage class of external (**C_EXT**) represented by the value 2, a base type of character (**T_CHAR**) represented by the value 2, and a derived type of pointer (**DT_PTR**) represented by the value 1. The **.endef** directive ends the definition. For more information about the structure of a COFF symbol table entry, the meaning of various fields, and the values each may contain, refer to the *CompactRISC Toolset - Object Tools Reference Manual*.

6.9.2 .dim

`.dim` ***expression***

.dim is the directive name.

expression specifies the size of one dimension of an array

The **.dim** directive defines the dimensions of an array. Each argument ***expression*** specifies the number of elements in one array dimension. The symbol table entry format allows the specification of up to four array dimensions.

The CompactRISC Assembler enters the specified expressions into the array dimension field of the auxiliary symbol table entry for the symbol that is being defined. If no auxiliary entry exists, the CompactRISC Assembler creates one.

Example

```
.dim 5,10
```

This example is a portion of the symbolic definition for a two-dimensional array. Dimension one is 5, dimension two is 10.

6.9.3 .line

`.line` ***expression***

.line is the directive name.

expression is the source file line number of the symbol declaration.

The **.line** directive specifies the source file line number on which a symbol has been declared. The CompactRISC Assembler enters the specified value, ***expression***, into the line number field of the auxiliary symbol table entry for the symbol that is being defined. The assembler generates an auxiliary entry if one does not exist.

The **.line** directive should be used when the symbol being defined is a block symbol. Block symbols include the special symbols **.bf** and **.ef** which define the beginning and ending of functions, the special symbols **.bb** and **.eb** which define the beginning and ending of blocks, and all symbols defined within a block. The **.line** directive should be used only where the Common Object File Format symbol table entry specification requires and accepts a line number. For additional information, refer to the *CompactRISC Toolset - Object Tools Reference Manual*.

Example

```
.line 25
```

This example is part of the definition of a block symbol declared on source line number 25.

6.9.4 .scl

`.scl` ***expression***

.scl is the directive name.

expression is the value of a storage classification as defined in the *CompactRISC Toolset - Object Tools Reference Manual*.

The **.scl** directive assigns a storage class value to the symbol definition. The storage class of a symbol affects the interpretation of the "value" field of the entry. Storage classes are as follows:

C_AUTO automatic variable, whose value is a stack offset.

C_EXT external symbol, whose value is a relocatable address.

C_STAT C style static or local variable, whose value is a relocatable address.

C_REG register variable, whose value is the number of the register. For example, if the register is r0 the register number is 0.

C_LABEL an assembly language label, whose value is a relocatable address.

C_MOS	member of a structure, whose value is the offset of the field from the start of the structure.
C_ARG	function argument, whose value is a stack offset.
C_STRTAG	structure tag (name), whose value is 0.
C_MOU	member of a union, whose value is the offset of the field from the start of the union.
C_UNTAG	union tag (name), whose value is 0.
C_TPDEF	type definition, whose value is 0.
C_ENTAG	enumeration tag (name), whose value is 0.
C_MOE	member of an enumeration, whose value is the enumeration number.
C_REGPARM	register parameter, whose value is the number of the register.
C_FIELD	bit field, whose value is the bit displacement.
C_BLOCK	beginning or end of block, whose value is a relocatable address.
C_FCN	beginning or end of a function, whose value is a relocatable address.
C_EOS	end of a structure, whose value is the structure size.
C_FILE	filename entry, whose value is the symbol table index of the next **.file** symbol or the beginning of the global symbols if there are no more **.file** symbols.
C_ALIAS	duplicate tag, whose value is the symbol table index of the tag definition.

Example `.scl 2`

This example specifies a storage classification of **C_EXT** (external), represented by the value 2.

6.9.5 .size

`.size` ***expression***

.size is the directive name.

expression specifies the size of a structured variable.

The **.size** directive specifies the total size of a structured type, array, or enumerated type. The CompactRISC Assembler enters the specified value into the size-field of the auxiliary symbol table entry for the symbol being defined. If no auxiliary entry exists, the Assembler generates one. For example, the C declaration:

```
char name_list[20] [200];
```

generates the following symbol specification:

```
1    .def _          name_list
2    .val _          name_list
3    .scl            2
4    .type           0362
5    .dim            20,200
6    .size           4000
7    .endef
8    .globl          _name_list
10   .comm           _name_list,4000
```

The storage size specified by the **.size** directive in line 6 is 4000 bytes (*20*200*sizeof(char*)), where the size of a character is one byte.

Example

```
.size 200
```

This example specifies a symbol's storage size as 200 bytes. The Assembler enters the value 200 into the size field of the auxiliary symbol table entry for the symbol that is being defined

6.9.6 .tag

.tag ***symbol***

.tag is the directive name.

symbol is a symbol. The symbol is the tag name of a data structure definition, for example, a C struct or union.

The **.tag** directive associates the tag name of a data structure with a symbol. The CompactRISC Assembler enters the symbol table index of the tag name into the tag index field of the auxiliary entry for the symbol that is being defined. If no auxiliary entry exists, the CompactRISC Assembler generates one.

Example

```
.def      _coord
          .scl 10; .type 010; .size 12; .endef
.def      _a
          .val 0; .scl 8; .type 04; .endef
.def      _b
          .val 4; .scl 8; .type 04; .endef
.def      _c
          .val 8; .scl 8; .type 04; .endef
.def      .eos
          .val 12; .scl 102; .tag _coord; .size 12; .endef
.def      _bar
          .val _bar; .scl 2; .type 010; .tag _coord; .size 12;
```

```
        .endef
        .globl _bar
        .comm _bar,12
```

This example defines the symbols associated with the C declarations:

```
struct          coord {
                int a;
                int b;
                int c;
};
struct          coord bar;
```

The special symbol **.eos** (end of structure) uses the **.tag** directive to point back to the definition of the structure coord.

The bar symbol, which is of type ***struct coord***, also uses the **.tag** directive to point to the entry for coord.

6.9.7 .type

`.type` ***expression***

.type is the directive name.

expression specifies the type of a symbol .

The **.type** directive specifies type information associated with the symbol that is being defined. The CompactRISC Assembler enters the ***expression*** into the type field of the main symbol table entry for the symbol that is being defined.

The type field consists of sixteen bits, of which the low-order four contain the base type. The remaining bits contain derived types, each of which is specified in a two-bit field. For definition of types and derived types see the *CompactRISC Toolset - Object Tools Reference Manual*.

Examples

```
1.      .type       (2 | (2 << 4)) | 1 << 6
2.      .type       4
```

The first example is a type definition associated with the C declaration:

```
char *fn();
```

The base type is **T_CHAR** (type character) represented by the value 2. The first derived type is **DT_FCN** (function) represented by the value 2. The second derived type is **DT_PTR** (pointer) represented by the value 1. The entire type field is interpreted as a pointer to a function that returns a character.

The second example is associated with the C declaration:

```
int flag;
```

The **.type** directive specifies the type **T_INT** (integer) represented by the value 4.

6.9.8 .val

.val ***expression***

.val is the directive name.

expression specifies the value of the symbol.

The **.val** directive specifies the value field of the main symbol table entry for the symbol that is being defined.

Example

```
.val _flag
```

This example sets the value field of the symbol table entry to the address of the symbol **_flag**.

6.9.9 .endef

.endef

.endef is the directive name.

The **.endef** directive causes the CompactRISC Assembler to end the definition of a Common Object File Format (COFF) symbol table entry for the specified symbol. The CompactRISC Assembler adds the new symbol table entry to the symbol table. The CompactRISC Assembler generates an auxiliary entry if the symbol specifications require one and fills in any symbol table index fields as necessary.

Example

```
.def            _flag
                .val      _flag
                .scl      2
                .type     4
.endef
```

This example is a symbolic definition associated with the C declaration:

```
int flag;
```

The **.endef** directive ends the definition.

6.10 LINE NUMBER TABLE CONTROL DIRECTIVE

Each section in the object file may have an associated line-number table, for the purpose of source-level debugging support. The line-number table maps source file line numbers to addresses within the section. Each line number table entry is either a function entry or a line number entry. Function entries record the symbol table index for the function. Line number entries record a line number offset from the start of the function and an associated physical address.

Function entries are generated automatically by the assembler when a function is defined, refer to Section 6.9. Line number table entries are created with the **.ln** directive.

Directive	Function
.ln	specifies a line number entry

6.10.1 .ln

.ln ***expression1*** [***expression1***]

.ln is the directive name.

expression1 specifies the source file line offset from the beginning of a function.

expression2 specifies an associated memory address. This value defaults to the current location.

This directive is used to equate higher level source code line numbers to assembly code, normally generated by compilers. ***Expression1*** must yield a value of absolute type that gives a line number in the source code. ***Expression2*** if present, must have a value of type TEXT, DATA, or BSS that gives the address within the section where the line number occurs. If the second operand is missing, the value of the current location counter is used as the address of the line number.

Example

```
.ln 1
```

This example defines a line number entry for the first line of a function. The associated memory address is the value of the current location counter.

6.11 MACRO-ASSEMBLER DIRECTIVES

The macro-assembler directives provide the macro and conditional assembly support. They enable the definition and usage of macros, and allow for the inclusion or deletion of optional assembly statements. Other macro-assembler directives help minimize programming errors and speed the development process. For more details see Chapter 7.

The macro-assembler directives are as follows:

Directive	Function
`.macro`	begins a macro-procedure definition
`.endm`	ends a macro-procedure definition
`.if`	begins a conditional macro-assembler statement
`.elsif`	begins an elsif close for the conditional macro-assembler statement
`.else`	begins an else close for the conditional macro-assembler statement
`.endif`	ends a conditional macro-assembler statement
`.repeat/.irp`	begins a macro repetitive block
`.endr`	ends a macro repetitive block
`.exit`	terminates processing of the current repetitive block
`.macro_on`	enables macro-procedure expansions
`.macro_off`	disables macro-procedure expansions
`.include`	includes another file
`.mwarning`	generates an assembler warning message
`.merror`	generates an assembler error message

6.11.1 .macro

`.macro` ***`macro-name`*** **`[`** ***`formal-arg`*** **`[ , `** ***`formal-arg`*** **`] ... ]`**

`macro-name` is the macro-procedure name. It may be any legal assembler symbol.

