MIPSpro™ Assembly Language Programmer's Guide

007–2418–006

CONTRIBUTORS

Originally written by Larry Huffman, David Graves

Engineering contributions by Bean Anderson, Jim Dehnert, Suneel Jain, Michael Murphy, George Pirocanac.

Cover design by Sarah Bolles, Sarah Bolles Design, and Dany Galgani, SGI Technical Publications.

New Features in This Manual

The `.restore` and `.save` directives have been added and are documented in Chapter 8, "Pseudo Op-Codes (Directives)", page 109.

The assembler now supports branches to label+offset.

Information regarding the assembly language file produced using the `-S` option to the compilers is documented in Chapter 7, "Writing Assembly Language Code", page 87. Note that this chapter was entitled “Linkage Conventions” in previous versions of this manual.

Record of Revision

Version	Description
	1996. Original Printing.
7.3	April 1999 Revised to support the MIPSpro 7.3 release.
005	September 2002 Revised to support the MIPSpro 7.4 release which runs on the IRIX operation system version 6.5 and later.
006	June 2003 Revised to support the MIPSpro 7.4.1m release.

Contents

Figures

Tables

Examples

About This Guide

This publication describes the assembly language supported by the IRIX operating system, its syntax rules, and how to write assembly programs. For information about assembling and linking an assembly language program, see the *MIPSpro N32/64 Compiling and Performance Tuning Guide*.

This book assumes that you are an experienced assembly language programmer. The assembler produces object modules from the assembly instructions that the C and Fortran compilers generate. It therefore lacks many of the functions normally present in an assembler. You should use the assembler only when you must:

- Maximize the efficiency of a routine, which might not be possible in C, Fortran, or another high-level language (for example, to write low-level I/O drivers).
- Access machine functions unavailable in high-level languages or satisfy special constraints such as restricted register usage.
- Change the operating system.
- Change the compiler system.

The assembler converts assembly language statements into machine code. In most assembly languages, each instruction corresponds to a single machine instruction; however, some assembly language instructions can generate several machine instructions. This feature results in assembly programs that can run without modification on future machines, which might have different machine instructions.

In this release, the assembler supports compilations in `-o32`, `-n32`, and `-64` mode. Some of the implications of these different data sizes are explained in this book. For more information, see the *MIPSpro 64-Bit Porting and Transition Guide*.

Many assembly language instructions have direct equivalents to machine instructions. For more information about the operations of a specific architecture, see the book that is appropriate for your hardware type.

Related Publications

This manual is one of a set of manuals that describes the compiler. The complete set of manuals is as follows:

- *MIPSpro 64-Bit Porting and Transition Guide*
- *MIPSpro N32/64 Compiling and Performance Tuning Guide*
- *MIPSpro N32 ABI Handbook*

Obtaining Publications

To obtain SGI documentation, go to the SGI Technical Publications Library at:

`http://docs.sgi.com.`

Conventions

The following conventions are used throughout this document:

Convention	Meaning
`command`	This fixed-space font denotes literal items such as commands, files, routines, path names, signals, messages, and programming language structures.
variable	Italic typeface denotes variable entries and words or concepts being defined.
`user input`	This bold, fixed-space font denotes literal items that the user enters in interactive sessions. (Output is shown in nonbold, fixed-space font.)
[ ]	Brackets enclose optional portions of a command or directive line.

Reader Comments

If you have comments about the technical accuracy, content, or organization of this publication, contact SGI. Be sure to include the title and document number of the publication with your comments. (Online, the document number is located in the front matter of the publication. In printed publications, the document number is located at the bottom of each page.)

You can contact SGI in any of the following ways:

- Send e-mail to the following address:

  techpubs@sgi.com

- Use the Feedback option on the Technical Publications Library Web page:

  `http://docs.sgi.com`

- Contact your customer service representative and ask that an incident be filed in the SGI incident tracking system.

- Send mail to the following address:

  Technical Publications
  SGI
  1600 Amphitheatre Parkway, M/S 535
  Mountain View, California 94043–1351

- Send a fax to the attention of "Technical Publications" at +1 650 932 0801.

SGI values your comments and will respond to them promptly.

Registers

This chapter describes the organization of data in memory, and the naming and usage conventions that the assembler applies to the CPU and FPU registers. See Chapter 7, "Writing Assembly Language Code", page 87, for information regarding register use and linkage.

Register Format

The CPU uses four data formats: a 64-bit doubleword, a 32-bit word, a 16-bit halfword and an 8-bit byte. Byte ordering within each of the larger data formats – doubleword, word or halfword – the CPU's byte ordering scheme (or endian issues), affects memory organization and defines the relationship between address and byte position of data in memory.

For R4000 and earlier systems, byte ordering is configurable into either big-endian or little-endian byte ordering (configuration occurs during hardware reset). When configured as a big-endian system, byte 0 is always the most-significant (leftmost) byte. When configured as a little-endian system, byte 0 is always the least-significant (rightmost byte).

The R8000 CPU, at present, supports big-endian only.

General Registers

For the MIPS1 and MIPS2 architectures, the CPU has thirty-two 32-bit registers. In the MIPS3 architecture and above, the size of each of the thirty-two integer registers is 64-bit.

Table 1-1, page 2, and Table 1-2, page 3, summarize the assembler's usage, conventions and restrictions for these registers. The assembler reserves all register names; you must use lowercase for the names. All register names start with a dollar sign (`$`).

The general registers have the names `$0..$31`. By including the file `regdef.h` (use `#include <regdef.h>`) in your program, you can use software names for some general registers.

The operating system and the assembler use the general registers `$1`, `$26`, `$27`, `$28`, and `$29` for specific purposes. Attempts to use these general registers in other ways can produce unexpected results.

Table 1-1 General (Integer) Registers (`-32`)

Register Name	Software Name (from `regdef.h`)	Use and Linkage
$0		Always has the value 0.
$1 or $at		Reserved for the assembler.
$2..$3	v0-v1	Used for expression evaluations and to hold the integer type function results. Also used to pass the static link when calling nested procedures.
$4..$7	a0-a3	Pass the first 4 words of actual integer type arguments; their values are not preserved across procedure calls.
$8..$11 $11..$15	t0-t7 t4-t7 or ta0-ta3	Temporary registers used for expression evaluations; their values aren't preserved across procedure calls.
$16..$23	s0-s7	Saved registers. Their values must be preserved across procedure calls.
$25	t9 or jp	PIC jump register.
$26..27 or $kt0..$kt1	k0-k1	Reserved for the operating system kernel.
$28 or $gp	gp	Contains the global pointer.
$29 or $sp	sp	Contains the stack pointer.
$30 or $fp	fp or s8	Contains the frame pointer (if needed); otherwise a saved register (like s0-s7).
$31	ra	Contains the return address and is used for expression evaluation.

Note: General register `$0` always contains the value 0. All other general registers are equivalent, except that general register `$31` also serves as the implicit link register for jump and link instructions. See Chapter 7, "Writing Assembly Language Code", page 87, for a description of register assignments.

Table 1-2 General (Integer) Registers (64-Bit)

Register Name	Software Name (from `regdef.h`)	Use and Linkage
$0		Always has the value 0.
$1 or $at		Reserved for the assembler.
$2..$3	v0-v1	Used for expression evaluations and to hold the integer type function results. Also used to pass the static link when calling nested procedures.
$4..$7 $8..$11	a0-a3 a4-a7 or ta0-ta3	Pass up to 8 words of actual integer type arguments; their values are not preserved across procedure calls.
$12..$15	t0-t3	Temporary registers used for expression evaluations; their values aren't preserved across procedure calls.
$16..$23	s0-s7	Saved registers. Their values must be preserved across procedure calls.
$24..$25	t8-t9	Temporary registers used for expression evaluations; their values aren't preserved across procedure calls.
$26..27 or $kt0..$kt1	k0-k1	Reserved for the operating system kernel.
$28 or $gp	gp	Contains the global pointer.
$29 or $sp	sp	Contains the stack pointer.

Register Name	Software Name (from `regdef.h`)	Use and Linkage
$30 or $fp	fp or s8	Contains the frame pointer (if needed); otherwise a saved register (such as s0-s7).
$31	ra	Contains the return address and is used for expression evaluation.

Special Registers

The CPU defines three special registers: PC (program counter), HI and LO, as shown in Table 1-3, page 4. The HI and LO special registers hold the results of the multiplication (`mult` and `multu`) and division (`div` and `divu`) instructions.

You usually do not need to refer explicitly to these special registers; instructions that use the special registers refer to them automatically.

Table 1-3 Special Registers

Name	Description
PC	Program Counter
HI	Multiply/Divide special register holds the most-significant 32 bits of multiply, remainder of divide
LO	Multiply/Divide special register holds the least-significant 32 bits of multiply, quotient of divide

Note: In MIPS3 architecture and later, the HI and Lo registers hold 64-bits.

Floating-Point Registers

The FPU has sixteen floating-point registers. Each register can hold either a single-precision (32-bit) or double-precision (64-bit) value. In case of a double-precision value, `$f0` holds the least-significant half, and `$f1` holds the

most-significant half. For 32-bit systems, all references to these registers use an even register number (for example, `$f4`). 64-bit systems can reference all 32 registers directly. The following tables summarize the assembler's usage conventions and restrictions for these registers.

Table 1-4 Floating-Point Registers (`-32`)

Register Name	Software Name (from `fgregdef.h`)	Use and Linkage
$f0..$f2	fv0-fv1	Hold results of floating-point type function ($f0) and complex type function ($f0 has the real part, $f2 has the imaginary part.
$f4..$f10	ft0-ft3	Temporary registers, used for expression evaluation whose values are not preserved across procedure calls.
$f12..$f14	fa0-fa1	Pass the first two single- or double-precision actual arguments; their values are not preserved across procedure calls.
$f16..$f18	ft4-ft5	Temporary registers, used for expression evaluation, whose values are not preserved across procedure calls.
$f20..$f30	fs0-fs5	Saved registers, whose values must be preserved across procedure calls.

Table 1-5 Floating-Point Registers (`-64`)

Register Name	Software Name(from `fgregdef.h`)	Use and Linkage
$f0, $f2	fv0,fv1	Hold results of floating-point type function ($f0) and complex type function ($f0 has the real part, $f2 has the imaginary part.
$f1, $f3, $f4..$f11	ft12, ft13, ft0–ft7	Temporary registers, used for expression evaluation; their values are not preserved across procedure calls.
$f12..$f19	fa0-fa7	Pass single– or double–precision actual arguments, whose values are not preserved across procedure calls.
$f20..$f23	ft8-ft11	Temporary registers, used for expression evaluation; their values are not preserved across proceadure calls.
$f24..$f31	fs0-fs7	Saved registers, whose values must be preserved across procedure calls.

Table 1-6 Floating-Point Registers (`-n32`)

Register Name	Software Name(from `fgregdef.h`)	Use and Linkage
$f0, $f2	fv0,fv1	Hold results of floating-point type function ($f0) and complex type function ($f0 has the real part, $f2 has the imaginary part.)
$f1, $f3 $f4..$f11	ft14, ft15, ft0-ft7	Temporary registers, used for expression evaluation; their values are not preserved across procedure calls.
$f12..$f19	fa0-fa7	Pass single– or double–precision actual arguments, whose values are not preserved across procedure calls.

Register Name	Software Name(from `fgregdef.h`)	Use and Linkage
$f21, $f23, $f25, $f27, $f29, $f31	ft8-ft13	Temporary registers, used for expression evaluation; their values are not preserved across procedure calls.
$f20, $f22, $f24, $f26, $f28, $f30	fs0-fs5	Saved registers, whose values must be preserved across procedure calls.

Floating-Point Condition Codes

The floating-point condition code registers hold the result of a floating-point comparison, and then decide whether or not to branch. For `-32` compilers, there is only register: $fcc0. For `-n32` and `-64` compilers, there are eight registers available: `$fcc0` through `$fcc7`.

Addressing

This chapter describes the formats that you can use to specify addresses. SGI CPUs use a byte addressing scheme. Access to halfwords requires alignment on even byte boundaries, and access to words requires alignment on byte boundaries that are divisible by four. Access to doublewords (for 64-bit systems) requires alignment on byte boundaries that are divisible by eight. Any attempt to address a data item that does not have the proper alignment causes an alignment exception.

Instructions to Load and Store Unaligned Data

The unaligned assembler load and store instructions may generate multiple machine language instructions. They do not raise alignment exceptions.

These instructions load and store unaligned data:

- Load doubleword left (LDL)
- Load word left (LWL)
- Load doubleword right (LDR)
- Load word right (LWR)
- Store doubleword left (SDL)
- Store word left (SWL)
- Store doubleword right (SDR)
- Store word right (SWR)
- Unaligned load doubleword (ULD)
- Unaligned load word (ULW)
- Unaligned load halfword (ULH)
- Unaligned load halfword unsigned (ULHU)
- Unaligned store doubleword (USD)
- Unaligned store word (USW)

- Unaligned store halfword (USH)

The following instructions load and store aligned data:

- Load doubleword (LD)
- Load word (LW)
- Load halfword (LH)
- Load halfword unsigned (LHU)
- Load byte (LB)
- Load byte unsigned (LBU)
- Store doubleword (SD)
- Store word (SW)
- Store halfword (SH)
- Store byte (SB)

Address Formats

The assembler accepts the following formats for addresses: Table 2-2 explains these formats in more detail.

Table 2-1 Address Formats

Format	Address
(*base-register*)	Base address (zero offset assumed)
expression	Absolute address
expression (*base-register*)	Based address
index-register (*base-register*)	Based address
relocatable-symbol	Relocatable address

Format	Address
relocatable-symbol ± *expression*	Relocatable address
relocatable-symbol ± *expression* (*index register*)	Indexed relocatable address

Address Descriptions

The assembler accepts any combination of the constants and operations described in this chapter for expressions in address descriptions.

Table 2-2 Assembler Addresses

Expression	Address Description
(*base-register*)	Specifies an indexed address, which assumes a zero offset. The *base-register* contents specify the address.
expression	Specifies an absolute address. The assembler generates the most locally efficient code for referencing a value at the specified address.
expression (*base-register*)	Specifies a based address. To get the address, the CPU adds the value of the expression to the contents of the base-register.
index-register (base-register)	Same as *expression*(*base-register*), except that the index register is used as the offset.
relocatable-symbol	Specifies a relocatable address. The assembler generates the necessary instruction(s) to address the item and generates relocatable information for the link editor.

Expression	Address Description
relocatable-symbol ± *expression*	Specifies a relocatable address. To get the address, the assembler adds or subtracts the value of the expression, which has an absolute value, from the relocatable symbol. The assembler generates the necessary instruction(s) to address the item and generates relocatable information for the link editor. If the symbol name does not appear as a label anywhere in the assembly, the assembler assumes that the symbol is external.
relocatable-symbol (*index register*)	Specifies an indexed relocatable address. To get the address, the CPU adds the index registerto the relocatable symbol's address. The assembler generates the necessary instruction(s) to address the item and generates relocatable information for the link editor. If the symbol name does not appear as a label anywhere in the assembly, the assembler assumes that the symbol is external.
relocatable ± *expression*	Specifies an indexed relocatable address. To get the address, the assembler adds or subtracts the relocatable symbol, the expression, and the contents of the index register. The assembler generates the necessary instruction(s) to address the item and generates relocation information for the link editor. If the symbol does not appear as a label anywhere in the assembly, the assembler assumes that the symbol is external.

Exceptions

This chapter describes the exceptions that you can encounter while running assembly programs. The system detects some exceptions directly, and the assembler inserts specific tests that signal other exceptions. This chapter lists only those exceptions that occur frequently.

Main Processor Exceptions

The following exceptions are the most common to the main processor:

- Address error exceptions, which occur when a data item is referenced that is not on its proper memory alignment or when an address is invalid for the executing process.
- Overflow exceptions, which occur when arithmetic operations compute signed values and the destination lacks the precision to store the result.
- Bus exceptions, which occur when an address is invalid for the executing process.
- Divide-by-zero exceptions, which occur when a divisor is zero.