`formal-arg` is a macro-variable defining a formal argument.

The **`.macro`** directive begins the macro-procedure definition. The macro-procedure associates a macro name with a sequence of statements which follow the **`.macro`** directive, up to the **`.endif`** directive.

Example `.macro clear_array size, base_reg`

Defines a macro procedure named ***clear_array*** with two formal arguments ***size*** and ***base_reg***.

6.11.2 .endm

`.endm [ ` ***macro_name*** ` ]`

.endm ends the macro-procedure definition.

The **.endm** directive marks the end of the macro-procedure definition. ***macro_name*** is an optional specification for the name of the macro to be ended.

Example

```
.macro    clear_arraysize, base-reg# defines clear_array
                                   # macro statements
.
.endm     clear_array              # ends the definition
                                   # of clear_array
```

6.11.3 .if

`.if` ***if_condition***

.if is an arithmetic macro-expression.

.The **.if** directive begins a conditional macro assembler statement. ***if_condition*** is a condition to be tested during macro processing phase. If found to be true, the statements following it (until a corresponding **.elseif, .else** or **.endif** directive) are processed and expanded by the macro processor.

Example

```
.if {reg_num} = 6
     movw $5, r{reg_num}
.elsif {reg_num} = 4
     movw $3, r{reg_num}
.else if {reg_num} = 0
     movw $1, r{reg_num}
.endif
```

If ***reg_num*** holds the value **6** this is expanded to:

```
     movw $5, r6
```

if ***reg_num*** holds the value **4** this is expanded to:

```
     movw $3, r4
```

and if ***reg_num*** holds the value **0** this is expanded to:

```
movw $1, r0
```

6.11.4 .elsif

.eisif ***elsif_condition***

.elsif_conditional_body
consists of valid assembly language statements, directives, macro-procedure calls and macro-assembler directives, repetitive blocks and macro-procedure definitions.

.elsif_condition
is an arithmetic macro-expression.

If, in a conditional block, ***if_condition*** is found to be false, the arguments of ***elsif_condition*** are evaluated until one is found to be true. If ***elsif_condition*** is found to be true, the corresponding ***elsif_conditional_body*** statements, following the ***elsif_condition***, are processed.

6.11.5 .else

.else ***else_condition***

.else_conditional_body
consists of valid assembly language statements, directives, macro-procedure calls and macro-assembler directives, repetitive blocks and macro-procedure definitions.

In a conditional block, if the previously specified ***if_condition*** or ***elsif_condition*** is found to be false, then the ***else_conditional_body*** statements (following the ***elsif_condition***) are processed.

6.11.6 .endif

.endif

Ends an **.if** conditional macro-assembler statement.

6.11.7 .repeat

.repeat [*iteration_count* [, *iteration_var*]]

. ***iteration_count***
specifies the number of iterations.

. ***iteration_var***
is a macro-variable name used as an iteration index.

The **.repeat** directive begins a macro repetitive block, which ends with a **.endr** directive. The number of repetitions is determined by the ***iteration_count*** argument. Repetitive blocks may appear inside a macro-procedure definition, in conditional blocks, and may be nested without limit.

If given, the ***iteration_var*** argument holds a string representing the current iteration number for each iteration. After the repetitive block has been processed, it holds the ***iteration_count*** value. If the ***iteration_count*** argument is evaluated as a negative or zero value, the statements in the block are read textually without being processed until an **.endr** directive is reached. If the ***iteration_count*** argument is not given, then the repetitive block is processed repeatedly until an **.exit** directive is processed (Section 6.11.10).

Example

```
.repeat         8, i
 movd           $0, r{{i} - 1}
.endr
```

(For CR32A) generates code that clears **r0** through **r7.**

6.11.8 .irp

.irp ***iteration_var*** , ***iteration_list***

. ***iteration_var***
is a macro-variable name to be used as an iteration variable.

. ***iteration_list***
is a macro-list

The **.irp** directive begins a special macro repetitive block, which ends with a **.endr** directive. For each element in the ***iteration_list*** argument, the macro-processor assigns its string value to ***iteration_var***, and processes the code between the **.irp** statement and the corresponding **.endr** statement. If the ***iteration_list*** argument is an empty macro-list, the statements in the block are read textually without being processed. After the repetitive block has been processed, ***iteration_var*** contains the last element of ***iteration_list***.

Example

```
.irp reg, [r0,r1,r2,r3,r4,r5,r6,r7]
     movd                $0,{reg}
.endr
```

(For CR32A) generates code that clears registers **r0** through **r7**.

6.11.9 .endr

```
.endr
```

The **.endr** directive ends a macro repetitive block.

6.11.10 .exit

.exit

Terminates the processing of the current repetitive block that begins with either a **.repeat** or **.irp** directive. Statements following this directive are read textually without being processed, until an **.endr** statement is encountered.

```
x:=1
.repeat
     .if {x} > 30
     .exit
     .endif
     .byte {x}
     x:={{x}*2}
.endr
```

generates the code

```
.byte 1
.byte 2
.byte 4
.byte 8
.byte 16
```

6.11.11 .macro_on and .macro_off

The **.macro_on** and **.macro_off** directives enable and disable macro-procedure expansions, respectively, in selective parts of the source text.

Example

```
.macro              addw op1,op
     br count_additions
.macro_off
```

```
     addw           {op1},{op2}
.macro_on
.endm
```

the following macro-procedure call:

```
     addw r1,r2
```

generates:

```
     br count_additions
     addw r1,r2
```

6.11.12 .include

`.include` ***`includled_file`***

`included_file`
is an existing file name.

The **`.include`** directive allows for the inclusion of text from another file as part of the file being assembled.

Example

```
.include filehdr.h
```

If the ***`included_file`*** does not contain the full directory path of the file to be included, the assembler searches for it in either the current directory, or in a directory specified with the **`-MI`** invocation option (macro include directory).

6.11.13 .mwarning

`.mwarning` ***`warning_message`***

`.mwarning` is the directive name.

The **`.mwarning`** directive generates an assembler warning message.

Example

```
xx:= 222
.mwarning current value of "xx" is : {xx}.
```

`.mwarning` is used to write the current value of macro-variable **`xx`** to the listing output. The assembler issues the following warning message:

```
Assembler (Macro-Processor): "filename.s" , line 2 , WARNING:
current value of "xx" is : 222
```

6.11.14 .merror

.merror *error_message*

The directive **.merror** generates an assembler error message.

Example

```
.merror Wrong value used for addr "address"
```

The assembler issues the following error message, and eliminates further assembly passes:

```
Assembler (Macro-Processor) Error:
"filename.s", line 1, statement is ==> .merror "err"
Wrong value used for addr "address"<==
ERROR: Wrong value used for addr "address"
```

Chapter 7
MACRO AND CONDITIONAL ASSEMBLER

7.1 INTRODUCTION

The CompactRISC Macro-assembler makes writing assembly programs easier. It eliminates the need to rewrite similar assembly source code repeatedly, and simplifies program documentation. The conditional assembler feature allows for the inclusion or deletion of optional assembly statements. Other macro-assembler features help minimize programming errors and speed the development process.

The macro-assembler is automatically invoked by the assembler.

7.1.1 Overview of the Major Macro-Assembler Features

The CompactRISC macro-assembler supports the following features:

- **Macro-procedures (“macros”)**

A macro-procedure is equivalent to the common term “macro”. For example, for the following macro-procedure:

Example

```
.macro move_bytes source_add,dest_add,length
     .repeat {length},i
# next repeat iteration
          addw $-4,sp
          storw r1,0(sp)
          loadb {source_add}+{i},r1
          storb r1,{dest_add}+{i}
          loadw 0(sp),r1
          addw $4,sp
     .endr
.endm
```

The macro-procedure call:

```
move_bytes aa,6(r13),3
generates the code:
# next repeat iteration
     addw $-4,sp
     storw r1,0(sp)
     loadb aa+1,r1
     storb r1,6(r13)+1
```

```
     loadw 0(sp),r1
     addw $4,sp
# next repeat iteration
     addw $-4,sp
     storw r1,0(sp)
     loadb aa+2,r1
     storb r1,6(r13)+2
     loadw 0(sp),r1
     addw $4,sp
# next repeat iteration
     addw $-4,sp
     storw r1,0(sp)
     loadb aa+3,r1
     storb r1,6(r13)+3
     loadw 0(sp),r1
     addw $4,sp
```

- **Conditional Code Generation**

Code may be generated according to conditions tested in the macro-assembly phase. For example the sequence:

Example

```
.macro excp num
     .if {STR_EQ[{OP_TYPE[{num}]}, EXPR]}
          .word {CNV_HEX[{18145 |{{num}*2}}]}
     .else
          .macro_off
               excp {num}
          .macro_on
     .endif
.endm
```

generates the encoding for all exceptions in CR32A (including undefined). This feature is fully described in Section 7.8.1.

- **Macro Variables**

String values may be assigned to macro-variables. These variables may be later utilized in place of the string value.

Example

```
base_reg:= r0
movqd 0, 0({base_reg})
is equivalent to:
movqd 0, 0(r0)
```

This feature is fully described in Section 7.9.1.

- **Repetitive Code Generation**

This feature allows for the easy repetition of sequences of statements, by specifying either

- the number of repetitions:

Example

```
.repeat 3,index
.align 4
.word {index}
.endr
```

which is equivalent to the code

```
.align 4
.word 1
.align 4
.word 2
.align 4
.word 3
```

- or a repetition list:

Example

```
.irp val, [25,3,1989]
.align 4
.word {val}
.endr
```

which is equivalent to the code

```
.align 4
.word 25
.align 4
.word 3
.align 4
.word 1989
```

This feature is fully described in Section 7.9.

- **Text Inclusion**

Text from another file may be included as part of the file being assembled. For example:

```
.include useful.definitions
```

places code that is the contents of file `useful.definitions`. This feature is fully described in Section 7.12.

- **Listing of Expanded Code**

A listing output may be produced to display all expanded code. The listing output can be generated after either the macro-processing phase or after the second phase of the assembly process. This feature is fully described in Section 7.3.

- **User Error and Warning Message**s

You can issue error and warning messages.

Example given the following macro:

```
.macro check_reg_number reg_number
.if {reg_number} > 7
.merror invalid register number specified.
.endifnnnnnnn
.endm
```

the call:

```
check_reg_number 10
```

results in the error message:

```
Assembler (Macro-Processor) Error:
 "filename.s", line 4, statement is ==> .merror invalid regis-
ter number specified<==
 . . . from line 9 : while calling "check_reg_number" with
"ARG_LIST"=[10]

ERROR: invalid register number specified
```

These features are fully described in Section 7.13.

- **Arithmetic Operations and Expressions**

Arithmetic operations and expressions (including arithmetic comparisons) can be performed on constants and variables.

For example, assuming the macro-variable `x` holds the string value `100`, the statement:

```
 result:= { {x} * ({x} - 1) }
```

is processed by the macro-assembler so that the macro-variable `result` holds the string value `9900`.

Arithmetic operations and expressions are fully described in Section 7.5.

- **Built-in Macro Functions**

Built-in macro-functions provide the following capabilities:

- Manipulation of strings and lists.
- Integer and floating-point conversions.
- Manipulation of instruction operand strings.

7.2 THE MACRO-PROCESSING PHASE

Assembly source text is processed by the assembler in two distinct phases: the macro-processing phase and the assembly phase.

The macro-processing phase involves the reading and processing of source text statement by statement. Strings between braces ({}) are handled and replaced with the appropriate value. If the resulting statement is a macro-directive statement or a macro-procedure call it is acted upon. All other statements are not processed by the macro-processor and are passed directly to the assembly phase.

The assembly phase is performed in two passes and generates the appropriate output files.

A more detailed explanation of the various stages of the macro-processing follows below.

A string between braces is handled as follows:

- A macro-variable name is replaced with the current value of the variable. For example, if the variable a holds the value ***xxy 100***, then ***{a}*** is replaced by ***xxy 100***.
- An arithmetic macro-expression is evaluated and replaced with the result. For example, ***{100*(10+10)}*** is replaced by ***2000***.
- A built-in macro-function call is evaluated and replaced by the result. For example, ***{STR_LEN[abcde]}*** is replaced by ***5***.
- All other braced strings cause an error message to be issued.

Pairs of braces may be nested, in which case the string contained by the inner pair of braces is evaluated and replaced first.

Example assuming the macro-variable **op** holds the value **r2**, the following statement:

```
movd $0,r{ {SUB_STR[ {op}, 2, 1]} + 4}
```

is replaced by:

```
movd $0,r6
```

- First, **{op}** is replaced by **r2**.
- Then, the function call **{SUB_STR[r2, 2, 1]}** is replaced by 2.
- Finally, **{2 + 4}** is arithmetically evaluated and replaced by the resulting string 6.

Braces are not processed in the macro-processing phase if they appear within either an ASCII constant (such as "{1+1}") or a character constant (such as '{').

A statement followed by a backslash (\) before a carriage-return (***<CR>***) is concatenated with its previous statement. The statements are then treated as a single statement (any number of lines may be concatenated in this way). This does not apply to comments. Comments are terminated by a carriage return (***<CR>***) even if preceded by a backslash (\). This feature may be used in macros for breaking complex expressions into several lines. All error messages refer to the first concatenated statement.

Macro-directives and macro-procedure calls are handled as follows:

- When the opcode field is the name of a previously defined macro-procedure, the statement is considered a macro-procedure call. The statements in the macro-procedure body are processed, as if they were encountered at this point, after matching the actual arguments with the formal arguments of the macro-procedure.
- When the statement is of the form "***symbol := value***", the statement is considered a macro-variable assignment.
  The variable statement on the left side of the **:=** is assigned the value specified on the right side.
- When the opcode field is a **.macro** directive, the statement is considered a macro-procedure definition.
  The statements following the **.macro** directive, but before the **.endm** directive, are read textually without being processed and are stored internally.
- When the opcode field is an **.if** directive, a conditional block is begun.
  Statements following a true clause are processed; statements following an untrue clause are read textually without being processed and are discarded.
- When the opcode field is a **.repeat** or a **.irp** directive, a repetitive block is begun. The statements of the block are repetitively processed, according to the operands of the **.repeat** or **.irp** directive.
- When the opcode field is a **.include** directive, the specified file is read and processed.

7.3 INVOCATION

Several aspects of macro-processing can be controlled by assembler invocation options. The following table presents these specific options:

Option	Definition
-MD*name* or **-MD*name=def***	Defines name to macro assembler as if by macro assignment statement.
-ML*filename*	Includes macro library file.
-MI*dir*	Specifies an include search directory.
-MO	Invoke only macro-processing phase.
-MP*filename*	Prints the macro-processing output.

The **-MD** option assigns an initial value to a macro variable.

The **-ML** option includes an already existing macro-library. A macro-library is any valid assembly file using the Version 4 macro-assembler features; as described in Section 7.12 for the ***included_file*** of the **.include** directive.

The **-MI** option sets a search directory for included files. The assembler searches for .include files which do not begin with a slash (**/**) in the directory of the specified input file first, then in the directory named in this option.

The **-MO** option invokes only the macro-processing phase of the assembler.

The **-MP** option causes the assembler to print the macro-processor's output to ***filename***. If ***filename*** is not given, the output is written to standard output.

7.4 MACRO VARIABLES

Macro-variables are variables that are active only during the macro-processing phase.

The name of a macro-variable may be any assembly symbol as defined in Section 3.5, i.e., a sequence of letters, digits, underscores (_) and periods (**.**). The first character may not be a digit. A period (**.**) should not be used as the first character of the variable name since it may be confused with a directive name.

The initial value of any macro-variable is the empty string, unless it has been assigned a value on the invocation line (see Section 7.3) through the **-MD** macro invocation option. Generally, you assign a value to a macro-variable through a macro-variable assignment statement.

The value of an undefined variable is an empty string.

`macro_var := { value }`

This assigns the value of the macro variable ***value*** to ***macro_var***, after stripping leading and trailing blanks. If ***value*** is omitted, an empty string is assigned to ***macro_var***.

A macro-variable is substituted with its current value when its name is enclosed within braces.

`{ macro_var }`

Examples

```
AAA := 5+5
```

assigns the string value 5+5 to the macro-variable AAA.

```
XXX := 7
XXX := {XXX}+1
```

assigns the string value 7+1 to the macro-variable XXX.

```
XXX := 7
XXX := {{XXX} + 1}
```

assigns the value 8 to the macro-variable XXX.

```
VAR_NAME:= XXX
{VAR_NAME} := 7
```

assigns the value 7 to the macro-variable XXX.

```
EEE := eee
FFF := fff
LLL := [ ddd, {EEE}, {FFF}, ggg]
```

assigns the value of the macro-list [ddd, eee, fff, ggg] to the macro-variable LLL.

7.5 ARITHMETIC MACRO-EXPRESSIONS

An arithmetic macro-expression is a string whose contents are a legal combination of integer constants, arithmetic operators, comparison operators and parentheses. This string can be evaluated as an integer value.

Examples of various arithmetic macro-expressions are:

- 1000

- 20+8*(3/2)
- assuming that the value of ***a*** is ***50*** and that the value of ***b*** is ***+***, then:
  {a} {b} 27
  is also a legal arithmetic macro-expression (equivalent to 50 + 27).

When an arithmetic macro-expression is enclosed between braces or used in an arithmetic context (for example, the clause of a **.if** / **.elsif** directive or as the first operand of a **.repeat** directive), it is evaluated by the macro-processor and substituted with a string representing its value. This string contains an integer constant in signed decimal notation with no leading blanks. Arithmetic macro-expressions are evaluated and converted by the macro-assembler to a 32-bit signed integer representation. All arithmetic operations are performed on 32-bit signed integer operands, and also return a 32-bit integer value.

Each arithmetic macro-operator in a macro-expression has a level of precedence. This determines the macro-expression's order of evaluation. Table 7-1 lists all the macro-operators and their precedence for evaluation.

You must follow these rules when writing arithmetic macro-expressions:

- All unary operators must precede a single term and cannot be used to separate two terms.
- All binary operators must separate two terms. For example, the macro-expression **8*4** is legal, but **8**4** is illegal.

Table 7-1. Macro Operation Precedence

Precedence	Operator	Name	Description of Operation
Unary Operator			
1	-	Unary minus	Two's complement (= negation).
1	~	Unary complement	One's complement.
Binary Operator			
2	*	Multiply	Multiply 1st term by 2nd term.
2	/	Divide	Divide 1st term by 2nd term.*
2	%	Modulus	Remainder from 1st term divided by 2nd term.**
2	<<	Shift left	Shift 1st term by 2nd term; emptied bits are zero-filled.

Table 7-1. Macro Operation Precedence (Continued)

Precedence	Operator	Name	Description of Operation
2	>>	Shift right	Shift 1st term by 2nd term; emptied bits are zero-filled.
2	~	Logical OR / complement	Bit-wise OR of 1st term and one's complement of 2nd term.
3	&	Logical AND	Bit-wise AND of 1st and 2nd terms.
3	|	Logical OR	Bit-wise OR of 1st and 2nd terms.
3	^	Logical XOR	Bit-wise XOR of 1st and 2nd terms.
4	+	Add	Add 1st and 2nd terms.
4	-	Subtract	Subtract 2nd term from 1st term.
5	=	Equal	1 if 1st and 2nd terms are equal, 0 otherwise.
5	<>	Not Equal	1 if 1st and 2nd terms are not equal, 0 otherwise
5	>	Greater Than	1 if 1st term is greater than 2nd term, 0 otherwise
5	<	Less Than	1 if 1st term is less than 2nd term, 0 otherwise
5	>=	Greater or Equal	1 if 1st term is greater than or equal to 2nd term, 0 otherwise
5	<=	Less or Equal	1 if 1st term is less than or equal to 2nd term, 0 otherwise
* Rounds toward 0, e.g., -7/3 = -2 and 7/3 = 2 ** e.g., -7%3 = -1 and 7%3 = 1.			