Floating-Point Exceptions

The following are the most common floating-point exceptions:

- Invalid operation exceptions which include:
  - Magnitude subtraction of infinities, for example: –1.
  - Multiplication of 0 by 1 with any signs.
  - Division of 0/0 or 1/1 with any signs.
  - Conversion of a binary floating-point number to an integer format when an overflow or the operand value for the infinity or NaN precludes a faithful representation in the format (see Chapter 4, "Lexical Conventions", page 15).
  - Comparison of predicates that have unordered operands, and that involve Greater Than or Less Than without Unordered.

  - Any operation on a signaling NaN.
- Divide-by-zero exceptions.
- Overflow exceptions occur when a rounded floating-point result exceeds the destination format's largest finite number.
- Underflow exceptions these occur when a result has lost accuracy and also when a nonzero result is between 2^{Emin} (2 to the minimum expressible exponent).
- Inexact exceptions.

Lexical Conventions

This chapter discusses lexical conventions for these topics:

This chapter uses the following notation to describe syntax:

- | (vertical bar) means “or”
- [ ](square brackets) enclose options
- ± indicates both addition and subtraction operations

Tokens

The assembler has these tokens:

- Identifiers
- Constants
- Operators

The assembler lets you put blank characters and tab characters anywhere between tokens; however, it does not allow these characters within tokens (except for character constants). A blank or tab must separate adjacent identifiers or constants that are not otherwise separated.

Comments

The pound sign character (#) introduces a comment. Comments that start with a # extend through the end of the line on which they appear. You can also use C-language notation /*...*/ to delimit comments.

The assembler uses `cpp` (the C language preprocessor) to preprocess assembler code. Because `cpp` interprets a # symbol in the first column as pragmas (compiler directives), do not start a # comment in the first column.

Identifiers

An identifier consists of a case-sensitive sequence of alphanumeric characters, including these:

- . (period)
- _ (underscore)
- $ (dollar sign)

The first character of an identifier cannot be numeric.

If an identifier is not defined to the assembler (only referenced), the assembler assumes that the identifier is an external symbol. The assembler treats the identifier like a `.globl` pseudo-operation (see Chapter 8, "Pseudo Op-Codes (Directives)", page 109). If the identifier is defined to the assembler and the identifier has not been specified as global, the assembler assumes that the identifier is a local symbol.

Constants

The assembler has these constants:

- Scalar constants
- Floating-point constants
- String constants

Scalar Constants

The assembler interprets all scalar constants as twos-complement numbers. In 32-bit mode, a scalar constant is 32 bits. 64 bits is the size of a scalar constant in 64-bit mode. Scalar constants can be any of the alphanumeric characters `0123456789abcdefABCDEF`. You can use an `all` or `LL` suffix to identify a 64–bit constant.

Scalar constants can be one of the following:

- Decimal constants, which consist of a sequence of decimal digits without a leading zero.
- Hexadecimal constants, which consist of the characters 0x (or 0X) followed by a sequence of digits.
- Octal constants, which consist of a leading zero followed by a sequence of digits in the range 0..7.

Floating-Point Constants

Floating-point constants can appear only in `.float` and `.double` pseudo-operations (directives) (see Chapter 8, "Pseudo Op-Codes (Directives)", page 109), and in the floating-point Load Immediate instructions (see Chapter 6, "Coprocessor Instruction Set", page 61). Floating-point constants have this format:

```
+d1[.d2] [e|E+d3]
```

where:

- *d1* is written as a decimal integer and denotes the integral part of the floating-point value.
- *d2* is written as a decimal integer and denotes the fractional part of the floating-point value.
- *d3* is written as a decimal integer and denotes a power of 10.
- The "+" symbol is optional.

For example:

```
21.73E-3
```

represents the number .02173.

Optionally, `.float` and `.double` directives may use hexadecimal floating-point constants instead of decimal ones. A hexadecimal floating-point constant consists of:

```
<+ or -> 0x <1 or 0 or nothing> . <hex digits> H 0x <hex digits>
```

The assembler places the first set of hex digits (excluding the 0 or 1 preceding the decimal point) in the mantissa field of the floating-point format without attempting to normalize it. It stores the second set of hex digits into the exponent field without biasing them. It checks that the exponent is appropriate if the mantissa appears to be denormalizing. Hexadecimal floating-point constants are useful for generating IEEE special symbols, and for writing hardware diagnostics.

For example, either of the following generates a single-precision "1.0":

```
.float 1.0e+0
.float 0x1.0h0x7f
```

String Constants

String constants begin and end with double quotation marks (").

The assembler observes C language backslash conventions. For octal notation, the backslash conventions require three characters when the next character can be confused with the octal number. For hexadecimal notation, the backslash conventions require two characters when the next character can be confused with the hexadecimal number (that is, use a 0 for the first character of a single character hex number).

The assembler follows the backslash conventions shown in Table 4-1.

Table 4-1 Backslash Conventions

Convention	Meaning
\a	Alert (0x07)
\b	Backspace (0x08)
\f	Form feed (0x0c)
\n	Newline (0x0a)
\r	Carriage return (0x0d)

Convention	Meaning
\t	horizontal tab (0x09)
\v	Vertical feed (0x0b)
\\	Backslash (0x5c)
\"	Double quotation mark (0x22)
\'	Single quotation mark (0x27)
\000	Character whose octal value is 000
Xnn	Character whose hexadecimal value is *nn*

Multiple Lines Per Physical Line

You can include multiple statements on the same line by separating the statements with semicolons. The assembler does not recognize semicolons as separators when they follow comment symbols (# or /*).

Section and Location Counters

Assembled code and data fall in one of the sections shown in Figure 4-1.

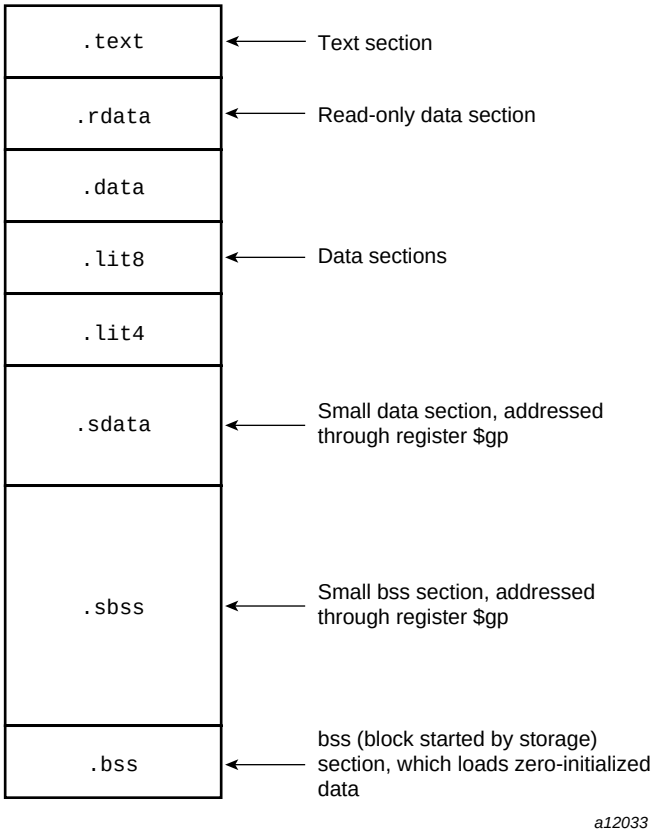

Figure 4-1 Section and Location Counters

The assembler always generates the `text` section before other sections. Additions to the text section happen in four-byte units. Each section has an implicit location counter, which begins at zero and increments by one for each byte assembled in the section.

The `bss` section holds zero-initialized data. If a `.lcomm` pseudo-op defines a variable (see Chapter 8, "Pseudo Op-Codes (Directives)", page 109), the assembler assigns that variable to the `bss` (block started by storage) section or to the sbss (short block started by storage) section depending on the variable's size. The default variable size for `sbss` is 8 or fewer bytes.

The command line option –G for each compiler (C, Pascal, Fortran 77, or the assembler), can increase the size of sbss to cover all but extremely large data items. The link editor issues an error message when the –G value gets too large. If a –G value is not specified to the compiler, 8 is the default. Items smaller than, or equal to, the specified size go in sbss. Items greater than the specified size go in bss.

Because you can address items much more quickly through $gp than through a more general method, put as many items as possible in sdata or sbss. The size of sdata and sbss combined must not exceed 64 KB.

Statements

Each statement consists of an optional label, an operation code, and the operand(s). The system allows these statements:

- Null statements
- Keyword statements

Label Definitions

A label definition consists of an identifier followed by a colon. Label definitions assign the current value and type of the location counter to the name. An error results when the name is already defined, the assigned value changes the label definition, or both conditions exist.

Label definitions always end with a colon. You can put a label definition on a line by itself.

A generated label is a single numeric value (1...255). To reference a generated label, put an f (forward) or a b (backward) immediately after the digit. The reference tells the assembler to look for the nearest generated label that corresponds to the number in the lexically forward or backward direction.

Null Statements

A null statement is an empty statement that the assembler ignores. Null statements can have label definitions. For example, this line has three null statements in it:

```
label: ; ;
```

Keyword Statements

A keyword statement begins with a predefined keyword. The syntax for the rest of the statement depends on the keyword. All instruction opcodes are keywords. All other keywords are assembler pseudo-operations (directives).

Expressions

An expression is a sequence of symbols that represent a value. Each expression and its result have data types. The assembler does arithmetic in twos-complemet integers (32 bits of precision in 32-bit mode; 64 bits of precision in 64-bit mode). Expressions follow precedence rules and consist of:

- Operators
- Identifiers
- Constants

Also, you may use a single character string in place of an integer within an expression. Thus:

```
.byte ‘‘a’’ ; .word ‘‘a’’+0x19
```

is equivalent to:

```
.byte 0x61 ; .word 0x7a
```

Precedence

Unless parentheses enforce precedence, the assembler evaluates all operators of the same precedence strictly from left to right. Because parentheses also designate index-registers, ambiguity can arise from parentheses in expressions. To resolve this ambiguity, put a unary + in front of parentheses in expressions.

The assembler has three precedence levels, which are listed here from lowest to highest precedence:

least binding, lowest precedence	binary +,-

↓

↓ binary *,/,5,<<,>>,^,&, |

↓

most binding, highest precedence unary —,+,~

Note: The assembler's precedence scheme differs from that of the C language.

Expression Operators

For expressions, you can rely on the precedence rules, or you can group expressions with parentheses. The assembler recognizes the operators listed in Table 4-2.

Table 4-2 Expression Operators

Operator	Meaning
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Remainder
<<	Shift Left
>>	Shift Right (sign NOT extended)
^	Bitwise Exclusive-OR
&	Bitwise AND
|	Bitwise OR
-	Minus (unary)
+	Identity (unary)
~	Complement

Data Types

The assembler manipulates several types of expressions. Each symbol you reference or define belongs to one of the categories shown in Table 4-3, page 24.

Table 4-3 Data Types

Type	Description
`undefined`	Any symbol that is referenced but not defined becomes *global undefined*, and this module will attempt to import it. The assembler uses 32-bit addressing to access these symbols. (Declaring such a symbol in a `.globl` pseudo-op merely makes its status clearer).
`sundefined`	A symbol defined by a `.extern` pseudo-op becomes *global small undefined* if its size is greater than zero but less than the number of bytes specified by the `–G` option on the command line (which defaults to 8). The linker places these symbols within a 64KB region pointed to by the `$gp` register, so that the assembler can use economical 16-bit addressing to access them.
`absolute`	A constant defined in an "=" expression.
`text`	The *text* section contains the program's instructions, which are not modifiable during execution. Any symbol defined while the `.text` pseudo-op is in effect belongs to the text section.
`data`	The *data* section contains memory that the linker can initialize to nonzero values before your program begins to execute. Any symbol defined while the `.data` pseudo-op is in effect belongs to the data section. The assembler uses 32-bit or 64-bit addressing to access these symbols (depending on whether you are in 32-bit or 64-bit mode).
`sdata`	This category is similar to *data*, except that defining a symbol while the `.sdata` ("small data") pseudo-op is in effect causes the linker to place it within a 64KB region pointed to by the `$gp` register, so that the assembler can use economical 16-bit addressing to access it.

Type	Description
`rdata`	Any symbol defined while the `.rdata` pseudo-op is in effect belongs to this category, which is similar to `data`, but may not be modified during execution.
`bss` and `sbss`	The *bss* and *sbss* sections consist of memory which the kernel loader initializes to zero before your program begins to execute. Any symbol defined in a `.comm` or `.lcomm` pseudo-op belongs to these sections (except that a `.data`, `.sdata`, or `.rdata` pseudo-op can override a `.comm` directive). If its size is less than the number of bytes specified by the `–G` option on the command line (which defaults to 8), it belongs to `sbss` ("small `bss`"), and the linker places it within a 64 KB region pointed to by the `$gp` register so that the assembler can use economical 16-bit addressing to access it. Otherwise, it belongs to `bss` and the assembler uses 32-bit or 64-bit addressing (depending on whether you are in 32-bit or 64-bit mode). Local symbols in `bss` or `sbss` defined by `.lcomm` are allocated memory by the assembler; global symbols are allocated memory by the link editor; and symbols defined by `.comm` are overlaid upon like-named symbols (in the fashion of Fortran `COMMON` blocks) by the link editor.

Symbols in the undefined and small undefined categories are always global (that is, they are visible to the link editor and can be shared with other modules of your program). Symbols in the `absolute`, `text`, `data`, `sdata`, `rdata`, `bss`, and `sbss` categories are local unless declared in a `.globl` pseudo-op.

Type Propagation in Expressions

When expression operators combine expression operands, the result's type depends on the types of the operands and on the operator. Expressions follow these type propagation rules:

- If an operand is undefined, the result is undefined.
- If both operands are absolute, the result is absolute.
- If the operator is + and the first operand refers to a relocatable `text`-section, `data`-section, `bss`-section, or an undefined external, the result has the postulated type and the other operand must be `absolute`.
- If the operator is – and the first operand refers to a relocatable `text`-section, `data`-section, or `bss`-section symbol, the second operand can be `absolute` (if it previously defined) and the result has the first operand's type; or the second operand can have the same type as the first operand and the result is `absolute`. If the first operand is external undefined, the second operand must be `absolute`.

- The operators * , /, % , << , >> , ~, ^ , & , and | apply only to absolute symbols.

Relocations

With `-n32` and `-64` compiles, it is possible to specify a relocation explicitly in assembly. For example:

```
lui $24,%hi(.data)
```

This example emits a `lui$24,0` instruction with a `R_MIPS_H16` relocation that references the `.data` symbol.

The following table lists the available relocations:

AS-SYNTAX	ELF Relocation
%hi	R_MIPS_HI16
%lo	R_MIPS_LO16
%gp_rel	R_MIPS_GPREL
%half	R_MIPS_16
%call6	R_MIPS_CALL6
%call_hi	R_MIPS_CALL_H16
%call_lo	R_MIPS_CALL_LO16
%got	R_MIPS_GOT
%got_disp	R_MIPS_GOT_DISP
%got_hi	R_MIPS_GOT_HI16
%got_lo	R_MIPS_GOT_LO16
%got_page	R_MIPS_GOT_PAGE
%got_ofst	R_MIPS_GOT_OFST
%neg	R_MIPS_SUB
%higher	R_MIPS_HIGHER
%highest	R_MIPS_HIGHEST

The Instruction Set

This chapter describes instruction notation and discusses assembler instructions for the main processor. Chapter 6, "Coprocessor Instruction Set", page 61, describes coprocessor notation and instructions.

Instruction Classes

The assembler has these classes of instructions for the main processor:

- **Load and Store Instructions.** These instructions load immediate values and move data between memory and general registers.
- **Computational Instructions**. These instructions do arithmetic and logical operations for values in registers.
- **Jump and Branch Instructions.** These instructions change program control flow.

In addition, there are two other classes of instruction:

- **Coprocessor Interface.** These instructions provide standard interfaces to the coprocessors.
- **Special Instructions.** These instructions do miscellaneous tasks.

Reorganization Constraints and Rules

To maximize performance, the goal of RISC designs is to achieve an execution rate of one machine cycle per instruction. When writing assembly language instructions, you must be aware of the rules to achieve this goal. You can find this information in the appropriate microprocessor manual for your architecture (for example, the *MIPS R8000 Microprocessor User's Manual*).

Instruction Notation

The tables in this chapter list the assembler format for each load, store, computational, jump, branch, coprocessor, and special instruction. The format consists of an op-code

and a list of operand formats. The tables list groups of closely related instructions; for those instructions, you can use any op-code with any specified operand.