1. Compound macro-expressions are valid. A macro-expression may be constructed from other macro-expressions using unary and binary operators. For example, the two individual macro-expressions **{A}+1** and **{B}+2** may be combined with a multiply operator and parentheses to form the single macro-expression **({A}+1)*({B}+2)**. Note that the parentheses override the default precedence rules.

2. Evaluation of a macro-expression is governed by three factors:

   - *Parentheses* - macro-expressions enclosed in parentheses are evaluated first. For example, **{8/4/2}** is evaluated as **1**, but **{8/(4/2)}** is evaluated as **4**.

   - *Precedence Groups* - an operation of a higher precedence group is evaluated before an operation of a lower precedence group whenever parentheses do not otherwise determine the evaluation order. For example, **{8+4/2}** is evaluated as **10**, but **{8/4+2}** is evaluated as **4**.

- *Left to Right Evaluation* - macro-expressions are evaluated from left to right whenever parentheses and precedence groups do not determine evaluation order. For example, **{8*4/2}** is evaluated as **16**, and **{8/4*2}** is evaluated as **4.**

7.6 MACRO LISTS

A macro-list is a sequence of strings separated by commas and enclosed between brackets. Each string in the macro-list is called an element. An element of a macro-list may itself be a macro-list, allowing for multilevel macro-lists.

Macro-lists are useful for implementing macro data-structures (such as arrays, records, stacks) in conjunction with built-in functions that perform macro-list manipulations, such as search, insertion and deletion of elements (see Section 7.16). Some examples of various types of macro-lists are:

Examples

```
[]
```

a macro-list with no elements

```
[xx,yy]
```

a macro-list with two elements: xx and yy.

```
[a,,]
```

a macro-list with three elements: a and two empty strings.

```
[[r1,r2],100]
```

a macro-list with two elements: a macro-list with two elements and the string 100.

```
[12[r2:w],@xx]
```

a macro-list with two elements.

7.7 BUILT-IN MACRO FUNCTIONS

The macro-assembler provides built-in functions to manipulate macro-strings, arithmetic constants, macro-lists and assembly operands.

The general syntax for calling a macro-function is:

{macro_func param_list}

macro_func is the name of the function

param_list is a macro-list in which each element is a parameter to the function.

Leading and trailing blanks of parameters are stripped before processing the macro-function. The macro-function call is then evaluated and replaced by the result of the function call.

Example

The macro-function call

```
{SUB_STR[ abcde , 3 , 2]}
```

is replaced by the string cd.

Below is a list of available built-in functions. The macro-list and operand functions are advanced features of the macro-assembler and therefore may not be necessary for all users. For a detailed description of these functions see Sections 7.15 through 7.18.

String Functions

- **{STR_LEN[*string*]}**
- **{STR_EQ[*string1, string2*]}**
- **{SUB_STR[*string, start* [*, length*]]}**
- **{STR_FIND[*string, substring*]}**

Macro-List Functions

- {LIST_GET[***list, element_number***]}
- **{SUB_LIST[*list, start* [, *length*]]}**
- **{LIST_FIND[*list, string*]}**
- {LIST_REPL[***list, element_number, string***]}
- **{LIST_INS[*list, string, element_number*]}**
- **{LIST_DEL[*list, element_number*]}**
- **{LIST_LEN[*list*]}**

Data Conversion Functions

- **{CNV_HEX[*integer_constant*]}**
- **{CNV_HEXF[*constant*]}**
- **{CNV_HEXL[*constant*]}**

Instruction Operand Functions

- **{OP_TYPE[*operand*]}**
- **{OP_REG[*operand*]}**
- **{OP_DISP1[*operand*]}**
- **{OP_DISPSIZE1[*operand*]}**
- **{OP_VAL[*operand*]}**
- **{OP_VALSIZE[*operand*]}**

7.8 CONDITIONAL ASSEMBLY

Sequences of statements may be generated according to conditions tested during the macro-processing phase.

7.8.1 Conditional Block

```
.if if_condition
      if_conditional_body
[ .elsif elsif_condition
       elsif_conditional_body ] ...
[ .else
       else_conditional_body ]
.endif
```

if_condition and ***elsif_condition(s)***
are arithmetic macro-expressions.

A condition evaluated by the macro-assembler as a non-zero value is considered to be true. See Section 7.5 for details of macro-expression evaluation.

In a conditional block the ***if_condition*** argument is evaluated first, and only if found to be true the statements in ***if_conditional_body*** are processed. If the ***if_condition*** is found to be false, the ***elsif_condition(s)*** arguments are evaluated until one of them is found to be true, in which case the corresponding ***elsif_conditional_body*** statements are processed. Otherwise, if an **.else** statement has been specified, the ***else_conditional_body*** statements are processed.

The types of statements that are allowed in ***conditional_bodies*** are valid assembly language statements, directives, macro-procedure call and macro-assembly directives, with all the conditional blocks, repetitive blocks and macro-procedure definitions being complete.

Example

```
.if {reg_num} > 5
     movd $5, r{reg_num}
.elsif {reg_num} > 3
     movd $3, r{reg_num}
.else
     movd $1, r{reg_num}
.endif
```

If ***reg_num*** holds the value 6 this is expanded to:

```
     movd $5, r6
```

if ***reg_num*** holds the value 4 this is expanded to:

```
movd $3, r4
```

and if ***reg_num*** holds the value 0 this is expanded to:

```
movd $1, r0
```

7.9 REPETITIVE DIRECTIVES

The basic constructs of a repetitive block are:

.repeat ***[iteration_count [, iteration_var]]***

repetitive_body

.endr

and

.irp ***iteration_var, iteration_list***

repetitive_body

.endr

Repetitive blocks may appear inside a macro-procedure definition, in conditional blocks, and may be nested without limit.

The types of statements allowed in a repetitive block are valid assembly language statements, directives, macro-procedure calls, macro-assembly directives (except the **.macro** and the **.endm** directives) with all conditional blocks and repetitive blocks being complete.

7.9.1 .repeat Directive

.repeat ***[iteration_count [, iteration_var]]***

iteration_count
: specifies the number of iterations.

iteration_var
: is a macro-variable name used as an iteration index.

The ***iteration_count*** argument is evaluated by the macro-processor. If its value is positive, the code following the **.repeat** statement through the corresponding **.endr** statement, is processed ***iteration_count*** times.

If given, the ***iteration_var*** argument holds a string representing the current iteration number for each iteration. It receives values from 1 to ***iteration_count***. After the processing of the repetitive block has been completed, it holds the ***iteration_count*** value.
If the ***iteration_count*** argument is evaluated as a negative or zero value, the statements in the block are read textually without being processed until an **.endr** directive is reached.
If the ***iteration_count*** argument is not given, then the repetitive block is processed repeatedly until an **.exit** directive is processed (see Section 7.9.3.)

Example

```
.repeat 8, i
movd  $0,r{{i} - 1}
.endr
```

generates code that clears **r0** through **r7**.

```
.repeat 4
 nop
.endr
```

generates four consecutive nop instructions.

7.9.2 .irp Directive

.irp ***iteration_var***, **iteration_list**

iteration_var
is a macro-variable name to be used as an iteration variable.

iteration_list
is a macro-list

For each element in the ***iteration_list*** argument, the macro-processor assigns its string value to ***iteration_var***, and process the code between the **.irp** statement and the corresponding **.endr** statement.

If the ***iteration_list*** argument is an empty macro-list, the statements in the block are read textually without being processed.

After the processing of the repetitive block has been completed, ***iteration_var*** contains the last element of ***iteration_list***.

Example

```
.irp reg, [r0,r1,r2,r3,r4,r5,r6,r7]
movd $0,{reg}
.endr
```

generates code that clears registers **r0** through **r7**.

7.9.3 `.exit` Directive

```
.exit
```

Terminates the processing of the current repetitive block. Statements following this directive are read textually without being processed, until an **`.endr`** statement is encountered.

Example

```
x:=1
.repeat
 .if {x} > 30
.exit
.endif
.byte {x}
x:={{x}*2}
.endr
```

generates the code

```
.byte 1
.byte 2
.byte 4
.byte 8
.byte 16
```

7.10 MACRO PROCEDURES (*MACROS*)

With a macro-procedure you can associate a macro name with a sequence of statements. This sequence can be generated by specifying the macro name in the opcode field, optionally with arguments.

7.10.1 Macro Procedure Definition

```
.macro macro-name [ formal-arg [ , formal-arg ] ... ]
macro-procedure-body
.endm [ macro-name ]
```

macro-name is the macro-procedure name. It may be any legal assembler symbol.

formal-arg is a macro-variable defining a formal argument.

macro-procedure-body
are the statements to be inserted into the assembler code when the macro-procedure is called.

The statements of the macro-procedure body are read textually without being processed and are stored internally.

Within a macro-procedure body, other macro-procedure definitions are not allowed and all conditional and repetitive blocks must be complete. If ***macro-name*** is given in the **.endm** directive, it must be the same as given in the corresponding **.macro** directive.

A macro-procedure can only be defined once in an assembly file and its definition must precede any call to it.

The formal arguments in the **.macro** directive specify the names of the macro-variables to be assigned values according to the actual arguments, when the macro-procedure is called and expanded. The specification of formal arguments in the definition of a macro-procedure is optional.