Operands can take any of these formats:

- Memory references. For example, a *relocatable symbol* +/− an *expression*(*register*).
- Expressions (for immediate values).
- Two or three operands. For example, `ADD $3,$4` is the same as `ADD $3,$3,$4`.

The operands in the table in this chapter have the following meanings

Operand	**Description**
address	Symbolic expression (see Chapter 2)
breakcode	Value that determines the break
destination	Destination register
destination/src1	Destination register is also source register 1
dest-copr	Destination coprocessor register
dest-gpr	Destination general register
expression	Absolute value
immediate	Expression with an immediate value
label	Symbolic label, possibly label+offset
operation	Coprocessor-specific operation
return	Register containing the return address
source	Source register
src1, src2	Source registers
src-copr	Coprocessor register from which values are assigned
src-gpr	General register from which values are assigned
target	Register containing the target
z	Coprocessor number in the range 0..2

Instruction Set

The tables in this section summarize the assembly language instruction set. Most of the assembly language instructions have direct machine equivalents.

Load and Store Instructions

Load and store are immediate type intructions that move data between memory and the general registers. Table 5-1 summarizes the load and store instruction format, and Table 5-2 and Table 5-3 provide more detailed descriptions for each load instruction. Table 5-4, page 35, and Table 5-5, page 37, provide details of each store instruction.

Table 5-1 Load and Store Format Summary

Description	Op-code	Operands
Load Address	LA	*destination*, *address*
Load Doubleword Address	DLA	
Load Byte	LB	
Load Byte Unsigned	LBU	
Load Halfword	LH	
Load Halfword Unsigned	LHU	
Load Linked *	LL	
Load Word	LW	
Load Word Left	LWL	
Load Word Right	LWR	
Load Doubleword	LD	
Unaligned Load Halfword	ULH	
Unaligned Load Halfword Unsigned	ULHU	

Description	Op-code	Operands
Unaligned Load Word	ULW	
Load Immediate	LI	*destination, expression*
Load Doubleword Immediate	DLI	
Store Double Right	SDR	
Unaligned Store Doubleword	USD	
Load Upper Immediate	LUI	
Store Byte	SB	*source, address*
Store Conditional *	SC	
Store Double	SD	
Store Halfword	SH	
Store Word Left	SWL	
Store Word Right	SWR	
Store Word	SW	
Unaligned Store Halfword	USH	
Unaligned Store Word	USW	
Load Doubleword	LD	*destination, address*
Load Linked Doubleword	LLD	
Load Word Unsigned	LWU	
Load Doubleword Left	LDL	
Load Doubleword Right	LDR	
Unaligned Load Double	ULD	
Store Doubleword	SD	*source, address*
Store Conditional Doubleword	SCD	
Store Double Left	SDL	
* not valid in MIPS1 architecture		

Load Instruction Descriptions

For all load instructions, the effective address is the 32-bit twos-complement sum of the contents of the index-register and the (sign-extended) 16-bit offset. Instructions that have symbolic labels imply an index register, which the assembler determines. The assembler supports additional load instructions, which can produce multiple machine instructions.

Note: Load instructions can generate many code sequences for which the link editor must fix the address by resolving external data items.

Table 5-2 Load Instruction Descriptions

Instruction Name	Description
Load Address (LA)	Loads the destination register with the effective 32-bit address of the specified data item.
Load Doubleword Address (DLA)	Loads the destination register with the effective 64-bit address of the specified data item (MIPS3 and above only).
Load Byte (LB)	Loads the least-significant byte of the destination register with the contents of the byte that is at the memory location specified by the effective address. The system treats the loaded byte as a signed value: bit seven is extended to fill the three most-significant bytes.
Load Byte Unsigned (LBU)	Loads the least-significant byte of the destination register with the contents of the byte that is at the memory location specified by the effective address. Because the system treats the loaded byte as an unsigned value, it fills the three most-significant bytes of the destination register with zeros.
Load Halfword (LH)	Loads the two least-significant bytes of the destination register with the contents of the halfword that is at the memory location specified by the effective address. The system treats the loaded halfword as a signed value. If the effective address is not even, the system signals an address error exception.
Load Halfword Unsigned (LHU)	Loads the least-significant bits of the destination register with the contents of the halfword that is at the memory location specified by the effective address. Because the system treats the loaded halfword as an unsigned value, it fills the two most-significant bytes of the destination register with zeros. If the effective address is not even, the system signals an address error exception.

Instruction Name	Description
Load Linked (LL)	Loads the destination register with the contents of the word that is at the memory location. This instruction performs an SYNC operation implicitly; all loads and stores to shared memory fetched prior to the LL must access memory before the LL, and loads and stores to shared memory fetched subsequent to the LL must access memory after the LL. Load Linked and Store Conditional can be use to update memory locations atomically. The system signals an address exception when the effective address is not divisible by four. This instruction is not valid in the MIPS1 architectures.
Load Word (LW)	Loads the destination register with the contents of the word that is at the memory location. The system replaces all bytes of the register with the contents of the loaded word. The system signals an address error exception when the effective address is not divisible by four.
Load Word Left (LWL)	Loads the sign; that is, Load Word Left loads the destination register with the most-significant bytes of the word specified by the effective address. The effective address must specify the byte containing the sign. In a big-endian system, the effective address specifies the lowest numbered byte; in a little-endian system, the effective address specifies the highest numbered byte. Only the bytes which share the same aligned word in memory are merged into the destination register.
Load Word Right (LWR)	Loads the lowest precision bytes; that is, Load Word Right loads the destination register with the least-significant bytes of the word specified by the effective address. The effective address must specify the byte containing the least-significant bits. In a big-endian configuration, the effective address specifies the highest numbered byte; in a little-endian configuration, the effective address specifies the lowest numbered byte. Only the bytes which share the same aligned word in memory are merged into the destination register.
Load Doubleword (LD)	LD is a machine instruction in the MIPS3 architecture. For the `-mips1` [default] and `-mips2` option: Loads the register pair (*destination* and *destination +1*) with the two successive words specified by the address. The destination register must be the even register of the pair. When the address is not on a word boundary, the system signals an address error exception. **Note:** This is retained for use with the `-mips1` and `-mips2` options to provide backward compatibility only.

Instruction Name	Description
Unaligned Load Halfword (ULH)	Loads a halfword into the destination register from the specified address and extends the sign of the halfword. Unaligned Load Halfword loads a halfword regardless of the halfword's alignment in memory.
Unaligned Load Halfword Unsigned (ULHU)	Loads a halfword into the destination register from the specified address and zero extends the halfword. Unaligned Load Halfword Unsigned loads a halfword regardless of the halfword's alignment in memory.
Unaligned Load Word (ULW)	Loads a word into the destination register from the specified address. Unaligned Load Word loads a word regardless of the word's alignment in memory.
Load Immediate (LI)	Loads the destination register with the 32-bit value of an expression that can be computed at assembly time. **Note:** Load Immediate can generate any efficient code sequence to put a desired value in the register.
Load Doubleword Immediate (DLI)	Loads the destination register with the 64-bit value of an expression that can be computed at assembly time. **Note:** Load Immediate can generate any efficient code sequence to put a desired value in the register (MIPS3 and above only).
Load Upper Immediate (LUI)	Loads the most-significant half of a register with the expression's value. The system fills the least-significant half of the register with zeros. The expression's value must be in the range –32768...65535.

Table 5-3 Load Instruction Descriptions for MIPS3/4 Architecture Only

Instruction Name	Description
Load Doubleword (LD)	Loads the destination register with the contents of the doubleword that is at the memory location. The system replaces all bytes of the register with the contents of the loaded doubleword. The system signals an address error exception when the effective address is not divisible by eight.
Load Linked Doubleword (LLD)	Loads the destination register with the contents of the doubleword that is currently in the memory location. This instruction performs a SYNC operation implicitly. Load Linked Doubleword and Store Conditional Doubleword can be used to update memory locations atomically.
Load Word Unsigned(LWU)	Loads the least-significant bits of the destination register with the contents of the word (32 bits) that is at the memory location specified by the effective address. Because the system treats the loaded word as an unsigned value, it fills the four most-significant bytes of the destination register with zeros. If the effective address is not divisible by four, the system signals an address error exception.
Load Doubleword Left (LDL)	Loads the destination register with the most-significant bytes of the doubleword specified by the effective address. The effective address must specify the byte containing the sign. In a big-endian configuration, the effective address specifies the lowest numbered byte; in a little-endian machine, the effective address specifies the highest numbered byte. Only the bytes which share the same aligned doubleword in memory are merged into the destination register.
Load Doubleword Right (LDR)	Loads the destination register with the least-significant bytes of the doubleword specified by the effective address. The effective address must specify the byte containing the least-significant bits. In a bid-endian machine, the effective address specifies the highest numbered byte. In a little-endian machine, the effective address specifies the lowest numbered byte. Only the bytes which share the same aligned doubleword in memory are merged into the destination register.
Unaligned Load Doubleword (ULD)	Loads a doubleword into the destination register from the specified address. ULD loads a doubleword regardless of the doubleword's alignment in memory.

Store Instruction Descriptions

For all machine store instructions, the effective address is the 32-bit twos-complement sum of the contents of the index-register and the (sign-extended) 16-bit offset. The

assembler supports additional store instructions, which can produce multiple machine instructions. Instructions that have symbolic labels imply an index-register, which the assembler determines.

Table 5-4 Store Instruction Descriptions

Instruction Name	Description
Store Byte (SB)	Stores the contents of the source register's least-significant byte in the byte specified by the effective address.
Store Conditional (SC)	Stores the contents of a word from the source register into the memory location specified by the effective address. This instruction implicitly performs a SYNC operation; all loads and stores to shared memory fetched prior to the sc must access memory before the sc, and loads and stores to shared memory fetched subsequent to the sc must access memory after the sc. If any other processor or device has modified the physical address since the time of the previous Load Linked instruction, or if an RFE or ERET instruction occurs between the Load Linked and this store instruction, the store fails. The success or failure of the store operation (as defined above) is indicated by the contents of the source register after execution of the instruction. A successful store sets it to 1; and a failed store sets it to 0. The machine signals an address exception when the effective address is not divisible by four. This instruction is not valid in the MIPS1 architectures.
Store Doubleword (SD)	SD is a machine instruction in the MIPS3 architecture. For the `-mips1` [default] and `-mips2` options: Stores the contents of the register pair in successive words, which the address specifies. The source register must be the even register of the pair, and the storage address must be word aligned. **Note:** This is retained for use with the `-mips1` and `-mips2` options to provide backward compatibility only.
Store Halfword (SH)	Stores the two least-significant bytes of the source register in the halfword that is at the memory location specified by the effective address. The effective address must be divisible by two; otherwise the machine signals an address error exception.

Instruction Name	Description
Store Word Left (SWL)	Stores the most-significant bytes of a word in the memory location specified by the effective address. The contents of the word at the memory location, specified by the effective address, are shifted right so that the leftmost byte of the unaligned word is in the addressed byte position. The stored bytes replace the corresponding bytes of the effective address. The effective address's last two bits determine how many bytes are involved.
Store Word Right (SWR)	Stores the least-significant bytes of a word in the memory location specified by the effective address. The contents of the word at the memory location, specified by the effective address, are shifted left so that the right byte of the unaligned word is in the addressed byte position. The stored bytes replace the corresponding bytes of the effective address. The effective address's last two bits determine how many bytes are involved.
Store Word (SW)	Stores the contents of a word from the source register in the memory location specified by the effective address. The effective address must be divisible by four; otherwise the machine signals an address error exception.
Unaligned Store Halfword (USH)	Stores the contents of the two least-significant bytes of the source register in a halfword that the address specifies. The machine does not require alignment for the storage address.
Unaligned Store Word (USW)	Stores the contents of the source register in a word specified by the address. The machine does not require alignment for the storage address.

Table 5-5 Store Instruction Descriptions for MIPS3/4 Architecture Only

Instruction Name	Description
Store Doubleword (SD)	Stores the contents of a doubleword from the source register in the memory location specified by the effective address. The effective address must be divisible by eight, otherwise the machine signals an address error exception.
Store Conditional Doubleword (SCD)	Stores the contents of a doubleword from the source register into the memory locations specified by the effective address. This instruction implicitly performs a SYNC operation. If any other processor or device has modified the physical address since the time of the previous Load Linked instruction, or if an ERET instruction occurs between the Load Linked instruction and this store instruction, the store fails and is inhibited from taking place. The success or failure of the store operation (as defined above) is indicated by the contents of the source register after execution of this instruction. A successful store sets it to 1; and a failed store sets it to 0. The machine signals an address exception when the effective address is not divisible by eight.
Store Doubleword Left (SDL)	Stores the most-significant bytes of a doubleword in the memory location specified by the effective address. It alters only the doubleword in memory which contains the byte indicated by the effective address.
Store Doubleword Right (SDR)	Stores the least-significant bytes of a doubleword in the memory location specified by the effective address. It alters only the doubleword in memory which contains the byte indicated by the effective address.
Unaligned Store Doubleword (USD)	Stores the contents of the source register in a doubleword specified by the address. The machine does not require alignment for the storage address.

Computational Instructions

The machine has general-purpose and coprocessor-specific computational instructions (for example, the floating-point coprocessor). This section describes general-purpose computational instructions.

Computational Instructions

Computational instructions perform the following operations on register values;

- arithmetic

- logical
- shift
- multiply
- divide

Table 5-6 summarizes the computational format summaries, and Table 5-7, page 41, and Table 5-8, page 47, describe these instructions in more detail.

Table 5-6 Computational Format Summaries

Description	Op-code	Operand
Add with Overflow	`ADD`	*destination*, *src1*, *src2*
Add without Overflow	`ADDU`	*destination*, *src1*, *src2*
AND	`AND`	*destination*, *src1*, *immediate*
Divide Signed	`DIV`	*destination/src1*, *immediate*
Divide Unsigned	`DIVU`	
Exclusive-OR	`XOR`	
Multiply	`MUL`	
Multiply with Overflow	`MULO`	
Multiply with Overflow Unsigned	`MULOU`	
NOT OR	`NOR`	
OR	`OR`	
Set Equal	`SEQ`	
Set Greater Than	`SGT`	
Set Greater/Equal	`SGE`	
Set Greater/Equal Unsigned	`SGEU`	
Set Greater Unsigned	`SGTU`	
Set Less Than	`SLT`	
Set Less/Equal	`SLE`	

Description	Op-code	Operand
Set Less/Equal Unsigned	SLEU	
Set Less Than Unsigned	SLTU	
Set Not Equal	SNE	
Subtract with Overflow	SUB	
Subtract without Overflow	SUBU	
Remainder Signed	REM	
Remainder Unsigned	REMU	
Rotate Left	ROL	
Rotate Right	ROR	
Shift Right Arithmetic	SRA	
Shift Left Logical	SLL	
Shift Right Logical	SRL	
Absolute Value	ABS	*destination*, *src1*
Negate with Overflow	NEG	*destination*/*src1*
Negate without Overflow	NEGU	
NOT	NOT	
Move	MOVE	*destination*, *src1*
Move Conditional on Not Zero	MOVN	*destination*, *src1, src2*
Move Conditional on Zero	MOVZ	
Multiply	MULT	*src1*,*src2*
Multiply Unsigned	MULTU	
Trap if Equal	TEQ	*src1*, *src2*
Trap if not Equal	TNE	*src1*, *immediate*
Trap if Less Than	TLT	
Trap if Less than, Unsigned	TLTU	
Trap if Greater Than or Equal	TGE	
Trap if Greater than or Equal, Unsigned	TGEU	

Description	Op-code	Operand
Doubleword Add with Overflow	DADD	*destination,src1, src2* *destination/src1,src2*
Doubleword Add without Overflow	DADDU	*destination, src1, immediate* *destination/src1, immediate*
Doubleword Divide Signed	DDIV	
Doubleword Divide Unsigned	DDIVU	
Doubleword Multiply	DMUL	
Doubleword Multiply with Overflow	DMULO	
Doubleword Multiply with Overflow Unsigned	DMULOU	
Doubleword Subtract with Overflow	DSUB	
Doubleword Subtract without Overflow	DSUBU	
Doubleword Remainder Signed	DREM	
Doubleword Remainder Unsigned	DREMU	
Doubleword Rotate Left	DROL	
Doubleword Rotate Right	DROR	
Doubleword Shift Right Arithmetic	DSRA	
Doubleword Shift Left Logical	DSLL	
Doubleword Shift Right Logical	DSRL	
Doubleword Absolute Value	DABS	*destination, src1*
Doubleword Negate with Overflow	DNEG	*destination/src1*
Doubleword Negate without Overflow	DNEGU	
Doubleword Multiply	DMULT	*src1, src2*
Doubleword Multiply Unsigned	DMULTU	*src1, immediate*

Computational Instruction Descriptions

Table 5-7 Computational Instruction Descriptions

Instruction Name	Description
Absolute Value (ABS)	Computes the absolute value of the contents of *src1* and puts the result in the destination register. If the value in *src1* is –2147483648, the machine signals an overflow exception.
Add with Overflow (ADD)	Computes the twos-complement sum of two signed values. This instruction adds the contents of *src1* to the contents of *src2*, or it can add the contents of *src1* to the immediate value. Add (with Overflow) puts the result in the destination register. When the result cannot be extended as a 32-bit number, the machine signals an overflow exception.
Add without Overflow (ADDU)	Computes the twos-complement sum of two 32-bit values. This instruction adds the contents of *src1* to the contents of *src2*, or it can add the contents of *src1* to the immediate value. Add (without Overflow) puts the result in the destination register. Overflow exceptions never occur.
AND (AND)	Computes the Logical AND of two values. This instruction ANDs (bit-wise) the contents of *src1* with the contents of *src2*, or it can AND the contents of *src1* with the immediate value. The immediate value is not sign extended. AND puts the result in the destination register.
Divide Signed (DIV)	Computes the quotient of two values. Divide (with Overflow) treats *src1* as the dividend. The divisor can be *src2* or the immediate value. The instruction divides the contents of *src1* by the contents of *src2*, or it can divide *src1* by the immediate value. It puts the quotient in the destination register. If the divisor is zero, the machine signals an error and may issue a BREAK instruction. The DIV instruction rounds toward zero. Overflow is signaled when dividing –2147483648 by –1. The machine may issue a BREAK instruction for divide-by-zero or for overflow. **Note:** The special case DIV $0,*src1*,*src2* generates the real machine divide instruction and leaves the result in the HI/LO register. The HI register contains the remainder and the LO register contains the quotient. No checking for divide-by-zero is performed.