Example

```
.macro clear_array size, base_reg
          # clears an array of 'size' double-words whose
          # base address is in 'base_reg'
.repeat {size}, elem_num
clear_elem {elem_num}, {base_reg}
.endr
.endm
.macro clear_elem elem_num, base_reg
          # clears element number 'elem_num' of
          # an array whose address is in 'base_reg'
movd $0, {4 * ({elem_num} - 1)}({base_reg})
.endm
clear_array 3, r4
```

expands to

```
          movd $0, 0(r4)
          movd $0, 4(r4)
          movd $0, 8(r4)
```

7.10.2 Macro Procedure Call and Expansion

macro-name [***actual-arg*** [, ***actual-arg***] ...]

A macro-procedure is called by specifying its name in the opcode field of the statement, provided it has already been defined. The name of the invoked macro-procedure may be followed by a sequence of actual arguments separated by commas.

When a macro-procedure call is processed, the current value of each macro-variable specified as formal argument is *saved*, and the macro-variable is assigned the value of its corresponding actual argument instead.

The body of the called macro-procedure is read from storage and processed as if it were inserted instead of the macro-procedure call statement. This is called macro-procedure expansion.

A macro-variable specified as a formal argument for the macro-procedure may be used in the macro-procedure body as any other macro-variable.

The number of actual arguments and the number of formal arguments do not have to correspond. If there are more formal arguments than actual arguments, the unmatched formal arguments are assigned the value of an empty string. If there are more actual than formal arguments, the unmatched actual arguments can be accessed by using the predefined macro-procedure **ARG_LIST**. See the following section for more details on **ARG_LIST**.

7.10.3 Predefined Macro Procedure Variables

Three macro-variables, **ARG_COUNT**, **ARG_LIST**, and **ARG_LABEL** are predefined macro-procedure variables. When a macro-procedure is called and expanded, their current values are saved, and they are assigned new values according to:

- **ARG_COUNT** - is assigned the number of arguments actually passed to the macro-procedure.
- **ARG_LIST** - is assigned the value of a macro-list, whose elements are the actual arguments to the macro-procedure. The first element of **ARG_LIST** is always the first actual argument.
- **ARG_LABEL** - is assigned the value of the label of the macro-procedure invocation. It is assigned a value only if a label appears on the same line as a macro invocation.

These predefined variables cannot be specified as formal arguments.

Example "print_i_call" creates a calling sequence for the subroutine "print_integers" by pushing its parameters, and the number of parameters on the stack.

```
#PIKE

.macro print_i_call

     i := 0
     addw ${-2* ({{ARG_COUNT} +1})},sp
     .irp arg,{ARG_LIST}
          movw {arg},{i}(sp)
          i := {{i}+2}
     .endr
     movw ${ARG_COUNT},{i}(sp)
```

```
     bal ra,print_integers
     addw ${2*({ARG_COUNT} +1)},sp
.endm
```

The following call:

```
print_i_call $100,xx,0(r3)
```

generates:

```
     addw $-8,sp
          movw $100,0(sp)
          movw xx,2(sp)
          movw 0(r3),4(sp)
          movw $3,6(sp)
          bal ra,print_integers
     addw $8,sp
```

7.11 .MACRO_ON AND .MACRO_OFF DIRECTIVES

The **macro_on** and **macro_off** directives enable and disable macro-procedure expansions, respectively, in selective parts of the source text. This is useful when macro-procedure names contradict opcode mnemonic or assembler directives. Thus, if for example opcode addd is redefined as a macro-procedure without the using the **.macro_off** directive (as shown below), it would develop into an infinite sequence of recursive macro-procedure calls. However, the **macro_off** directive allows disabling of macro-procedure expansions. As can be seen for:

```
.macro addw op1,op2
 bal ra, count_additions
.macro_off
 addw {op1},{op2}
.macro_on
.endm
```

the following macro-procedure call:

```
addd r1,r2
```

generates:

```
bal ra, count_additions
addd r1,r2
```

7.12 TEXT INCLUSION

This feature allows for the inclusion of text from another file as part of the file being assembled. The inclusion of text can also be specified from the invocation line by use of the **-ML** macro-library option.

`.include` ***included_file***

included_file
is an existing file name

An **.include** directive causes the macro-processor to process statements from the file named ***included_file*** before processing the statements following the **.include** directive in the original file.

By default, if the ***included_file*** argument does not start with a **/**, only the directory in which the source file resides is searched. Additional directories for the ***included_file*** argument can be searched as specified on the invocation line using the macro Include Search Directory option **(-MI)**.

Included files may contain any valid assembly directives and statements, macro-procedure call or macro-assembly directives (in particular **.include** directives), macro-procedure calls or macro-assembly directives, with all conditional blocks, repetitive blocks and macro-procedure definitions being complete.

Example

```
.include filehdr.h
```

7.13 MACRO WARNING AND ERROR MESSAGES

The directives **.mwarning** and **.merror** generate assembler warning and error messages.

7.13.1 .mwarning Directive

`.mwarning` ***warning_message***

When a statement with a **.mwarning** directive is processed by the macro-processor, a warning message with the source file name, the current line number and ***warning_message*** is displayed on the assembler listing output (or written to the standard error file, if no listing output has been requested in the invocation line).

Example

```
xx:= 222
.mwarning current value of "xx" is : {xx}.
```

In this example the **.mwarning** directive may be used to write the current value of macro-variables on the listing output. The assembler issues the following warning message:

```
Assembler (Macro-Processor): "filename.s", line 2 , WARNING :
current value of "xx" is : 222
```

7.13.2 .merror Directive

```
.merror error_message
```

When a statement with a **.merror** directive is processed by the macro-processor, an error message with the source file name, the current line number and ***error_message*** is displayed on the listing output (or written to the standard error file, if no listing has been requested in invocation line). The assembly process that follows is terminated after the macro-processing phase is completed, and the second phase, the assembly phase, is suppressed.

Example

```
.merror Wrong value used for addr "address"
```

The assembler issues the following error message:

```
Assembler (Macro-Processor) Error:
"f.s", line 1, statement is ==> .merror Wrong value used for
addr "address" <==
ERROR: Wrong value used for addr "address"
```

7.14 LISTING CONTROL

Macro processor expansions can be output in two ways. After the macro processing phase, expansions can be output to the assembler. After the full assembly process is completed, a complete assembly listing file can be produced.

To display macro processor expansions after the macro processing phase, invoke the assembler with the **-MP** option. The display contains the expansions of the macro processor as assembly statements, with other non-macro assembly statements. See Section 7.3 for full details on the **-MP** option.

To list macro processor expansions after the full assembly process, invoke the assembler with the **-L** option. This option produces a complete listing. When the **-L** option is used, the **.list** and **.nolist** directives can be used to select parts of the assembly source file to be listed. In addition, qualifiers can be used with these directives to include or exclude certain levels of macro expansions. **.list** turns the qualifiers ON and **.nolist** turns them OFF.

The qualifiers are:

mac_source When ***mac_source*** is ON, the assembler lists user source lines, before any macro expansions or macro substitutions have been made. The default setting is ON.

mac_expansions When ***mac_expansions*** is ON, the assembler lists user source lines, after all macro substitutions have been performed on them. The default setting is OFF.

mac_directives When ***mac_directives*** is ON, the macro directives also appear in the source listing. The default setting is ON.

It is not necessary to include the **.list** directive to use the default settings of the qualifiers. The **-L** option automatically produces a list and assumes the default qualifier settings.

For source level debugging, use the default settings of the qualifiers to produce a listing in which the displayed lines correspond to the line numbering recognized by the debugger.

For assembly level debugging, set ***mac_source*** and ***mac_directives*** OFF and ***mac_expansions*** ON to produce a listing in which the displayed lines correspond to the actual generated code.

When both ***mac_source*** and ***mac_expansions*** are OFF, no listing is produced. This combination is equivalent to **.nolist** with no parameters.

You should not use the ***mac_source*** option when both ***mac_directives*** and ***mac_expansions*** are OFF. This combination produces output which is difficult to read.

In the default setup, the expansions of macro procedure calls, **.repeat,** and **.irp** blocks, are not listed.

Example (Default)

```
mac_source= ON
mac_expansions= OFF
mac_directives= ON
```

This source file:

```
#PIKE
.macro zero_reg regno
     movw $0,r{regno}
.endm
lab1:
zero_reg 0
.repeat 7,i
zero_reg {i}
.endr
```

Produces this listing:

```
CompactRISC Assembler Version X.X  Date  Page: 1

##### File "ex8.s" #####

1                                    #PIKE
2                                    .macro zero_reg regno
2                                           movw $0,r{regno}
2                                    .endm
5
6    T00000000                       lab1:
7    T00000000      0038             zero_reg 0
8                                    .repeat 7,i
8                                    zero_reg {i}
8    T00000002      20384038        .endr
                    60388038
                    a038c038
                    e038
```

When ***mac_expansions*** is ON, and ***mac_source*** and ***mac_directives*** are both OFF, only the output of the macro processing phase, as passed on to phase-1 of the assembler, is listed.

Example

mac_source= OFF
mac_expansions= ON
mac_directives= OFF

This source file:

```
#PIKE
.list mac_expansions
.nolist mac_source mac_directives
.macro zero_reg regno
     movw $0,r{regno}
.endm
lab1:
zero_reg 0
.repeat 7,i
zero_reg {i}
.endr
```

Produces this listing:

```
CompactRISC Assembler Version X.XX Date    Page: 1

##### File "ex9.s" #####

1                               #PIKE
2                               .list mac_expansions
3                               .nolist mac_source
mac_directives
7
8    T00000000                  lab1:
9    T00000000      0038        movw $0,r0
10   T00000002      2038        movw $0,r1
10   T00000004      4038        movw $0,r2
10   T00000006      6038        movw $0,r3
10   T00000008      8038        movw $0,r4
10   T0000000a      a038        movw $0,r5
10   T0000000c      c038        movw $0,r6
10   T0000000e      e038        movw $0,r7
```

When both ***mac_source*** and ***mac_expansions*** are ON, each source line expanded by the macro assembler is printed twice: first as it appears in the source, and then as it appears after the expansion.