Instruction Name	Description
Divide Unsigned (DIVU)	Computes the quotient of two unsigned 32-bit values. Divide (unsigned) treats *src1* as the dividend. The divisor can be *src2* or the immediate value. This instruction divides the contents of *src1* by the contents of *src2*, or it can divide the contents of *src1* by the immediate value. Divide (unsigned) puts the quotient in the destination register. If the divisor is zero, the machine signals an exception and may issue a BREAK instruction. See the note for DIV concerning $0 as a destination. Overflow exceptions never occur.
Exclusive-OR (XOR)	Computes the XOR of two values. This instruction XORs (bit-wise) the contents of *src1* with the contents of *src2*, or it can XOR the contents of *src1* with the immediate value. The immediate value is not sign extended. Exclusive-OR puts the result in the destination register.
Move (MOVE)	Moves the contents of *src1* to the destination register.
Move Conditional on Not Zero (MOVN)	Conditionally moves the contents of *src1* to the destination register after testing that *src2* is not equal to zero (MIPS4 only.)
Move Conditional on Zero (MOVZ)	Conditionally moves the contents of *src1* to the destination register after testing that *src2* is equal to zero (MIPS4 only).
Multiply (MUL)	Computes the product of two values. This instruction puts the 32-bit product of *src1* and *src2*, or the 32-bit product of *src1* and the immediate value, in the destination register. The machine does not report overflow. **Note:** Use MUL when you do not need overflow protection: it's often faster than MULO and MULOU. For multiplication by a constant, the MUL instruction produces faster machine instruction sequences than MULT or MULTU instructions can produce.
Multiply (MULT)	Computes the 64-bit product of two 32-bit signed values. This instruction multiplies the contents of *src1* by the contents of *src2* and puts the result in the HI and LO registers (see Chapter 1). No overflow is possible. **Note:** The MULT instruction is a real machine language instruction.

Instruction Name	Description
Multiply Unsigned (MULTU)	Computes the product of two unsigned 32-bit values. It multiplies the contents of *src1* and the contents of *src2* and puts the result in the HI and LO registers (see Chapter 1). No overflow is possible.
	Note: The MULTU instruction is a real machine language instruction.
Multiply with Overflow (MULO)	Computes the product of two 32-bit signed values. Multiply (with Overflow) puts the 32-bit product of *src1* and *src2*, or the 32-bit product of *src1* and the immediate value, in the destination register. When an overflow occurs, the machine signals an overflow exception and may execute a BREAK instruction.
	Note: For multiplication by a constant, MULO produces faster machine instruction sequences than MULT or MULTU can produce; however, if you do not need overflow detection, use the MUL instruction. It's often faster than MULO.
Multiply with Overflow Unsigned (MULOU)	Computes the product of two 32-bit unsigned values. Multiply (with Overflow Unsigned) puts the 32-bit product of *src1* and *src2*, or the product of *src1* and the immediate value, in the destination register. This instruction treats the multiplier and multiplicand as 32-bit unsigned values. When an overflow occurs, the machine signals an overflow exception and may issue an BREAK instruction.
	Note: For multiplication by a constant, MULOU produces faster machine instruction sequences than MULT or MULTU can reproduce; however, if you do not need overflow detection, use the MUL instruction. It's often faster than MULOU.
Negate with Overflow (NEG)	Computes the negative of a value. This instruction negates the contents of *src1* and puts the result in the destination register. If the value in *src1* is –2147483648, the machine signals an overflow exception.
Negate without Overflow (NEGU)	Negates the integer contents of *src1* and puts the result in the destination register. The machine does not report overflows.
NOT (NOT)	Computes the Logical NOT of a value. This instruction complements (bit-wise) the contents of *src1* and puts the result in the destination register.

Instruction Name	Description
NOT OR (NOR)	Computes the NOT OR of two values. This instruction combines the contents of *src1* with the contents of *src2* (or the immediate value). NOT OR complements the result and puts it in the destination register.
OR (OR)	Computes the Logical OR of two values. This instruction ORs (bit-wise) the contents of *src1* with the contents of *src2*, or it can OR the contents of *src1* with the immediate value. The immediate value is not sign-extended. OR puts the result in the destination register.
Remainder Signed (REM)	Computes the remainder of the division of two unsigned 32-bit values. The machine defines the remainder REM(*i*,*j*) as *i*–(*j**div(*i*,*j*)) where $j \cdot 0$. Remainder (with Overflow) treats *src1* as the dividend. The divisor can be *src2* or the immediate value. This instruction divides the contents of *src1* by the contents of *src2*, or it can divide the contents of *src1* by the immediate value. It puts the remainder in the destination register. The REM instruction rounds toward zero, rather than toward negative infinity. For example, div(5,–3)=–1, and rem(5,–3)=2. For divide-by-zero, the machine signals an error and may issue a BREAK instruction.
Remainder Unsigned (REMU)	Computes the remainder of the division of two unsigned 32-bit values. The machine defines the remainder REM(*i*,*j*) as *i*–(*j**div(*i*,*j*)) where $j \cdot 0$. Remainder (unsigned) treats *src1* as the dividend. The divisor can be *src2* or the immediate value. This instruction divides the contents of *src1* by the contents of *src2*, or it can divide the contents of *src1* by the immediate value. Remainder (unsigned) puts the remainder in the destination register. For divide-by-zero, the machine signals an error and may issue a BREAK instruction.
Rotate Left (ROL)	Rotates the contents of a register left (toward the sign bit). This instruction inserts in the least-significant bit any bits that were shifted out of the sign bit. The contents of *src1* specify the value to shift, and the contents of *src2* (or the immediate value) specify the amount to shift. Rotate Left puts the result in the destination register. If *src2* (or the immediate value) is greater than 31, *src1* shifts by (*src2* MOD 32).
Rotate Right (ROR)	Rotates the contents of a register right (toward the least-significant bit). This instruction inserts in the sign bit any bits that were shifted out of the least-significant bit. The contents of *src1* specify the value to shift, and the contents of *src2* (or the immediate value) specify the amount to shift. Rotate Right puts the result in the destination register. If *src2* (or the immediate value) is greater than 32, *src1* shifts by *src2* MOD 32.

Instruction Name	Description
Set Equal (SEQ)	Compares two 32-bit values. If the contents of *src1* equal the contents of *src2* (or *src1* equals the immediate value) this instruction sets the destination register to one; otherwise, it sets the destination register to zero.
Set Greater Than (SGT)	Compares two signed 32-bit values. If the contents of *src1* are greater than the contents of *src2* (or *src1* is greater than the immediate value), this instruction sets the destination register to one; otherwise, it sets the destination register to zero.
Set Greater/Equal (SGE)	Compares two signed 32-bit values. If the contents of *src1* are greater than or equal to the contents of *src2* (or *src1* is greater than or equal to the immediate value), this instruction sets the destination register to one; otherwise, it sets the destination register to zero.
Set Greater/Equal Unsigned (SGEU)	Compares two unsigned 32-bit values. If the contents of *src1* are greater than or equal to the contents of *src2* (or *src1* is greater than or equal to the immediate value), this instruction sets the destination register to one; otherwise, it sets the destination register to zero.
Set Greater Than Unsigned (SGTU)	Compares two unsigned 32-bit values. If the contents of *src1* are greater than the contents of *src2* (or *src1* is greater than the immediate value), this instruction sets the destination register to one; otherwise, it sets the destination register to zero.
Set Less Than (SLT)	Compares two signed 32-bit values. If the contents of *src1* are less than the contents of *src2* (or *src1* is less than the immediate value), this instruction sets the destination register to one; otherwise, it sets the destination register to zero.
Set Less/Equal (SLE)	Compares two signed 32-bit values. If the contents of *src1* are less than or equal to the contents of *src2* (or *src1* is less than or equal to the immediate value), this instruction sets the destination register to one; otherwise, it sets the destination register to zero.
Set Less/Equal Unsigned (SLEU)	Compares two unsigned 32-bit values. If the contents of *src1* are less than or equal to the contents of *src2* (or *src1* is less than or equal to the immediate value) this instruction sets the destination register to one; otherwise, it sets the destination register to zero.
Set Less Than Unsigned (SLTU)	Compares two unsigned 32-bit values. If the contents of *src1* are less than the contents of *src2* (or *src1* is less than the immediate value), this instruction sets the destination register to one; otherwise, it sets the destination register to zero.
Set Not Equal (SNE)	Compares two 32-bit values. If the contents of *scr1* do not equal the contents of *src2* (or *src1* does not equal the immediate value), this instruction sets the destination register to one; otherwise, it sets the destination register to zero.

Instruction Name	Description
Shift Left Logical (SLL)	Shifts the contents of a register left (toward the sign bit) and inserts zeros at the least-significant bit. The contents of *src1* specify the value to shift, and the contents of *src2* or the immediate value specify the amount to shift. If *src2* (or the immediate value) is greater than 31 or less than 0, *src1* shifts by *src2* MOD 32.
Shift Right Arithmetic (SRA)	Shifts the contents of a register right (toward the least-significant bit) and inserts the sign bit at the most-significant bit. The contents of *src1* specify the value to shift, and the contents of *src2* (or the immediate value) specify the amount to shift. If *src2* (or the immediate value) is greater than 31 or less than 0, *src1* shifts by the result of *src2* MOD 32.
Shift Right Logical (SRL)	Shifts the contents of a register right (toward the least-significant bit) and inserts zeros at the most-significant bit. The contents of *src1* specify the value to shift, and the contents of *src2* (or the immediate value) specify the amount to shift. If *src2* (or the immediate value) is greater than 31 or less than 0, *src1* shifts by the result of *src2* MOD 32.
Subtract with Overflow (SUB)	Computes the twos-complement difference for two signed values. This instruction subtracts the contents of *src2* from the contents of *src1*, or it can subtract the contents of the immediate from the *src1* value. Subtract (with Overflow) puts the result in the destination register. When the true result's sign differs from the destination register's sign, the machine signals an overflow exception.
Subtract without Overflow (SUBU)	Computes the twos-complement difference for two 32-bit values. This instruction subtracts the contents of *src2* from the contents of *src1*, or it can subtract the contents of the immediate from the *src1* value. Subtract (without Overflow) puts the result in the destination register. Overflow exceptions never happen.
Trap if Equal (TEQ)	Compares two 32-bit values. If the contents of *src1* equal the contents of *src2* (or *src1* equals the immediate value), a trap exception occurs.
Trap if Not Equal (TNE)	Compares two 32-bit values. If the contents of *src1* do not equal the contents of *src2* (or *src1* does not equal the immediate value), a trap exception occurs.
Trap if Less Than (TLT)	Compares two signed 32-bit values. If the contents of *src1* are less than the contents of *src2* (or *src1* is less than the immediate value), a trap exception occurs.

Instruction Name	Description
Trap if Less Than Unsigned (TLTU)	Compares two unsigned 32-bit values. If the contents of *src1* are less than the contents of *src2* (or *src1* is less than the immediate value), a trap exception occurs.
Trap if Greater than or Equal (TGE)	Compares two signed 32-bit values. If the contents of *src1* are greater than the contents of *src2* (or *src1* is greater than the immediate value), a trap exception occurs.
Trap if Greater than or Equal Unsigned (TGEU)	Compares two unsigned 32-bit values. If the contents of *src1* are greater than the contents of *src2* (or *src1* is greater than the immediate value), a trap exception occurs.

Table 5-8 Computational Instruction Descriptions for MIPS3/4 Architecture

Instruction Name	Description
Doubleword Absolute Value (DABS)	Computes the absolute value of the contents of *src1*, treated as a 64-bit signed value, and puts the result in the destination register. If the value in *src1* is -2**63, the machine signals an overflow exception.
Doubleword Add with Overflow (DADD)	Computes the twos-complement sum of two 64-bit signed values. The instruction adds the contents of *src1* to the contents of *src2*, or it can add the contents of *src1* to the immediate value. When the result cannot be extended as a 64-bit number, the system signals an overflow exception.
Doubleword Add without Overflow (DADDU)	Computes the twos-complement sum of two 64-bit values. The instruction adds the contents of *src1* to the contents of *src2*, or it can add the contents of *src1* to the immediate value. Overflow exceptions never occur.
Doubleword Divide Signed (DDIV)	Computes the quotient of two 64-bit values. DDIV treats *src1* as the dividend. The divisor can be *src2* or the immediate value. It puts the quotient in the destination register. If the divisor is zero, the system signals an error and may issue a BREAK instruction. The DDIV instruction rounds towards zero. Overflow is signaled when dividing -2**63 by -1. **Note:** The special case DDIV *$0,src1,src2* generates the real doubleword divide instruction and leaves the result in the HI/LO register. The HI register contains the quotient. No checking for divide-by-zero is performed.

Instruction Name	Description
Doubleword Divide Unsigned (DDIVU)	Computes the quotient of two unsigned 64-bit values. DDIVU treats *src1* as the dividend. The divisor can be *src2* or the immediate value. It puts the quotient in the destination register. If the divisor is zero, the system signals an exception and may issue a BREAK instruction. See note for DDIV concerning $0 as a destination. Overflow exceptions never occur.
Doubleword Multiply (DMUL)	Computes the product of two values. This instruction puts the 64-bit product of *src1* and *src2*, or the 64-bit product of *src1* and the immediate value, in the destination register. Overflow is not reported. **Note:** Use DMUL when you do not need overflow protection. It is often faster than DMULO and DMULOU. For multiplication by a constant, the DMUL instruction produces faster machine instruction sequences than DMULT or DMULTU can produce.
Doubleword Multiply (DMULT)	Computes the 128-bit product of two 64-bit signed values. This instruction multiplies the contents of *src1* by the contents of *src2* and puts the result in the HI and LO registers. No overflow is possible. **Note:** The DMULT instruction is a real machine language instruction.
Doubleword Multiply Unsigned (DMULTU)	Computes the product of two unsigned 64-bit values. It multiplies the contents of *src1* and the contents of *src2*, putting the result in the HI and LO registers. No overflow is possible. **Note:** The DMULTU instruction is a real machine language instruction.
Doubleword Multiply with Overflow (DMULO)	Computes the product of two 64-bit signed values. It puts the 64-bit product of *src1* and *src2*, or the 64-bit product of *src1* and the immediate value, in the destination register. When an overflow occurs, the system signals an overflow exception and may execute a BREAK instruction. **Note:** For multiplication by a constant, DMULO produces faster machine instruction sequences than DMULT or DMULTU can produce; however, if you do not need overflow detection, use the DMUL instruction. It is often faster than DMULO.

Instruction Name	Description
Doubleword Multiply with Overflow Unsigned (DMULOU)	Computes the product of two 64-bit unsigned values. It puts the 64-bit product of *src1* and *src2*, or the 64-bit product of *src1* and the immediate value, into the destination register. When an overflow occurs, the system signals an overflow exception and may issue a BREAK instruction. **Note:** For multiplication by a constant, DMULO produces faster machine instruction sequences than DMULT or DMULTU can produce; however, if you do not need overflow detection, use the DMUL instruction. It is often faster than DMULO.
Doubleword Negate with Overflow (DNEG)	Computes the negative of a 64-bit value. The instruction negates the contents of *src1* and puts the result in the destination register. If the value of *src1* is -2**63, the system signals an overflow exception.
Doubleword Negate without Overflow (DNEGU)	Negates the 64-bit contents of *src1* and puts the result in the destination register. Overflow is not reported.
Doubleword Remainder Signed (DREM)	Computes the remainder of the division of two signed 64-bit values. It treats *src1* as the dividend. The divisor can be *src2* or the immediate value. The DREMU instruction puts the remainder in the destination register. If the divisor is zero, the system signals an error and may issue a BREAK instruction.
Doubleword Remainder Unsigned (DREMU)	Computes the remainder of the division of two unsigned 64-bit values. It treats *src1* as the dividend. The divisor can be *src2* or the immediate value. The DREMU instruction puts the remainder in the destination register. If the divisor is zero, the system signals an error and may issue a BREAK instruction.
Doubleword Rotate Left (DROL)	Rotates the contents of a 64-bit register left (towards the sign bit). This instruction inserts in the least-significant bit any bits that were shifted out of the sign bit. The contents of *src1* specify the value to shift, and contents of *src2* (or the immediate value) specify the amount to shift. If *src2* (or the immediate value) is greater than 63, *src1* shifts by *src2* MOD 64.
Doubleword Rotate Right (DROR)	Rotates the contents of a 63-bit register right (towards the least-significant bit). This instruction inserts in the sign bit any bits that were shifted out of the least-significant bit. The contents of *src1* specify the value to shift, and the contents of *src2* (or the immediate value) specify the amount to shift. If *src2* (or the immediate value is greater than 63, *src1* shifts by *src2* MOD 64.

Instruction Name	Description
Doubleword Shift Left Logical (DSLL)	Shifts the contents of a 64-bit register left (towards the sign bit) and inserts zeros at the least-significant bit. The contents of *src1* specify the value to shift, and the contents of *src2* (or the immediate value) specify the amount to shift. If *src2* (or the immediate value) is greater than 63, *src1* shifts by *src2* MOD 64.
Doubleword Shift Right Arithmetic (DSRA)	Shifts the contents of a 64-bit register right (towards the least-significant bit) and inserts the sign bit at the most-significant bit. The contents of *src2* (or the immediate value) specify the amount to shift. If *src2* (or the immediate value) is greater than 63, *src1* shifts by *src2* MOD 64.
Doubleword Shift Right Logical (DSRL)	Shifts the contents of a 64-bit register right (towards the least-significant bit) and inserts zeros at the most-significant bit. The contents of *src1* specify the value to shift, and the contents of *src2* (or the immediate value) specify the amount to shift. If *src2* (or the immediate value) is greater than 63, *src1* shifts by *src2* MOD 64.
Doubleword Subtract with Overflow (DSUB)	Computes the twos-complement difference for two signed 64-bit values. This instruction subtracts the contents of *src2* from the contents of *src1*, or it can subtract the immediate value from the contents of *src1*. It puts the result in the destination register. When the true result's sign differs from the destination register's sign, the system signals an overflow exception.
Doubleword Subtract without Overflow (DSUBU)	Computes the twos complement difference for two unsigned 64-bit values. This instruction subtracts the contents of *src2* from the contents of *src1*, or it can subtract the immediate value from the contents of *src1*. It puts the result in the destination register. Overflow exceptions never happen.

Jump and Branch Instructions

The jump and branch instructions let you change an assembly program's control flow. This section of the book describes jump and branch instructions.

Jump and Branch Instructions

Jump and branch instructions change the flow of a program. Table 5-9, page 51, summarizes the formats of jump and branch instructions.

Table 5-9 Jump and Branch Format Summary

Description	Op-Code	Operand
Jump	J	*address*
Jump and Link	JAL	*address target return,target*
Branch on Equal	BEQ	*src1,src2,label*
Branch on Greater	BGT	*src1,immediate,label*
Branch on Greater/Equal	BGE	
Branch on Greater/Equal Unsigned	BGEU	
Branch on Greater Than Unsigned	BGTU	
Branch on Less Than	BLT	
Branch on Less/Equal	BLE	
Branch on Less/Equal Unsigned	BLEU	
Branch on Less Than Unsigned	BLTU	
Branch on Not Equal	BNE	
Branch	B	*label*
Branch and Link	BAL	
Branch on Equal Likely*	BEQL	*src1,src2,label*
Branch on Greater Than Likely*	BGTL	*src1, immediate,label*
Branch on Greater/Equal Likely*	BGEL	
Branch on Greater/Equal Unsigned Likely*	BGEUL	
Branch on Greater Than Unsigned Likely*	BGTUL	
Branch on Less Than Likely*	BLTL	
Branch on Less/Equal Likely*	BLEL	
Branch on Less/Equal Unsigned Likely*	BLEUL	
Branch on Less Than Unsigned Likely*	BLTUL	
Branch on Not Equal Likely*	BNEL	
Branch on Equal to Zero	BEQZ	*src1,label*

Description	Op-Code	Operand
Branch on Greater/Equal Zero	BGEZ	
Branch on Greater Than Zero	BGTZ	
Branch on Greater or Equal to Zero and Link	BGEZAL	
Branch on Less Than Zero and Link	BLTZAL	
Branch on Less/Equal Zero	BLEZ	
Branch on Less Than Zero	BLTZ	
Branch on Not Equal to Zero	BNEZ	
Branch on Equal to Zero Likely*	BEQZL	*src1,label*
Branch on Greater/Equal Zero Likely*	BGEZL	
Branch on Greater Than Zero Likely*	BGTZL	
Branch on Greater or Equal to Zero and Link Likely*	BGEZALL	
Branch on Less Than Zero and Link Likely*	BLTZALL	
Branch on Less/Equal Zero Likely*	BLEZL	
Branch on Less Than Zero Likely*	BLTZL	
Branch on Not Equal to Zero Likely*	BNEZL	
* not valid on MIPS1 architecture		

Jump and Branch Instruction Descriptions

In Table 5-10, page 52, the branch instructions, branch destinations must be defined in the source being assembled.

Table 5-10 Jump and Branch Instruction Descriptions

Instruction Name	Description
Branch (B)	Branches unconditionally to the specified label.
Branch and Link (BAL)	Branches unconditionally to the specified label and puts the return address in general register $31.

Instruction Name	Description
Branch on Equal (BEQ)	Branches to the specified label when the contents of *src1* equal the contents of *src2*, or when the contents of *src1* equal the immediate value.
Branch on Equal to Zero (BEQZ)	Branches to the specified label when the contents of *src1* equal zero.
Branch on Greater Than (BGT)	Branches to the specified label when the contents of *src1* are greater than the contents of *src2*, or it can branch when the contents of *src1* are greater than the immediate value. The comparison treats the comparands as signed 32-bit values.
Branch on Greater/Equal Unsigned (BGEU)	Branches to the specified label when the contents of *src1* are greater than or equal to the contents of *src2*, or it can branch when the contents of *src1* are greater than or equal to the immediate value. The comparison treats the comparands as unsigned 32-bit values.
Branch on Greater/Equal Zero (BGEZ)	Branches to the specified label when the contents of *src1* are greater than or equal to zero.
Branch on Greater/Equal Zero and Link (BGEZAL)	Branches to the specified label when the contents of *src1* are greater than or equal to zero and puts the return address in general register $31. When this write is done, it destroys the contents of the register. See the MIPS Microprocessor User's Manual appropriate to your architecture for more information. Do **not** use BGEZAL $31.
Branch on Greater or Equal (BGE)	Branches to the specified label when the contents of *src1* are greater than or equal to the contents of *src2*, or it can branch when the contents of *src1* are greater than or equal to the immediate value. The comparison treats the comparands as signed 32-bit values.
Branch on Greater Than Unsigned (BGTU)	Branches to the specified label when the contents of *src1* are greater than the contents of *src2*, or it can branch when the contents of *src1* are greater than the immediate value. The comparison treats the comparands as unsigned 32-bit values.
Branch on Greater Than Zero (BGTZ)	Branches to the specified label when the contents of *src1* are greater than zero.
Branch on Less Than Zero (BLTZ)	Branches to the specified label when the contents of *src1* are less than zero. The program must define the destination.
Branch on Less Than (BLT)	Branches to the specified label when the contents of *src1* are less than the contents of *src2*, or it can branch when the contents of *src1* are less than the immediate value. The comparison treats the comparands as signed 32-bit values.

Instruction Name	Description
Branch on Less/Equal Unsigned (BLEU)	Branches to the specified label when the contents of *src1* are less than or equal to the contents of *src2*, or it can branch when the contents of *src1* are less than or equal to the immediate value. The comparison treats the comparands as unsigned 32-bit values.
Branch on Less/Equal Zero (BLEZ)	Branches to the specified label when the contents of *src1* are less than or equal to zero. The program must define the destination.
Branch on Less or Equal (BLE)	Branches to the specified label when the contents of *src1* are less than or equal to the contents of *src2*, or it can branch when the contents of *src1* are less than or equal to the immediate value. The comparison treats the comparands as signed 32-bit values.
Branch on Less Than Unsigned (BLTU)	Branches to the specified label when the contents of *src1* are less than the contents of *src2*, or it can branch when the contents of *src1* are less than the immediate value. The comparison treats the comparands as unsigned 32-bit values.
Branch on Less Than Zero and Link (BLTZAL)	Branches to the specified label when the contents of *src1* are less than zero and puts the return address in general register $31. Because the value is always stored in register 31, there is a chance of a stored value being overwritten before it is used. See the MIPS microprocessor user's manual appropriate to your architecture for more information. Do **not** use BGEZAL $31
Branch on Not Equal (BNE)	Branches to the specified label when the contents of *src1* do not equal the contents of *src2*, or it can branch when the contents of *src1* do not equal the immediate value.
Branch on Not Equal to Zero (BNEZ)	Branches to the specified label when the contents of *src1* do not equal zero.
Jump (J)	Unconditionally jumps to a specified location. A symbolic address or a general register specifies the destination. The instruction `J $31` returns from a JAL call instruction.

Instruction Name	Description
Jump And Link (JAL)	Unconditionally jumps to a specified location and puts the return address in a general register. A symbolic address or a general register specifies the target location. By default, the return address is placed in register $31. If you specify a pair of registers, the first receives the return address and the second specifies the target. The instruction JAL `procname` transfers to `procname` and saves the return address. For the two-register form of the instruction, the target register may not be the same as the return-address register. For the one-register form, the target may not be `$31`.
Branch Likely Instructions	Same an the ordinary branch instruction (without the "Likely"), except in a branch likely instruction, the instruction in the delay slot is nullified if the conditional branch is not taken.
	Note: The branch likely instructions should be used only inside a `.set noreorder` schedule in an assembly program. The assembler does not attempt to schedule the delay slot of a branch likely instruction.

Special Instructions

The main processor's special instructions do miscellaneous tasks. See Table 5-11.

Special Instruction Descriptions

Table 5-11 Special Instruction Descriptions

Instruction Name	Description
Break (BREAK)	Unconditionally transfers control to the exception handler. The *breakcode* operand is interpreted by software conventions. The *breakcode1* operand is used to fill the low-order 10 bits of the 20-bit immediate field in the BREAK instruction. The optional second operand, *breakcode2*, fills the high-order 10 bits.
Exception Return (ERET)	Returns from an interrupt, exception or error trap. Similar to a branch or jump instruction, ERET executes the next instruction before taking effect. Use this on R4000 processor machines in place of RFE.

Instruction Name	Description
Move From HI Register (MFHI)	Moves the contents of the HI register to a general-purpose register.
Move From LO Register (MFLO)	Moves the contents of the LO register to a general-purpose register.
Move To HI Register (MTHI)	Moves the contents of a general-purpose register to the HI register.
Move To LO Register (MTLO)	Moves the contents of a general-purpose register to the LO register.
Restore From Exception (RFE)	Restores the previous interrupt called and user/kernel state. This instruction can execute only in kernel state and is unavailable in user mode.
Syscall (SYSCALL)	Causes a system call trap. The operating system interprets the information set in registers to determine what system call to do.

Coprocessor Interface Instructions

The coprocessor interface instructions provide standard ways to access your machine's coprocessors. See Table 4-1, page 18 and Table 5-13, page 58.

Coprocessor Interface Summary

Table 5-12 Coprocessor Interface Formats

Description	Op-code	Operand
Load Word Coprocessor *z*	LWC*z*	*dest-copr*,*address*
Load Double Coprocessor *z**	LDC*z*	
Store Word Coprocessor *z*	SWC*z*	*src-copr*, *address*
Store Double Coprocessor *z**	SDC*z*	
Move From Coprocessor *z*	MFC*z*	*dest-gpr*, *source*
Move To Coprocessor *z*	MTC*z*	*src-gpr*, *destination*

Description	Op-code	Operand
Doubleword Move From Coprocessor *z***	DMFC*z*	
Doubleword Move To Coprocessor *z***	DMTC*z*	
Branch Coprocessor *z* False	BC*z*F	*label*
Branch Coprocessor *z* True	BC*z*T	
Branch Coprocessor *z* False Likely*	BC*z*FL	
Branch Coprocessor *z* True Likely*	BC*z*TL	
Coprocessor *z* Operation	C*z*	*expression*
Control From Coprocessor *z*	CFC*z*	*dest-gpr*, *source*
Control To Coprocessor z	CTC*z*	*src-gpr*, *destination*

* not valid on MIPS1 architecture
** not valid on MIPS1 and MIPS2 architectures

Note: You cannot use coprocessor load and store instructions with the system control coprocessor (cp0).

Coprocessor Interface Instruction Descriptions

Table 5-13 Coprocessor Interface Instruction Descriptions

Instruction Name	Description
Branch Coprocessor *z* True (BCzT)	Branches to the specified label when the specified coprocessor asserts a true condition. The z selects one of the coprocessors. A previous coprocessor operation sets the condition.
Branch Coprocessor *z* False (BCzF)	Branches to the specified label when the specified coprocessor asserts a false condition. The z selects one of the coprocessors. A previous coprocessor operation sets the condition.
Branch Coprocessor *z* True Likely (BCzTL)	Branches to the specified label when the specified coprocessor asserts a true condition. If the conditional branch is not taken, the instruction in the branch delay slot is nullified.
	Note: The branch likely instructions should be used only within a `.set noreorder` block. The assembler does not attempt to schedule the delay slot of a branch likely instruction.
Branch Coprocessor *z* False Likely (BCzFL)	Branches to the specified label when the specified coprocessor asserts a false condition. If the conditional branch is not taken, the instruction in the branch delay slot is nullified.
	Note: The branch likely instructions should be used only within a `.set noreorder` block. The assembler does not attempt to schedule the delay slot of a branch likely instruction.
Control From Coprocessor *z* (CFC*z*)	Stores the contents of the coprocessor control register specified by the source in the general register specified by *dest-gpr*.
Control To Coprocessor (CTC*z*)	Stores the contents of the general register specified by *src-gpr* in the coprocessor control register specified by the destination.
Coprocessor *z* Operation (Cz)	Executes a coprocessor-specific operation on the specified coprocessor. The z selects one of four distinct coprocessors.