Example

mac_source= ON
mac_expansions= ON
mac_directives= OFF

This source file:

```
#PIKE
.list mac_expansions
.macro zero_reg regno
     movw $0,r{regno}
.endm
lab1:
zero_reg 0
.repeat 7,i
zero_reg {i}
.endr
```

produces this listing:

```
GNX Assembler Version X.XX Date   Page: 1

##### File "ex10.s" #####

1                               #PIKE
2                               .list mac_expansions
3                               .macro zero_reg regno
3                                    movw $0,r{regno}
3                               .endm
6
```

```
7    T00000000              lab1:
8                           zero_reg 0
8                                movw $0,r{regno}
8    T00000000      0038         movw $0,r0
9                           .repeat 7,i
9                           zero_reg {i}
9                           zero_reg 1
9                                movw $0,r{regno}
9    T00000002      2038         movw $0,r1
9                           zero_reg {i}
9                           zero_reg 2
9                                movw $0,r{regno}
9    T00000004      4038         movw $0,r2
9                           zero_reg {i}
9                           zero_reg 3
9                                movw $0,r{regno}
9    T00000006      6038         movw $0,r3
9                           zero_reg {i}
9                           zero_reg 4
9                                movw $0,r{regno}
9    T00000008      8038         movw $0,r4
9                           zero_reg {i}
9                           zero_reg 5
9                                movw $0,r{regno}
9    T0000000a      a038         movw $0,r5
9                           zero_reg {i}
9                           zero_reg 6
9                                movw $0,r{regno}
9    T0000000c      c038         movw $0,r6
9                           zero_reg {i}
9                           zero_reg 7
9    T0000000e      e038         movw $0,r{regno}
9                                movw $0,r7
9                           .end
```

After expansion of a macro or a **.repeat/.irp** block has started, it cannot be reversed. However, it is possible to expand only one level by starting a macro or **.repeat/.irp** block with ***mac_expansion*** ON, and switch it OFF inside a block. Only the outer level is expanded.

Example This source file

```
#PIKE
.list mac_expansions
.macro zero_reg regno
     movw $0,r{regno}
.endm
lab1:
zero_reg 0
.repeat 7,i
.if {i} = 2
```

```
     .nolist mac_expansions
.endif
zero_reg {i}
.endr
```

produces this listing:

```
GNX Assembler Version X.XX  Date   Page: 1

##### File "ex11.s" #####

1                               #PIKE
2                               .list mac_expansions
3                               .macro zero_reg regno
3                                    movw $0,r{regno}
3                               .endm
6
7    T00000000                  lab1:
8                               zero_reg 0
8                                    movw $0,r{regno}
8    T00000000      0038             movw $0,r0
9                               .repeat 7,i
9                               .if {i} = 2
9                               .if 1 = 2
9                               .endif
9                               zero_reg {i}
9                               zero_reg 1
9                                    movw $0,r{regno}
9    T00000002      2038             movw $0,r1
9                               .if {i} = 2
9                               .if 2 = 2
9                                    .nolist mac_expansions
9                               .endif
9    T00000004      4038       zero_reg {i}
9                               .if {i} = 2
9                               .endif
9    T00000006      6038       zero_reg {i}
9                               .if {i} = 2
9                               .endif
9    T00000008      8038       zero_reg {i}
9                               .if {i} = 2
9                               .endif
9    T0000000a      a038       zero_reg {i}
9                               .if {i} = 2
9                               .endif
9    T0000000c      c038       zero_reg {i}
9                               .if {i} = 2
9                               .endif
9    T0000000e      e038       zero_reg {i}
                                .endr
```

7.15 STRING FUNCTIONS

The macro-assembler provides a set of built-in functions to manipulate strings: string length, string comparison, substring extraction, and sub-string search.

Characters in strings are counted starting number 1. For example, in the string **abcde, a** is character number 1, **b** is character number 2, and so on.

7.15.1 String Length

{STR_LEN[***string***]}

Evaluates as the number of characters in ***string***.

Examples

{STR_LEN[abcd]} is evaluated as 4.

{STR_LEN[ab cd]} is evaluated as 5.

{STR_LEN[]} is evaluated as 0.

7.15.2 String Comparison

{STR_EQ[***string1***, ***string2***]}

Evaluates as 1 if ***string1*** and **string2** are the same, and as 0 if they are not.

Example

```
#SR
.macro movmem src_add, dest_add
     .if {STR_EQ[{src_add}, {dest_add}]}
     .else
          addd $-4,sp
          stord r1, 0(sp)
          loadd src_add,r1
          stord r1,dest_add
          loadd 0(sp),r1
          addd $4,sp
     .endif
.endm
```

7.15.3 Substring Extraction

{SUB_STR[***string, start*** [, ***length***]]}

Extracts a substring of the ***string*** argument from position ***start***. Generally, length is taken to be substring size. If the ***length*** argument is omitted or is greater than the remaining length of the ***string*** argument, then the length of the substring is the remaining length of the string.

The function call is evaluated as an empty string when:

- ***start*** is less than or equal to zero.
- ***start*** is greater than the length of the string.
- ***length*** is less than or equal to zero.

Examples:

1. {SUB_STR[abcdefgh,2,3]} evaluates to bcd.
2. {SUB_STR[abcdefgh,3]} evaluates to cdefgh.
3. {SUB_STR[abcdefgh,1000,3]} evaluates to an empty string.

7.15.4 Substring Search

{STR_FIND[***string***, ***substring***]}

Evaluates as the position of the first character of ***substring*** in its first occurrence in ***string***. If ***substring*** is not found, the value of the function is ***0***.

Examples:

1. {STR_FIND[abcdefgh,cde]} evaluates to 3.
2. {STR_FIND[abcabc,c]} evaluates to 3.
3. {STR_FIND[abcdefgh,zz]} evaluates to 0.

7.16 MACRO-LIST FUNCTIONS

A macro-list is a string that contains substrings separated by commas and that is enclosed between brackets. Each of these substrings is called an ***element*** of the macro-list. See Section 7.6 for more details about macro-lists.

The macro-assembler includes a set of built-in functions to process macro-lists that allow creation and manipulation of array-like and other complex structures (stacks, queues, etc.). The built-in functions are: sub-list extraction, retrieval, search, insertion and deletion of elements into/from lists. Another built-in function returns the number of elements in a macro-list.

Elements in macro-lists are counted starting from the left with number 1. For example, in the macro-list [**aa, bb, cc, dd**], element number 1 is **aa**, element number 2 is **bb**, and so on.

7.16.1 Get Element From List

{LIST_GET[***list, element_number***]}

Evaluates as the element whose number is specified by ***element_number***.

Example {LIST_GET[[a,b,c,d],2]} is evaluated as the string b.

7.16.2 Sublist Extraction

{SUB_LIST[***list, start*** [, ***length***]]}

Evaluates as a macro-list of ***length*** elements from the ***list***, starting at element number ***start***. If ***length*** is omitted or is greater than the number of remaining elements, all remaining elements are included in the sublist.

In the following cases, the function call is evaluated as an empty macro-list ***[]***:

- ***start*** is less than or equal to zero.
- ***start*** is greater than the number of elements in ***list***.
- ***length*** is less than or equal to zero.

Examples:

1. {SUB_LIST[[a,b,c,d,e,f,g,h],2,3]} is evaluated as [b,c,d].
2. {SUB_LIST[[a,b,c,d,e,f,g,h],3]]} is evaluated as [c,d,e,f,g,h].
3. {SUB_LIST[[a,b,c,d,e,f,g,h],1000,3]} is evaluated as [].

7.16.3 Find An Element In List

{LIST_FIND[***list, string***]}

Evaluates as the position (element number) of the first occurrence of ***string*** as an element of ***list***. If ***string*** is not an element of ***list***, the function call is evaluated as **0**.

Example After the assignment:

dummy_list:=[hhh,r1,ii,x,hh,x]

then:

1. {LIST_FIND[{dummy_list},r1]}is evaluated as 2.
2. {LIST_FIND[{dummy_list},yyy]}is evaluated as 0.

3. {LIST_FIND[{dummy_list},x]}is evaluated as 4.

7.16.4 Replace An Element In A List

{LIST_REPL[***list, element_number, string***]}

Evaluates as ***list*** after replacing the element, whose number is specified by ***element_number***, with the given ***string***. This macro-function is useful, when a macro-list is handled as an array, for assigning a value to a specified element in a macro-list.

Example

```
dum_list:=[xx,yy,zz]
dum_list:={LIST_REPL[{dum_list},2,aa]}
```

The second element of ***dum_list*** has been "assigned" (replaced with) the string ***aa***, and ***dum_list*** now holds the value ***[xx,aa,zz]***.

7.16.5 Insert An Element Into A List

{LIST_INS[***list, string, element_number***]}

Evaluates as ***list*** after inserting ***string*** as an element before the element specified by ***element_number***.

Example

```
list1:=[aa,bb,cc]
list2:={LIST_INS[{list1},dd,3]}
```

list2 holds the value [aa,bb,dd,cc].

7.16.6 Delete An Element From A List

{LIST_DEL[***list, element_number***]}

Evaluates as ***list*** after removing the element whose number is specified by ***element_number***.

Example

```
list1:=[aa,bb,cc]
list2:={LIST_DEL[{list1},2]}
```

list2 holds the value [aa,cc].

list1 remains to hold the original value [aa,bb,cc].

7.16.7 Number Of Elements In A List

```
{LIST_LEN[ list ]}
```

Evaluates as the number of elements in *list*.

Example

```
vars_list:=[-12(fp),-16(fp),-20(fp),r0,r1[r4:b],r2]
{LIST_LEN[{vars_list}]}.
```

evaluates to 6.

7.16.8 Example of Macro-List Function Usage

Included here is an example showing the capability of the different macro-list functions. A stack-list is implemented using the macro-list functions. We define a set of macro-procedures: **PUSH**, **POP**, **TOP**, **RESET**.

```
.macro          PUSH      list_name,element
                          # pushes an element into a stack list
     {list_name}:={LIST_INS[{{list_name}},{element},1]
# {list_name} evaluates to the NAME of the list.
# {{list_name}} evaluates to its VALUE.
.endm
.macro          POP       list_name,el_var_name
                          # returns the first element of a list, and
                          # removes that first element from it.
     {el_var_name}:={LIST_GET[{{list_name}},1]}
     {list_name}:={LIST_DEL[{{list_name}},1]}
.endm
.macro          TOP       list_name,el_var_name
                          # returns the last element of a list.
     {el_var_name}:={LIST_GET[{{list_name}},1]}
.endm
.macro          RESET     list_name
                          # assign an empty list value [] to the list.
     {list_name}:=[]
.endm
```

In the following sequence of macro-procedure calls, the values of the variables after each call are specified in the comments.

```
var:=
RESETstack1
RESETstack2
                    # value of :
                    # stack1       | stack2   | var
                    # ======       | ======   | ===
                    # []           | []       | empty string
```

```
PUSH stack1,aa      # [aa]         | []       | empty string
PUSH stack1,bb      # [bb,aa]      | []       | empty string
PUSH stack1,cc      # [cc,bb,aa]   | []       | empty string
POP  stack1,var     # [bb,aa]      | []       | cc
PUSH stack2,{var}   # [bb,aa]      | [cc]     | cc
TOP  stack1,var     # [bb,aa]      | [cc]     | bb
RESETstack1         # []           | [cc]     | bb
```

7.17 DATA CONVERSION FUNCTIONS

The macro-assembler provides a set of built-in functions to convert strings representing assembly numerical constants (as defined in Section 3.4) into hexadecimal digit strings. These are integer hexadecimal, float hexadecimal or long float hexadecimal.

7.17.1 Convert To Integer Hexadecimal

{CNV_HEX[***integer_constant***]}

Evaluates as a string of eight hexadecimal digits representing the constant in hexadecimal integer format. The ***integer_constant*** may be specified in any of the integer notations.

Example Given the definition

```
const := 1024
```

then {**CNV_HEX**[{***const***}]} is evaluated as **X'00000400**.

7.17.2 Convert To Float Hexadecimal

{CNV_HEXF[***constant***]}

Evaluates as a string of eight hexadecimal digits representing the constant in float-hexadecimal format. If ***constant*** is not a single precision floating point constant, it is first converted to this representation.

Examples:

1. {CNV_HEXF[{5-4}]} is evaluated as f'3f800000.
2. {CNV_HEXF[1.0e0]} is evaluated as f'3f800000.
3. {CNV_HEXF[l'3ff0000000000000]} # long representation of 1. is evaluated as f'3f800000.

7.17.3 Convert To Long Float Hexadecimal

{CNV_HEXL[***constant***]}

Evaluates as a string of the 16 hexadecimal digits representing the ***constant*** in long hexadecimal-decimal format. If ***constant*** is not a long floating point constant, it is first converted to this representation.

Examples:

1. {CNV_HEXL[{5-4}]} evaluates to `1'3ff0000000000000`.
3. {CNV_HEXL[1.0e0]} evaluates to `1'3ff0000000000000`.
4. {CNV_HEXL[F'3F8000000]} # long representation of 1. evaluates to `1'3ff0000000000000`.

7.18 INSTRUCTION OPERAND FUNCTIONS

The macro-assembler includes a set of built-in functions for processing instruction operands, including recognition of operand type and extraction of subfields from operands strings. These functions provide for ease in using the diversity of operands types and addressing modes provided by the CompactRISC architecture and the CompactRISC assembler.

For example, given an operand string specifying a memory location, another operand string can be created which points to the double word next to that location (i.e., "location+4"). If the operand is a symbol, a leading **4+** string can be concatenated to the operand string. If the operand has been specified with a leading **@** (absolute addressing mode), **4+** can be inserted after the **@**. However with many other operand notations adding such an offset to the location is not as simple. Therefore some convenient built-in functions are provided which recognize the notation (type) in which the operand has been specified, and extract subfields in operand strings.

7.18.1 Recognize The Type Of An Operand

{OP_TYPE[***operand***]}

Evaluates as a string describing the type of *operand*, or as an empty string if the string is not a legal operand. A list of possible operands types are:

EXPR : any legal combination of symbols, constants and arithmetic operators optionally followed by a displacement size specification (:s , :m , :l).

examples: xx:s
12
ss+3+(kk-9):l

GREG	: r0,r1 ...
FREG	: f0,f1,...
LREG	: l0,l1,...
PREG	: processor register : cfg,sp ..
REG_REL	: expression1(register)
EXPL_SB_REL	: ^expression1
IMM	: $expression1

Examples 1. {OP_TYPE[12(sp)]}

is evaluated as REG_REL.

2. {OP_TYPE[$xx+121]}

is evaluated as IMM.

Note The **OP_TYPE** built-in macro-function can not always provide the definite addressing mode that is used for the operand. Information returned by this function is just the most accurate conclusion that can be drawn about the nature of the operand, through scanning the operand string and without any knowledge of the context in which the operand appears or of the type of the user-symbols (e.g., labels) involved in the operand. Since this knowledge is mandatory for determining the exact addressing mode in which the operand is encoded, and since the macro-processing phase is done prior to the assembly phase, this information is unavailable during the macro-processing phase.

7.18.2 Operand Subfields

The following are the various operand subfield functions

{OP_REG[***operand***]}

If the operand is a register, it is evaluated as that register. If the operand has a base register, it is evaluated as the base register. No register is returned if the operand is a register list.

Example {OP_REG[yy+8(sp)]}

is evaluated as sp.

{OP_DISP1[operand]}

If the operand contains at least one displacement field, with or without a displacement size specification, the function call is evaluated as the innermost displacement string without the displacement size specification. Otherwise the empty string is returned.

Note that when the operand type is **DREF_SYM**, the innermost displacement string is returned.

Example `{OP_DISP1[yy+8(sp)]}`

is evaluated as `yy+8`.

`{OP_DISPSIZE1[ operand ]}`

If the innermost displacement string has a size specified, that size specification is returned. Otherwise, the empty string is returned.

Example `{OP_DISPSIZE1[yy+8(sp)]}`

evaluates to an empty string.

`{OP_VAL[ operand ]}`

If the operand is **EXPR, ABS, IMM, EXPL_PC_REL,** or **EXPL_SB_REL**, it is evaluated as the expression without the size or any preceding literals.

Example `{OP_VAL[$yy+8]}`

is evaluated as `yy+8`.

`{OP_VALSIZE[ operand ]}`

If the operand is **EXPR, ABS, IMM, EXPL_PC_REL**, or **EXPL_SB_REL**, it evaluates to its specified size.

Example `{OP_VALSIZE[$yy+8:s]}`

is evaluated as `:s`

The following table defines the subfields that are relevant to various operand types.

Type	Reg	Disp	Size1	Val	Valsize	List
EXPR				+	+	
GREG	+					
FREG	+					

Type	Reg	Disp	Size1	Val	Valsize	List
LREG	+					
PREG	+					
REG_REL	+	+	+			
EXPL_PC_REL				+	+	
EXPL_SB_REL				+	+	
IMM				+	+	

Appendix A
DIRECTIVE SUMMARY

The following is a comprehensive summary of the CompactRISC Assembler Directives.

SYMBOL GENERATION	
`.set` ***symbol, expression***	sets ***symbol*** to the value and type specified by ***expression***. Scope is local.

DATA GENERATION	
[***label***] `.` ***string***	generates a string constant. ***string*** specifies constant value.
[***label***] `.byte`([[***repetition-factor***]]***string***),,,	generates byte constant or string. ***expression*** or ***string*** specifies constant value. ***repetition-factor*** specifies number of occurrences of value.
[***label***] `.word`([[***repetition-factor***]] ***{expression | string}***),,,	generates word constants. ***expression*** specifies constant value.
[***label***] `.double`([[***repetition-factor***]] ***{expression | string}***),,,	generates double-word constants.
[***label***] `.float`([[***repetition-factor***]]***expression***),,,	generates single-precision floating-point constants.
[***label***] `.long`([[***repetition-factor***]]***expression***),,,	generates double-precision floating-point constants.
[***label***] `.field`([***subfield-length***] ***subfield-value***),,,	generates bit fields. ***subfield-length*** specifies length of field. ***subfield-value*** specifies field value.

STORAGE ALLOCATION	
[*label*] **.blkb** [*expression*]	allocates consecutive bytes of memory for storage. ***expression*** specifies the number of bytes.
[*label*] **.blkw** [*expression*]	allocates consecutive words of memory for storage. ***expression*** specifies the number of words.
[*label*] **.blkd** [*expression*]	allocates consecutive double-words of memory for storage. expression specifies the number of double-words.
[*label*] **.blkf** [*expression*]	allocates consecutive double-words for single-precision floating-point storage. ***expression*** specifies the number of double-words.
[*label*] **.blkl** [*expression*]	allocates consecutive quad-words for double-precision floating-point storage. ***expression*** specifies the number of quad-words.
[*label*] **.space** ***expression***	allocates consecutive bytes for storage. ***expression*** specifies the number of bytes.

LISTING CONTROL	
[*label*] **.title** ***string***	prints the specified ***string*** at the top of each page of the listing file.
[*label*] **.subtitle** ***string***	prints the specified ***string*** at the top of each page of the listing file and below the title string (if any).
[*label*] **.nolist**	suppresses the printing of source statements to the listing file.
[*label*] **.list**	restores the printing of source statements to the listing file.
[*label*] **.eject**	causes the listing to continue at the top of the next page.
[*label*] **.width** ***expression***	sets the width (in characters) of the listing line to the value specified by ***expression***.

LINKAGE CONTROL	
.globl ***symbol***	declares ***symbol*** "global," that is, available for use by other software modules. ***symbol*** may or may not be defined in the current assembly.
.comm ***symbol, expression***	declares global data ***symbol***, assigns it the external undefined type, and allocates for it the number of bytes specified by the ***expression***. The ***symbol*** will be placed in the .bss section by the linker.