Instruction Name	Description
Load Word Coprocessor *z* (LWCz)	Loads the destination with the contents of a word that is at the memory location specified by the effective address. The z selects one of four distinct coprocessors. Load Word Coprocessor replaces all register bytes with the contents of the loaded word. If bits 0 and 1 of the effective address are not zero, the machine signals an address exception.
Load Double Coprocessor *z* (LDCz)	Loads a doubleword from the memory location specified by the effective address and makes the data available to coprocessor unit *z*. The manner in which each coprocessor uses the data is defined by the individual coprocessor specifications. This instruction is not valid in MIPS1 architectures. If any of the three least-significant bits of the effective address are non-zero, the machine signals an address error exception.
Move From Coprocessor *z* (MFCz)	Stores the contents of the coprocessor register specified by the source in the general register specified by *dest-gpr*.
Move To Coprocessor *z* (MTCz)	Stores the contents of the general register specified by *src-gpr* in the coprocessor register specified by the *destination*.
Doubleword Move From Coprocessor *z* (DMFCz)	Stores the 64-bit contents of the coprocessor register specified by the source into the general register specified by *dest-gpr*.
Doubleword Move To Coprocessor *z* (DMTCz)	Stores the 64-bit contents of the general register *src-gpr* into the coprocessor register specified by the *destination*.
Store Word Coprocessor *z* (SWCz)	Stores the contents of the coprocessor register in the memory location specified by the effective address. The z selects one of four distinct coprocessors. If bits 0 and 1 of the effective address are not zero, the machine signals an address error exception.
Store Double Coprocessor *z* (SDCz)	Coprocessor z sources a doubleword, which the processor writes the memory location specified by the effective address. The data to be stored is defined by the individual coprocessor specifications. This instruction is not valid in MIPS1 architecture. If any of the three least-significant bits of the effective address are non-zero, the machine signals an address error exception.

Coprocessor Instruction Set

This chapter describes the coprocessor instructions for these coprocessors:

- System control coprocessor (cp0) instructions
- Floating-point coprocessor instructions

See Chapter 5, "The Instruction Set", page 27, for a description of the main processor's instructions and the coprocessor interface instructions.

Instruction Notation

The tables in this chapter list the assembler format for each coprocessor's load, store, computational, jump, branch, and special instructions. The format consists of an op-code and a list of operand formats. The tables list groups of closely related instructions; for those instructions, you can use any op-code with any specified operand.

Note: The system control coprocessor instructions do not have operands.

Operands can have any of these formats:

- Memory references: for example, a relocatable symbol +/– an expression(register)
- Expressions (for immediate values)
- Two or three operands: for example, ADD $3,$4 is the same as ADD $3,$3,$4

The following terms are used to discuss floating-point operations:

- infinite: A value of +1 or –1.
- infinity: A symbolic entity that represents values with magnitudes greater than the largest value in that format.
- ordered: The usual result from a comparison, namely: <,=, or >.
- NaN: Symbolic entities that represent values not otherwise available in floating-point formats. There are two kinds of NaNs. *Quiet NaNs* represent unknown or uninitialized values. *Signaling NaNs* represent symbolic values and

values that are too big or too precise for the format. Signaling NaNs raise an invalid operation exception whenever an operation is attempted on them.

- unordered: The condition that results from a floating-point comparison when one or both operands are NaNs.

Floating-Point Instructions

The floating-point coprocessor has these classes of instructions:

- Load and Store Instructions: Load values and move data between memory and coprocessor registers.
- Move Instructions: Move data between registers.
- Computational Instructions: Do arithmetic and logical operations on values in coprocessor registers.
- Relational Instructions: Compare two floating-point values.

A particular floating-point instruction may be implemented in hardware, software, or a combination of hardware and software.

Floating-Point Formats

The formats for the single- and double-precision floating-point constants are shown in Figure 6-1, page 63:

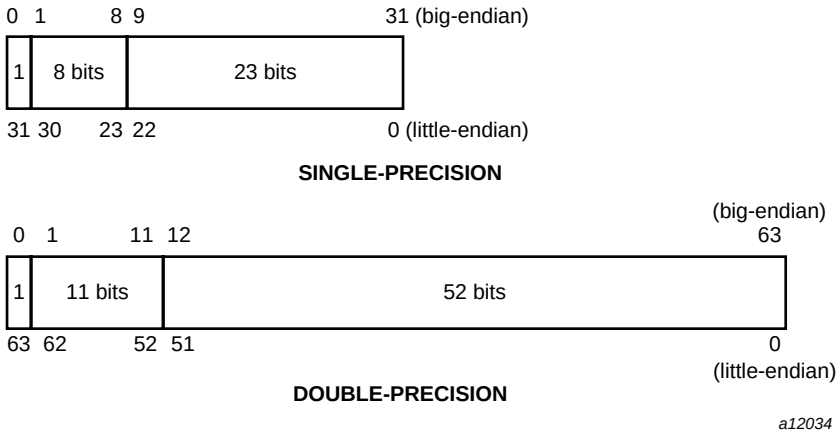

Figure 6-1 Floating Point Formats

Floating-Point Load and Store Formats

Floating-point load and store instructions must use even registers. The operands in the following tables have the following meanings:

Operand	Meaning
address	Offset (base)
destination	Destination register
source	Source register

Table 6-1 Floating-Point Load and Store Formats

Description	Op-Code	Operand
Load Fp		
Double	L.D	*destination, address*
Single	L.S	
Load Indexed Fp		
Double	LDXC1	*destination, index(base)*

Description	Op-Code	Operand
Single	LWXC1	
Load Immediate Fp		
Double	LI.D	*destination, floating-point constant*
Single	LI.S	
Store Fp		
Double	S.D	*source, address*
Single	S.S	
Store Indexed Fp		
Double	SDXC1	*destination, index(base)*
Single	SWXC1	

Floating-Point Load and Store Descriptions

This section groups the instructions by function. See "Floating-Point Instructions", page 62, for the op-codes. Table 6-2, page 64, describes the floating-point Load and Store instructions.

Table 6-2 Floating-Point Load and Store Descriptions

Instruction	Description
Load Fp Instructions	Load eight bytes for double-precision and four bytes for single-precision from the specified effective address into the destination register, which must be an even register (32-bit only) . The bytes must be word aligned. Note: It is recommended that you use doubleword alignment for double-precision operands. It is required in the MIPS2 architecture (R4000 and later).
Load Indexed Fp Instructions	Indexed loads follow the same description as the load instructions above except that indexed loads use *index+base* to specify the effective address (64-bit only).

Instruction	Description
Store Fp Instructions	Stores eight bytes for double-precision and four bytes for single-precision from the source floating-point register in the destination register, which must be an even register (32-bit only). Note: It is recommended that you use doubleword alignment for double-precision operands. It is required in the MIPS2 architecture and later.
Store Indexed Fp Instructions	Indexed stores follow the same description as the store instructions above except that indexed stores use *index+base* to specify the effective address (64-bit only).

Floating-Point Computational Formats

This part of Chapter 6 describes floating-point computational instructions. The operands in Table 6-5, page 70 and Table 6-7, page 75 have the following meaning:

Operand	**Meaning**
destination	Destination register
gpr	General-purpose register
source	Source register

Table 6-3 Floating-Point Computational Instructions

Description	Op-code	Operand
Absolute Value Fp		
Double	ABS.D	*destination*, *src1*
Single	ABS.S	
Negate Fp		
Double	NEG.D	
Single	NEG.S	
Add Fp		
Double	ADD.D	*destination*, *src1*, *src2*
Single	ADD.S	

Description	Op-code	Operand
Divide Fp		
Double	DIV.D	
Single	DIV.S	
Multiply Fp		
Double	MUL.D	
Single	MUL.S	
Subtract Fp		
Double	SUB.D	
Single	SUB.S	
Multiply Add FP		
Double	MADD.D	*destination, src1, src2, src3*
Single	MADD.S	
Negative Multiply Add FP		
Double	NMADD.D	
Single	NMADD.S	
Multiply Subtract FP		
Double	MSUB.D	
Single	MSUB.S	
Negative Multiply Subtract FP		
Double	NMSUB.D	
Single	NMSUB.S	
Convert Source to Specified Fp Precision		
Double to Single Fp	CVT.S.D	*destination, src1*
Fixed Point to Single Fp	CVT.S.W	
Single to Double Fp	CVT.D.S	
Fixed Point to Double Fp	CVT.D.W	
Single to Fixed Point Fp	CVT.W.S	

Description	Op-code	Operand
Double to Fixed Point Fp	CVT.W.D	
Truncate and Round Operations		
Truncate to Single Fp	TRUNC.W.S	*destination*, *src*, *gpr*
Truncate to Double Fp	TRUNC.W.D	
Round to Single Fp	ROUND.W.S	
Round to Double Fp	ROUND.W.D	
Ceiling to Double Fp	CEIL.W.D	
Ceiling to Single Fp	CEIL.W.S	
Ceiling to Double Fp, Unsigned	CEILU.W.D	
Ceiling to Single Fp, Unsigned	CEILU.W.S	
Floor to Double Fp	FLOOR.W.D	
Floor to Single Fp	FLOOR.W.S	
Floor to Double F, Unsigned	FLOORU.W.D	
Floor to Single Fp Unsigned	FLOORU.W.S	
Round to Double Fp Unsigned	ROUNDU.W.D	
Round to Single Fp Unsigned	ROUNDU.W.S	
Truncate to Double Fp Unsigned	TRUNCU.W.D	
Truncate to Single Fp Unsigned	TRUNCU.W.S	
Convert Source to Specified Fp Precision		
Long Fixed Point to Single Fp	CVT.S.L	*destination*, *src1*
Long Fixed Point to Double FP	CVT.D.L	
Single to Long Fixed Point FP	CVT.L.S	
Double to Long Fixed Point FP	CVT.L.D	
Truncate and Round Operations		
Truncate Single to Long Fixed Point	TRUNC.L.S	*destination*, *src*, *gpr*

Description	Op-code	Operand
Truncate Double to Long Fixed Point	TRUNC.L.D	
Round Single to Long Fixed Point	ROUND.L.S	
Round Double to Long Fixed Point	ROUND.L.D	
Ceiling Single to Long Fixed Point	CEIL.L.S	
Ceiling Double to Long Fixed Point	CEIL.L.D	
Floor Single to Long Fixed Point	FLOOR.L.S	
Floor Double to Long Fixed Point	FLOOR.L.D	
Reciprocal Approximation Operations		
Reciprocal Approximation Single Fp	RECIP.S	*destination*, *src1*
Reciprocal Approximation Double Fp	RECIP.D	
Reciprocal Square Root Single Fp	RSQRT.S	
Reciprocal Square Root Double Fp	RSQRT.D	

Floating-Point Computational Instruction Descriptions

This section groups the instructions by function. Refer to Table 6-5, page 70, and Table 6-7, page 75, for the op-code names. Table 6-4 describes the floating-point computational instructions.

Table 6-4 Floating-Point Computational Instruction Descriptions

Instruction	Description
Absolute Value Fp Instructions	Compute the absolute value of the contents of *src1* and put the specified precision floating-point result in the destination register.
Add Fp Instructions	Add the contents of *src1* (or the destination) to the contents of *src2* and put the result in the destination register. When the sum of two operands with opposite signs is exactly zero, the sum has a positive sign for all rounding modes except round toward –1. For that rounding mode, the sum has a negative sign.
Convert Source to Another Precision Fp Instructions	Convert the contents of *src1* to the specified precision, round according to the rounding mode, and put the result in the destination register.
Mutiply-Then-Add Fp Instructions	Multiply the contents of *src2* and *src3*, then add the result to *src1* and store in the destination register (MADD). The NMADD instruction does the same mutiply then add, but then negates the sign of the result (64-bit only).
Mutiply-Then-Subtract Fp Instructions	Multiply the contents of *src2* and *src3*, then subtract *src1* from the product and store in the destination register (MSUB). The NMSUB instruction does the same mutiply then subtract, but then negates the sign of the result (64-bit only).
Truncate and Round instructions	The TRUNC instructions truncate the value in the source floating-point register and put the resulting integer in the destination floating-point register, using the third (general-purpose) register to hold a temporary value. (This is a macro-instruction.) The ROUND instructions work like TRUNC, but round the floating-point value to an integer instead of truncating it.
Divide Fp Instructions	Compute the quotient of two values. These instructions treat *src1* as the dividend and *src2* as the divisor. Divide Fp instructions divide the contents of *src1* by the contents of *src2* and put the result in the destination register. If the divisor is a zero, the machine signals a error if the divide-by-zero exception is enabled.
Multiply Fp Instructions	Multiplies the contents of *src1* (or the destination) with the contents of *src2* and puts the result in the destination register.
Negate FP Instructions	Compute the negative value of the contents of *src1* and put the specified precision floating-point result in the destination register.

Instruction	Description
Subtract Fp Instructions	Subtract the contents of *src2* from the contents of *src1* (or the destination). These instructions put the result in the destination register. When the difference of two operands with the same signs is exactly zero, the difference has a positive sign for all rounding modes except round toward –1. For that rounding mode, the sum has a negative sign.
Reciprocal Approximation Instructions	For RECIP, the reciprocal of the value in *src1* is approximated and placed into the destination register. For RSQRT the reciprocal of the square root of the value in *src1* is approximated and placed into the destination register.

Floating-Point Relational Operations

Table 6-5 summarizes the floating-point relational instructions. The first column under Condition gives a mnemonic for the condition tested. As the "branch on true/false" condition can be used logically to negate any condition, the second column supplies a mnemonic for the logical negation of the condition in the first column. This provides a total of 32 possible conditions. The four columns under Relations give the result of the comparison based on each condition. The final column states if an invalid operation is signaled for each condition.

For example, with an equal condition (EQ mnemonic in the True column), the logical negation of the condition is not equal (NEQ), and a comparison that is equal is True for equal and False for greater than, less than, and unordered, and no Invalid Operation Exception is given if the relation is unordered.

Table 6-5 Floating-Point Relational Operators

Conditions: Mnemonics True	Conditions: Mnemonics False	Relations: Greater Than	Less Than	Equal	Unordered	Invalid Operation Exception if Unordered
F	T	F	F	F	F	no
UN	OR	F	F	F	T	no
EQ	NEQ	F	F	T	F	no

Conditions: Mnemonics True	Conditions: Mnemonics False	Relations: Greater Than	Less Than	Equal	Unordered	Invalid Operation Exception if Unordered
UEQ	OLG	F	F	T	T	no
OLT	UGE	F	T	F	F	no
ULT	OGE	F	T	F	T	no
OLE	UGT	F	T	T	F	no
ULE	OGT	F	T	T	T	no
SF	ST	F	F	F	F	yes
NGLE	GLE	F	F	F	T	yes
SEQ	SNE	F	F	T	F	yes
NGL	GL	F	F	T	T	yes
LT	NLT	F	T	F	F	yes
NGE	GE	F	T	F	T	yes
LE	NLE	F	T	T	F	yes
NGT	GT	F	T	T	T	yes

The mnemonics found in Table 6-5 have following meanings:

Mnemonic	Meaning	Mnemonic	Meaning
F	False	T	True
UN	Unordered	OR	Ordered
EQ	Equal	NEQ	Not Equal
UEQ	Unordered or Equal	OLG	Ordered or Less than or Greater than
OLT	Ordered Less Than	UGE	Unordered or Greater than or Equal
ULT	Unordered or Less Than	OGE	Ordered Greater than or Equal
OLE	Ordered Less than or Equal	UGT	Unordered or Greater Than
ULE	Unorderd or Less than or Equal	OGT	Ordered Greater Than

Mnemonic	Meaning	Mnemonic	Meaning
SF	Signaling False	ST	Signaling True
NGLE	Not Greater Than or Less Than or Equal	GLE	Greater Than, or Less Than or Equal
SEQ	Signaling Equal	SNE	Signaling Not Equal
NGL	Not Greater Than or Less Than	GL	Greater Than or Less Less Than
LT	Less Than	NLT	Not Less Than
NGE	Not Greater Than	GE	Greater Than or Equal or Equal
LE	Less Than or Equal	NLE	Not Less Than or Equal
NGT	Not Greater Than	GT	Greater Than

To branch on the result of a relational:

```
/* branching on a compare result */

c.eq.s $fcc0,$f1,$f12  /* compare the single-precision values */
bc1t $fcc0, true       /* if $f1 equals $f2, branch to true */
bc1f $fcc0, false      /* if $f1 does not equal $f2, branch */
                       /* to false */
```

Floating-Point Relational Instruction Formats

The following are the floating-point relational instruction formats.