.**code_label** ***symbol***	qualifies a symbol as a label of code. In the CR16 architectures, the address of such a label is shifted by 1 to the right, in some cases, when it is used as an operand.

SEGMENT CONTROL

.**dsect** ***symbol***, [***expression***]	begins a user-defined segment ***symbol***, with current location counter ***expression***.
.**text**	sets location counter to type text and the value of the next available .**text** address.
.**data**	sets location counter to type data and the value of the next available .**data** address.
.**bss** ***symbol, expression1, expression2***	defines a symbol of type bss, aligns the bss location counter to a multiple of ***expression2***, and increments the bss location counter by ***expression1*** bytes. The current location counter is not affected.
.**udata**	sets location counter to type bss and the value of the next available .**bss** address.
.**org** ***expression***	advances the location counter to ***expression***. ***expression*** must be of the same type as the location counter or of type absolute.
.**align** ***expression1[,expression2]***	sets location counter offset to a multiple of ***expression1*** or the sum of a multiple of ***expression1*** and ***expression2***. The location counter type remains unchanged. New value (>= current value.
.**ident** ***string***	places the string argument in the .comment section of the object file. This directive may be used more than once. The .comment section is given the section attribute of STYP_INFO. The linker will combine all .**comment** sections at link time.
.**section** ***section_name[,string]***	defines a section with attributes. The ***section_name*** is the name of the section, and each character in *string* represents an attribute. Symbols declared within a section belong to that particular section. A section is active until the next .section, **.text, .data**, **.udata, .link** or **.static** directive. In the default case, reference to symbols of a user-defined section are referenced via the absolute addressing mode.

FILE NAME DIRECTIVE	
.**file** "***symbol***"	assigns the source filename ***symbol*** to the current assembly.

SYMBOL TABLE ENTRY DEFINITION DIRECTIVES	
.**def** ***symbol***	specifies the start of the definition of a symbol table entry for ***symbol***.
.**dim** ***expression,,,***	specifies the dimensions of an array variable. Up to four dimensions may be specified.
.**line** ***expression***	specifies the source file line number, ***expression***, on which a symbol is defined.
.**scl** ***expression***	specifies the storage classification, ***expression***, of a symbol.
.**size** ***expression***	***expression*** specifies the size in bytes of a symbol.
.**tag** ***symbol***	***symbol*** specifies the tag name of a structured data type.
.**type** ***expression***	specifies the type, ***expression***, of a symbol.
.**val** ***expression***	***expression*** specifies the value of the symbol that is being defined.
.**endef**	terminates the definition of a symbol table entry.

LINE NUMBER TABLE CONTROL DIRECTIVE	
.**ln** ***expression1*** [, ***expression2***]	specifies the source file line number offset from the start of a function and an optional, associated memory address.

MACRO DEFINITION DIRECTIVES	
.**macro** ***macro-name formal-argument-list***	
	begins the definition of the macro-procedure.
.**endm**	end the macro-procedure definition.
.if ***if_condition***	begins a conditional macro assembler statement.
.**elsif** ***elsif_condition***	specifies an **elsif** clause for the conditional macro assembler statement.
.**else** ***else_conditional_body***	specifies an **else** clause for the conditional macro assembler statement.

.endif	ends the conditional macro assembler statement.
.repeat [***iteration_count***[**,*iteration_var***]]	begins a macro repetitive block.
.irp ***iteration_var, iteration_list***	begins a special macro repetitive block.
.endr	ends a macro repetitive block.
.exit	ends the processing of the current repetitive block.
.macro_on	enables macro-procedure expansions.
.macro_off	disables macro-procedure expansions.
.include ***included_file***	allows for the inclusion of text from another file.
.mwarning ***warning_message***	generates an assembler warning message.
.merror ***error_message***	generates an assembler error message.

Appendix B
RESERVED SYMBOLS

All instructions and registers, and other symbols defined in the architecture, are reserved symbols.

In addition to these, the assembler itself has the following reserved symbols:

.align
.ascii
.blkb
.blkd
.blkf
.blkl
.blkw
.byte
.callseq
.code_label
.comm
.cross_bound
.def
.dim
.double
.dsect
.dspmincr
.dspminit
.dspmtype
.dspmwrap
.eject
.endef
.field
.file
.float
.fp_mask
.frame_size
.freg_mask
.fsize
.globl
.gp_mask
.ident

.intr
.line
.list
.ln
.long
.minit
.msize
.nolist
.org
.reg_mask
.regoff
.retaddr
.scl
.section
.set
.size
.space
.subtitle
.tag
.title
.type
.type_ext
.udata
.val
.width
.word
mac_directives
mac_expansions
mac_source
reordered_code

Appendix C
GLOSSARY

.gnxrc A GNX target specification file that is used by GNX tools to obtain the CPU, FPU, MMU, system bus-width, and OS target specifications.

.Assembly Program segment Part of an assembly program that resides in a contiguous area. Every GNX assembly program produces at least three program segments in the output object file: text, data, and bss. These segments correspond to the .text, .data, and .bss sections of the COFF file. Other *Series 32000* segments or user-defined sections may be included in the assembly source file.

Assembly directive Provides the assembler with control information. Directives define labels, generate data, define procedures, control program listings, control macro-assembly, allocate storage, control linkage, define module table entries, control line number tables, control program segments, define symbol table entries, and define file names.

Assembly expression A combination of terms and operators which evaluate to a single value and type. Valid expressions include addresses and integer expressions, but not floating-point expressions.

Assembly label A user-defined symbol specified at the beginning of an assembly statement, followed by a colon (:) or a double colon (::).

Assembly statement Composed of an optional label, which is a user-defined symbol; followed by an optional instruction or directive mnemonic that is an assembler-reserved symbol; followed by optional operands that are composed of symbols, constant values, and delimiters.

Built-in Macro Functions A set of macro-assembler functions used to manipulate strings, lists, type conversions and *Series 32000* operands.

COFF Acronym for the Common Object File Format. This is the standard object file format for many software development tools, as well as the CompactRISC software development tools. A COFF file contains machine code and data and additional information for relocation and debugging purposes.

Calling convention A standard CompactRISC convention for calling procedures from either an assembly or a HLL written code. It defines the way parameters are passed, register usage and how a value should be returned.

Compound Assembly expression An expression constructed from other assembly expressions using unary and binary operators.

Conditional Macro Statement Sequences of statements specified between the .if and the .endif directives. They are generated according to a condition specified with the .if directive.

Cpp An acronym for the C preprocessor.

Cross configuration When the compilation and execution of the compiled program are done on different machines (the host and target machines are different).

DBUG CompactRISC symbolic debugger. DBUG provides a window-oriented user interface for both X-windows and ASCII terminals. It is used for the symbolic debugging of high level and assembly language programs.

Development board The 32000 based system used for developing/running programs and user applications.

Displacement An integer constant that is specified as part of an instruction operand. Its value is an offset added to a specified base address for operand address calculation. It may be encoded as either a byte, word, or doubleword.

Displacement operand A displacement size specification that determines a displacement encoding as either small, medium, or large-word.

Dummy segment Defines a symbolic offset for each of its defined labels. It does not contain generated code or data and does not allocate space. It is useful for overlaying portions of specific segments.

External symbol A symbol which is defined outside the assembled file. It can be defined either in another assembly file or in a HLL file.

Floating-point constant An immediate floating-point value. Can be either a four byte single precision value or an eight byte double precision value.

Global symbols Global symbols are symbols to be used by multiple software modules, either assembly or HLL modules.

Host machine The machine on which the assembler runs.

Initialized data segment Contains initialized data, follows the .data directive, and corresponds to the .data section of the COFF file. The initialized data segment has the same functionality as initialized data in the C language. This functionality enables the start-up of a target system with an automatically initialized data area.

Instruction operand The instruction operand is defined by the microprocessor architecture as one of nine possible addressing modes: register, immediate, absolute, register-relative, memory space.

Integer constant An immediate integer value. Can be specified either in decimal, hexadecimal or octal format. Integers can be used within assembly expressions that are part of either an instruction or directive operand.

Link segment A special segment of the assembly program that corresponds to the .link section of the COFF file. The link segment defines a module's link table, thereby supporting *Series 32000* modularity. The actual link table entries are specified following the assembly .link directive.

Location counter A relocatable memory address of the current statement within the currently assembled segment.

Macro Procedure Known by the more common name: macro. Consists of legal assembly statements to be expanded on a macro call, according to given parameters.

Object file A file that is the output of either the assembler or the compiler. It contains compiled code, data and additional control information such as relocation or symbolic information. The assembler's object file conforms to the COFF Common Object File Format.

Option A parameter, specified on the command line, used to control the utility.

Output listing An optional assembler output of the assembled source file. It displays the original assembly source, along with additional useful information. Each source line has an annotated line number, segment type information, and the generated code or data. Macro expansions are also displayed where applicable.

Relative value A symbol or expression that specifies an address within one of the COFF sections or the corresponding assembly program segment. Because such addresses are not bound to actual memory locations until link-time, their value is relative to the base or starting address of the segment. Relative values are relocatable, and have a relocatable entry in the generated COFF object file. They are resolved later at link-time.

Relocatable object files Output of the assembly process. Relocatable object files may be linked to create executable files for a *Series 32000* target.

Repetitive Macro Statement Sequences of statements specified between the .repeat or .irp directives and the .endr directive. They are generated repeatedly according to an iteration index specified with the .repeat directive.

Return value An integer or floating-point value that is returned by a function through register R0 or F0, respectively, according to the CompactRISC standard calling convention.

Series 32000 instruction A *Series 32000* instruction mnemonic. Should appear within a text section in order to be executed.

Source file Assembler input. The source file is a text file containing the source program to be assembled.

Target machine The machine on which the program being compiled will run.

Text segment Contains code for execution, follows the .text directive and corresponds to the .text section of the COFF file.

Uninitialized data segment Contains uninitialized data, follows the .bss or the .udata directive. Corresponds to the .bss section of the COFF file.

INDEX

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W

X

Y

Z