Table 6-6 Floating-Point Relational Instruction Formats

Description	Op-code	Operand
Compare F		
Double	C.F.D	*src1,src2*
Single	C.F.S	
Compare UN		
Double	C.UN.D	

Description	Op-code	Operand
Single	C.UN.S	
***Compare EQ**		
Double	C.EQ.D	
Single	C.EQ.S	
Compare UEQ		
Double	C.UEQ.D	
Single	C.UEQ.S	
Compare OLT		
Double	C.OLT.D	
Single	C>OLT.S	
Compare ULT		
Double	C.ULT.D	
Single	C.ULT.S	
Compare OLE		
Double	C.OLE.D	
Single	C.OLE.S	
Compare ULE		
Double	C.ULE.D	
Single	C.ULE.S	
Compare SF		
Double	C.SF.D	
Single	C.SF.S	
Compare NGLE		
Double	C.NGLE.D	*src1, src2*
Single	C.NGLE.S	
Compare SEQ		
Double	C.SEQ.D	

Description	Op-code	Operand
Single	C.SEQ.S	
Compare NGL		
Double	C.NGL.D	
Single	C.NGL.S	
***Compare LT**		
Double	C.LT.D	
Single	C.LT.S	
Compare NGE		
Double	C.NGE.D	
Single	C.NGE.S	
***Compare LE**		
Double	C.LE.D	
Single	C.LE.S	
Compare NGT		
Double	C.NGT.D	
Single	C.NGT.S	

Note: These are the most common Compare instructions. The MIPS coprocessor instruction set provides others for IEEE compatibility.

Floating-Point Relational Instruction Descriptions

This section describes the relational instruction descriptions by function. Refer to Chapter 1 for information regarding registers.

Table 6-7 Floating-Point Relational Instruction Descriptions

Instruction	Description
Compare EQ Instructions	Compare the contents of *src1* with the contents of *src2*. If *src1* equals *src2* a true condition results; otherwise, a false condition results. The machine does not signal an exception for unordered values.
Compare F Instructions	Compare the contents of *src1* with the contents of *src2*. These instructions always produce a false condition. The machine does not signal an exception for unordered values.
Compare LE	Compare the contents of *src1* with the contents of *src2*. If *src1* is less than or equal to *src2*, a true condition results; otherwise, a false condition results. The machine signals an exception for unordered values.
Compare LT	Compare the contents of *src1* with the contents of *src2*. If *src1* is less than *src2*, a true condition results; otherwise, a false condition results. The machine signals an exception for unordered values.
Compare NGE	Compare the contents of *src1* with the contents of *src2*. If *src1* is less than *src2* (or the contents are unordered), a true condition results; otherwise, a false condition results. The machine signals an exception for unordered values.
Compare NGL	Compare the contents of *src1* with the contents of *src2*. If *src1* equals *src2* or the contents are unordered, a true condition results; otherwise, a false condition results. The machine signals an exception for unordered values.
Compare NGLE	Compare the contents of *src1* with the contents of *src2*. If *src1* is unordered, a true condition results; otherwise, a false condition results. The machine signals an exception for unordered values.
Compare NGT	Compare the contents of *src1* with the contents of *src2*. If *src1* is less than or equal to *src2* or the contents are unordered, a true condition results; otherwise, a false condition results. The machine signals an exception for unordered values.
Compare OLE Instructions	Compare the contents of *src1* with the contents of *src2*. If *src1* is less than or equal to *src2*, a true condition results; otherwise, a false condition results. The machine does not signal an exception for unordered values.

Instruction	Description
Compare OLT Instructions	Compare the contents of *src1* with the contents of *src2*. If *src1* is less than *src2*, a true condition results; otherwise, a false condition results. The machine does not signal an exception for unordered values.
Compare SEQ Instructions	Compare the contents of *src1* with the contents of *src2*. If *src1* equals *src2*, a true condition results; otherwise, a false condition results. The machine signals an exception for unordered values.
Compare SF Instructions	Compare the contents of *src1* with the contents of *src2*. This always produces a false condition. The machine signals an exception for unordered values.
Compare ULE Instructions	Compare the contents of *src1* with the contents of *src2*. If *src1* is less than or equal to *src2* (or *src1* is unordered), a true condition results; otherwise, a false condition results. The machine does not signal an exception for unordered values.
Compare UEQ Instructions	Compare the contents of *src1* with the contents of *src2*. If *src1* equals *src2* (or *src1* and *src2* are unordered), a true condition results; otherwise, a false condition results. The machine does not signal an exception for unordered values.
Compare ULT Instructions	Compare the contents of *src1* with the contents of *src2*. If *src1* is less than *src2* (or the contents are unordered), a true condition results; otherwise, a false condition results. The machine does not signal an exception for unordered values.
Compare UN Instructions	Compare the contents of *src1* with the contents of *src2*. If either *src1* or *src2* is unordered, a true condition results; otherwise, a false condition results. The machine does not signal an exception for unordered values.

Floating-Point Move Formats

The floating-point **move** instructions move data from source to destination registers (only floating-point registers are allowed).

Description	Op-code	Operand
Move FP		
Single	MOV.S	*destination*,*src1*
Double	MOV.D	
Move Conditional on FP False	MOVF	*gpr_dest, gpr_src, cc*
Move Conditional on FP True	MOVT	*gpr_dest, gpr_src, cc*
Floating-Point Move Conditional on FP False		
Single	MOVF.S	*destination, src1, cc*
Double	MOVF.D	
Floating-Point Move Conditional on FP True		
Single	MOVT.S	*destination, src1, cc*
Double	MOVT.D	
Floating-Point Move Conditional on Not Zero		
Single	MOVN.S	*destination, src1, gpr*
Double	MOVN.D	
Floating-Point Move Conditional on Zero		
Single	MOVZ.S	*gpr_destination, gpr_src1, gpr*
Double	MOVZ.D	

Floating-Point Move Instruction Descriptions

This section describes the floating-point move instructions.

Table 6-8 Floating-Point Move Instruction Descriptions

Instruction	Description
Move FP Instructions	Move the double or single-precision contents of *src1* to the destination register, maintaining the specified precision.
Conditional Move Instructions	Move the general-purpose register, *src1*, to the destination register if the condition code (*cc*) is zero (MOVF) or is one (MOVT).
Conditonal FP Move Instructions	Conditionally, move the double-precision or single-precision contents of *src1* to the destination register, maintaining the specified precision.
Floating-Point Conditonal Move Instructions	Conditionally, move a floating-point value from *src1* to the destination register if the *gpr_register* is zero (MOVZ) or not equal to zero (MOVN).

System Control Coprocessor Instructions

The system control coprocessor (cp0) handles all functions and special and privileged registers for the virtual memory and exception handling subsystems. The system control coprocessor translates addresses from a large virtual address space into the machine's physical memory space. The coprocessor uses a translation lookaside buffer (TLB) to translate virtual addresses to physical addresses.

System Control Coprocessor Instruction Formats

These coprocessor system control instructions do not have operands.

Description	**Op-code**
Cache (not valid in MIPS1 and MIPS2 architectures)	CACHE
Translation Lookaside Buffer Probe	TLBP
Translation Lookaside Buffer Read	TLBR
Translation Lookaside Buffer Write Random	TLBWR
Translation Lookaside Write Index	TLBWI

Synchronize (Not valid in MIPS1 architectures)	SYNC

System Control Coprocessor Instruction Descriptions

This section describes the system control coprocessor instructions.

Table 6-9 System Control Coprocessor Instruction Descriptions

Instruction	Description
Cache (CACHE)	Cache is the R4000 instruction to perform cache operations. The 16-bit offset is sign-extended and added to the contents of general register base to form a virtual address. The virtual address is translated to a physical address using the TLB. The 5-bit sub-opcode ("op") specifies the cache operation for that address. Part of the virtual address is used to specify the cache block for the operation. Possible operations include invalidating a cache block, writeback to a secondary cache or memory, etc. **Note:** This instruction is not valid in MIPS1 or MIPS2 architectures.
Translation Lookaside Buffer Probe (TLBP)	Probes the translation lookaside buffer (TLB) to see if the TLB has an entry that matches the contents of the EntryHi register. If a match occurs, the machine loads the Index register with the number of the entry that matches the EntryHi register. If no TLB entry matches, the machine sets the high-order bit of the Index register.
Translation Lookaside Buffer Read (TLBR)	Loads the EntryHi and EntryLo registers with the contents of the translation lookaside buffer (TLB) entry specified in the TLB Index register.
Translation Lookaside BufferWrite Random (TLBWR)	Loads the specified translation lookaside buffer (TLB) entry with the contents of the EntryHi and EntryLo registers. The contents of the TLB Random register specify the TLB entry to be loaded.
Translation Lookaside Buffer Write Index (TLBWI)	Loads the specified translation lookaside buffer (TLB) entry with the contents of the EntryHI and EntryLO registers. The contents of the TLB Index register specify the TLB entry to be loaded.
Synchronize (SYNC)	Ensures that all loads and stores fetched before the sync are completed, before allowing any following loads or stores. Use of sync to serialize certain memory references may be required in multiprocessor environments. **Note:** This instruction is not valid in the MIPS1 architecture.

Control and Status Register

Floating-point coprocessor control register 31 contains status and control information. See Figure 6-2. It controls the arithmetic rounding mode and the enabling of user-level traps, and indicates exceptions that occurred in the most recently executed instruction, and any exceptions that may have occurred without being trapped:

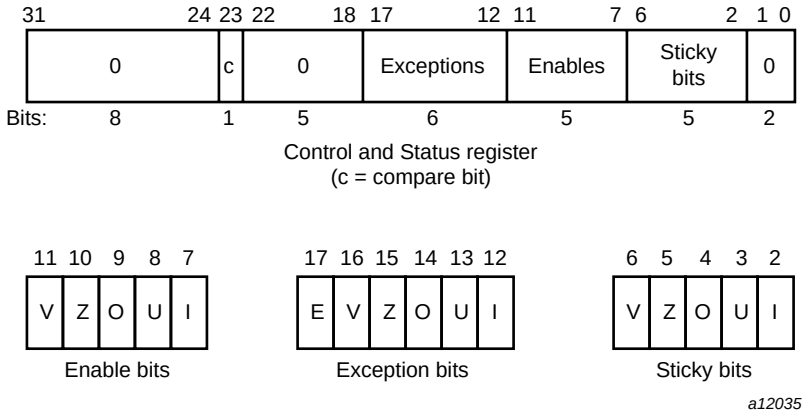

Figure 6-2 Floating Control and Status Register 31

The exception bits are set for instructions that cause an IEEE standard exception or an optional exception used to emulate some of the more hardware-intensive features of the IEEE standard.

The exception field is loaded as a side-effect of each floating-point operation (excluding loads, stores, and unformatted moves). The exceptions which were caused by the immediately previous floating-point operation can be determined by reading the exception field.

The meaning of each bit in the exception field is given below. If two exceptions occur together on one instruction, the field will contain the inclusive-OR of the bits for each exception:

Exception Field Bit	Description
E	Unimplemented Operation
I	Inexact Exception

O	Overflow Exception
U	Underflow Exception
V	Invalid Operation
Z	Division-by-Zero

The unimplemented operation exception is normally invisible to user-level code. It is provided to maintain IEEE compatibility for non-standard implementations.

The five IEEE standard exceptions are listed below:

Field	**Description**
I	Inexact Exception
O	Overflow Exception
U	Underflow Exception
V	Invalid Operation
Z	Division-by-Zero

Each of the five exceptions is associated with a trap under user control, which is enabled by setting one of the five bits of the enable field, shown above.

When an exception occurs, both the corresponding exception and status bits are set. If the corresponding enable flag bit is set, a trap is taken. In some cases the result of an operation is different if a trap is enabled.

The status flags are never cleared as a side effect of floating-point operations, but may be set or cleared by writing a new value into the status register, using a “move to coprocessor control” instruction.

The floating-point compare instruction places the condition which was detected into the`c’ bit of the control and status register, so that the state of the condition line may be saved and restored. The`c’ bit is set if the condition is true, and cleared if the condition is false, and is affected only by compare and move to control register instructions.

Exception Trap Processing

For each IEEE standard exception, a status flag is provided that is set on any occurrence of the corresponding exception condition with no corresponding exception trap signaled. It may be reset by writing a new value into the status register. The flags may be saved and restored individually, or as a group, by software. When no

exception trap is signaled, a default action is taken by the floating-point coprocessor, which provides a substitute value for the original, exceptional, result of the floating-point operation. The default action taken depends on the type of exception, and in the case of the Overflow exception, the current rounding mode.

Invalid Operation Exception

The invalid operation exception is signaled if one or both of the operands are invalid for an implemented operation. The result, when the exception occurs without a trap, is a quiet NaN when the destination has a floating-point format, and is indeterminate if the result has a fixed-point format. The invalid operations are:

- Addition or subtraction: magnitude subtraction of infinities, such as (+ 1) – (– 1).
- Multiplication: 0 times 1, with any signs.
- Division: 0 over 0 or 1 over 1, with any signs.
- Square root of x: where x is less than zero.
- Conversion of a floating-point number to a fixed-point format when an overflow, or operand value of infinity or NaN, precludes a faithful representation in that format.
- Comparison of predicates involving < or > without ?, when the operands are "unordered".
- Any operation on a signaling NaN.

Software may simulate this exception for other operations that are invalid for the given source operands. Examples of these operations include IEEE-specified functions implemented in software, such as Remainder: *x* REM *y*, where *y* is zero or *x* is infinite; conversion of a floating-point number to a decimal format whose value causes and overflow or is infinity of NaN; and transcendental functions, such as ln (–5) or $\cos^{-1}(3)$.

Division-by-zero Exception

The division by zero exception is signaled on an implemented divide operation if the divisor is zero and the dividend is a finite nonzero number. The result, when no trap occurs, is a correctly signed infinity.

If division by zero traps are enabled, the result register is not modified, and the source registers are preserved.

Software may simulate this exception for other operations that produce a signed infinity, such as ln(0), sec(p/2), csc(0) or 0^{-1}.

Overflow Exception

The overflow exception is signaled when what would have been the magnitude of the rounded floating-point result, were the exponent range unbounded, is larger than the destination format's largest finite number. The result, when no trap occurs, is determined by the rounding mode and the sign of the intermediate result.

If overflow traps are enabled, the result register is not modified, and the source registers are preserved.

Underflow Exception

Two related events contribute to underflow. One is the creation of a tiny non-zero result between 2^{Emin} (minimum expressible exponent) which, because it is tiny, may cause some other exception later. The other is extraordinary loss of accuracy during the approximation of such tiny numbers by denormalized numbers.

The IEEE standard permits a choice in how these events are detected, but requires that they must be detected the same way for all operations.

The IEEE standard specifies that "tininess" may be detected either: "after rounding" (when a nonzero result computed as though the exponent range were unbounded would lie strictly between 2^{Emin}), or "before rounding" (when a nonzero result computed as though the exponent range and the precision were unbounded would lie strictly between 2^{Emin}). The architecture requires that tininess be detected after rounding.

Loss of accuracy may be detected as either "denormalization loss" (when the delivered result differs from what would have been computed if the exponent range were unbounded), or "inexact result" (when the delivered result differs from what would have been computed if the exponent range and precision were both unbounded). The architecture requires that loss of accuracy be detected as inexact result.

When an underflow trap is not enabled, underflow is signaled (via the underflow flag) only when both tininess and loss of accuracy have been detected. The delivered result might be zero, denormalized, or 2^{Emin}. When an underflow trap is enabled, underflow is signaled when tininess is detected regardless of loss of accuracy.

If underflow traps are enabled, the result register is not modified, and the source registers are preserved.

Inexact Exception

If the rounded result of an operation is not exact or if it overflows without an overflow trap, then the inexact exception is signaled. The rounded or overflowed result is delivered to the destination register, when no inexact trap occurs. If inexact exception traps are enabled, the result register is not modified, and the source registers are preserved.

Unimplemented Operation Exception

If an operation is specified that the hardware may not perform, due to an implementation restriction on the supported operations or supported formats, an unimplemented operation exception may be signaled, which always causes a trap, for which there are no corresponding enable or flag bits. The trap cannot be disabled.

This exception is raised at the execution of the unimplemented instruction. The instruction may be emulated in software, possibly using implemented floating-point unit instructions to accomplish the emulation. Normal instruction execution may then be restarted.

This exception is also raised when an attempt is made to execute an instruction with an operation code or format code which has been reserved for future architectural definition. The unimplemented instruction trap is not optional, since the current definition contains codes of this kind.

This exception may be signaled when unusual operands or result conditions are detected, for which the implemented hardware cannot handle the condition properly. These may include (but are not limited to), denormalized operands or results, NaN operands, trapped overflow or underflow conditions. The use of this exception for such conditions is optional.

Floating-Point Rounding

Bits 0 and 1 of the coprocessor control register 31 sets the rounding mode for floating-point. The machine allows four rounding modes:

- **Round to nearest** rounds the result to the nearest representable value. When the two nearest representable values are equally near, this mode rounds to the value with the least significant bit zero. To select this mode, set bits 1..0 of control register 31 to 0.

- **Round toward zero** rounds toward zero. It rounds to the value that is closest to and not greater in magnitude than the infinitely precise result. To select this mode, set bits 1..0 of control register 31 to 1.
- **Round toward positive infinity** rounds to the value that is closest to and not less than the infinitely precise result. To select this mode, set bits 1..0 of control register 31 to 2.
- **Round toward negative infinity** rounds toward negative infinity. It rounds to the value that is closest to and not greater than the infinitely precise result. To select this mode, set bits 1..0 of control register 31 to 3.

To set the rounding mode:

```
/* setting the rounding mode */
RoundNearest = Ox0
RoundZero = Ox1
RoundPosInf = Ox2
RoundNegInf = Ox3
      cfc1 rt2, $31          # move from coprocessor 1
      and rt, Oxfffffffc     # zero the round mode bits
      or rt, RoundZero       # set mask as round to zero
      ctc1 rt, $f31          # move to coprocessor 1
```

Writing Assembly Language Code

This chapter gives rules and examples to follow when designing an assembly language program. The chapter includes a tutorial section that contains information about how calling sequences work. This involves writing a skeleton version of your prospective assembly routine using a high-level language, and then compiling it with the –S option to generate a human-readable assembly language file. The assembly language file can then be used as the starting point for coding your routine. See "Using the .s Assembly Language File", page 100 for details about the assembly language file produced with this option.

This assembler works in either 32-bit, high performance 32-bit (N32) or 64-bit compilation modes. While these modes are very similar, due to the difference in data, register and address sizes, the N32 and 64-bit assembler linkage conventions are not always the same as those for 32-bit mode. For details on some of these differences, see the *MIPSpro 64-Bit Porting and Transition Guide* and the *MIPSpro N32 ABI Handbook*.

The procedures and examples in this chapter, for the most part, describe 32-bit compilation mode. In some cases, specific differences necessitated by 64-bit mode are highlighted.

Introduction

When you write assembly language routines, you should follow the same calling conventions that the compilers observe, for two reasons:

- Often your code must interact with compiler-generated code, accepting and returning arguments or accessing shared global data.
- The symbolic debugger gives better assistance in debugging programs using standard calling conventions.

The conventions for the compiler system are a bit more complicated than some, mostly to enhance the speed of each procedure call. Specifically:

- The compilers use the full, general calling sequence only when necessary; where possible, they omit unneeded portions of it. For example, the compilers don't use a register as a frame pointer whenever possible.

- The compilers and debugger observe certain implicit rules rather than communicating via instructions or data at execution time. For example, the debugger looks at information placed in the symbol table by a ".frame" directive at compilation time, so that it can tolerate the lack of a register containing a frame pointer at execution time.

Program Design

This section describes some general areas of concern to the assembly language programmer:

- Stack frame requirements on entering and exiting a routine.
- The "shape" of data (scalars, arrays, records, sets) laid out by the various high-level languages.

For information about register format, and general, special, and floating-point registeres, see Chapter 1.

The Stack Frame

This discussion of the stack frame, particularly regarding the graphics, describes 32-bit operations. In 32-bit mode, restrictions such as stack addressing are enforced strictly. While these restrictions are not enforced rigidly for 64-bit stack frame usage, their observance is probably still a good coding practice, especially if you count on reliable debugging information.

The compilers classify each routine into one of the following categories:

- Non-leaf routines, that is, routines that call other procedures.
- Leaf routines, that is, routines that do not themselves execute any procedure calls. Leaf routines are of two types:
  - Leaf routines that require stack storage for local variables
  - Leaf routines that do not require stack storage for local variables.

You must decide the routine category before determining the calling sequence.

To write a program with proper stack frame usage and debugging capabilities, use the following procedure:

1. Regardless of the type of routine, you should include a .ent pseudo-op and an entry label for the procedure. The .ent pseudo-op is for use by the debugger, and the entry label is the procedure name. The syntax is:

```
     .ent    procedure_name
procedure_name:
```

2. If you are writing a leaf procedure that does not use the stack, skip to step 3. For leaf procedure that uses the stack or non-leaf procedures, you must allocate all the stack space that the routine requires. The syntax to adjust the stack size is:

```
     subu    $sp,framesize
```

   where framesize is the size of frame required; framesize must be a multiple of 16. Space must be allocated for:

- Local variables.
- Saved general registers. Space should be allocated only for those registers saved. For non-leaf procedures, you must save `$31`, which is used in the calls to other procedures from this routine. If you use registers `$16--$23`, you must also save them.
- Saved floating-point registers. Space should be allocated only for those registers saved. If you use registers `$f20--$f30` (for 32-bit) or `$f24--$f31` (for 64-bit), you must also save them.
- Procedure call argument area. You must allocate the maximum number of bytes for arguments of any procedure that you call from this routine.

Note: Once you have modified `$sp`, you should not modify it again for the rest of the routine.

---

3. Now include a `.frame` pseudo-op:

```
     .frame framereg,framesize,returnreg
```

   The virtual frame pointer is a frame pointer as used in other compiler systems but has no register allocated for it. It consists of the *framereg* (`$sp`, in most cases) added to the framesize (see step 2 above). The following figures show the stack components for `-32` and `-n32` and `-64`.

The *returnreg* specifies the register containing the return address (usually `$31`). These usual values may change if you use a varying stack pointer or are specifying a kernel trap routine.

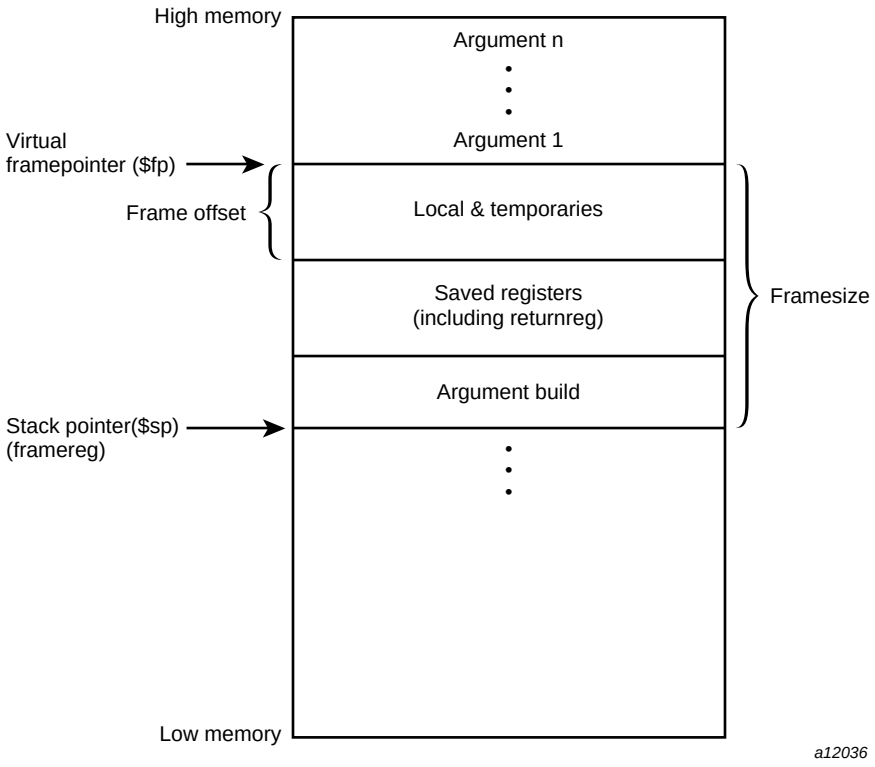

Figure 7-1 Stack Organization for `-32`

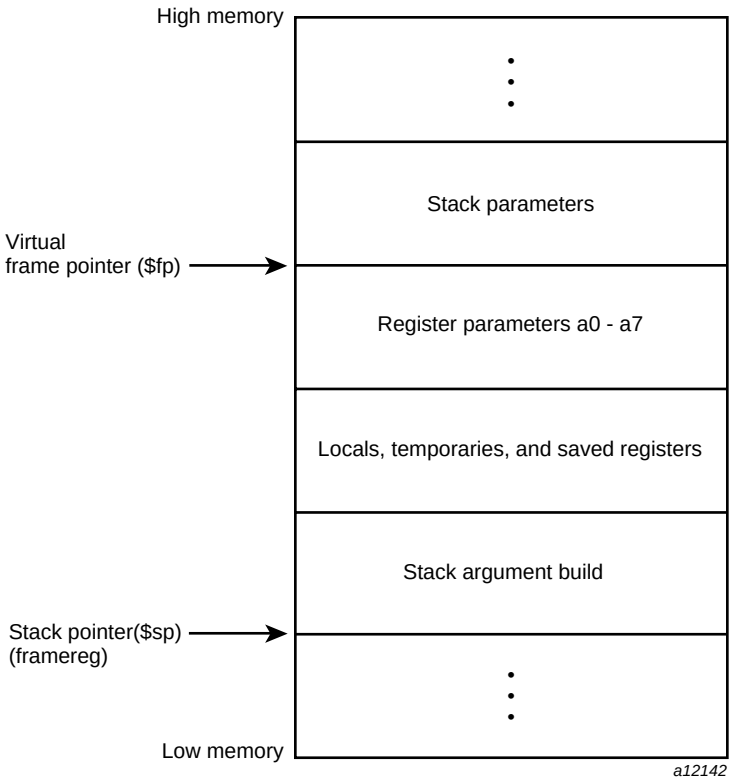

Figure 7-2 Stack Organization for `-n32` and `-64`

4. If the procedure is a leaf procedure that does not use the stack, skip to step 7. Otherwise you must save the registers you allocated space for in step 2.

   To save the general registers, use the following operations:

   ```
   .mask    bitmask,frameoffset
   sw reg,framesize+frameoffset-N($sp)
   ```

   The `.mask` directive specifies the registers to be stored and where they are stored. A bit should be on in bitmask for each register saved (for example, if register `$31` is saved, bit 31 should be '1' in bitmask. Bits are set in bitmask in little-endian order, even if the machine configuration is big-endian).The *frameoffset* is the offset from the virtual frame pointer (this number is usually negative). *N* should be 0

for the highest numbered register saved and then incremented by four for each subsequently lower numbered register saved. For example:

```
sw    $31,framesize+frameoffset($sp)
sw    $17,framesize+frameoffset-4($sp)
sw    $16,framesize+frameoffset-16($sp)
```

Figure 7-3, page 93, illustrates this example.

Now save any floating-point registers that you allocated space for in step 2 as follows:

```
.fmask    bitmask,frameoffset
s.[sd]    reg,framesize+frameoffset-N($sp)
```

Notice that saving floating-point registers is identical to saving general registers except we use the `.fmask` pseudo-op instead of `.mask`, and the stores are of floating-point singles or doubles.The discussion regarding saving general registers applies here as well, but remember that N should be incremented by 16 for doubles.The stack framesize must be a multiple of 16.

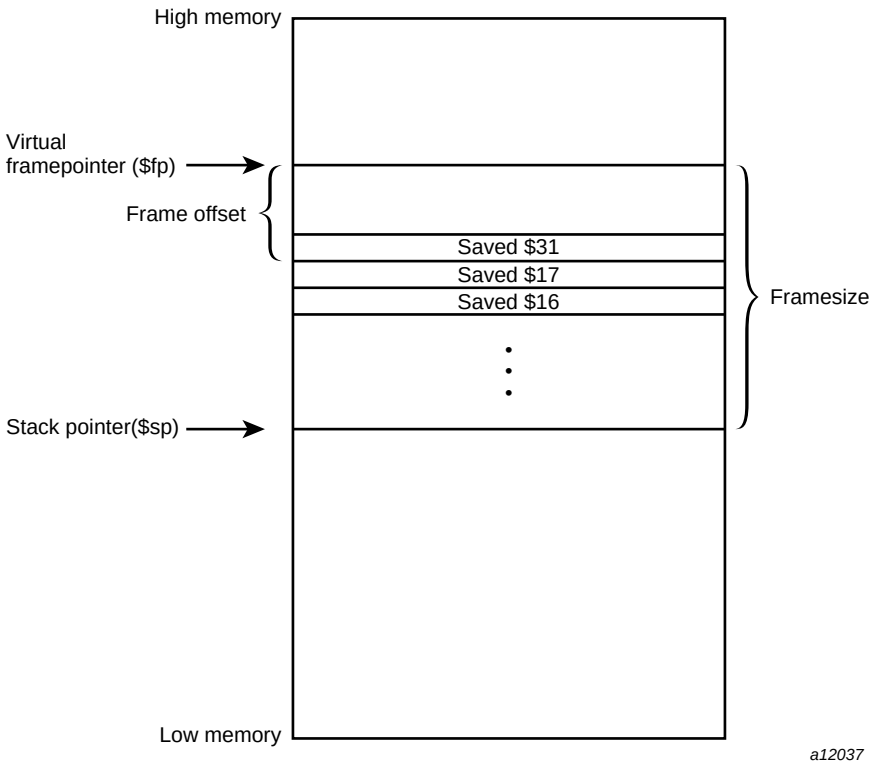

Figure 7-3 Stack Example

5. This step describes parameter passing: how to access arguments passed into your routine and passing arguments correctly to other procedures. For information on high-level language-specific constructs (call-by-name, call-by-value, string or structure passing), refer to the *MIPSpro N32/64 Compiling and Performance Tuning Guide*.

   As specified in step 2, space must be allocated on the stack for all arguments even though they may be passed in registers. This provides a saving area if their registers are needed for other variables.

   General registers must be used for passing arguments. For 32-bit compilations, general registers `$4`–`$7` and float registers `$f12`, `$f14` are used for passing the first four arguments (if possible). You must allocate a pair of registers (even if it's

a single precision argument) that start with an even register for floating-point arguments appearing in registers.

For 64-bit compilations, general registers `$4`–`$11` and float registers `$f12`, through `$f19` are used for passing the first eight arguments (if possible).

In Table 7-1 and Table 7-2, the "f*N*" arguments are considered single- and double-precision floating-point arguments, and "n*N*" arguments are everything else. The ellipses (...) mean that the rest of the arguments do not go in registers regardless of their type. The "stack" assignment means that you do not put this argument in a register. The register assignments occur in the order shown in order to satisfy optimizing compiler protocols:

Table 7-1 Parameter Passing (`-32`)

Argument List	Register and Stack Assignments
f1, f2	$f12, $f14
f1, n1, f2	$f12, $6, stack
f1, n1, n2	$f12, $6 $7
n1, n2, n3, n4	$4, $5, $6, $7
n1, n2, n3, f1	$4, $5, $6, stack
n1, n2, f1	$4, $5, ($6, $6)
n1, f1	$4, ($6, $7)

Table 7-2 Parameter Passing (`-n32` and `-64`)

Argument List	Register and Stack Assignments
d1,d2	$f12, $f13
s1,s2	$f12, $f13
s1,d1	$f12, $f13
d1,s1	$f12, $f13
n1,d1	$4,$f13

Argument List	Register and Stack Assignments
d1,n1,d1	$f12, $5,$f14
n1,n2,d1	$4, $5,$f14
d1,n1,n2	$f12, $5,$6
s1,n1,n2	$f12, $5,$6
d1,s1,s2	$f12, $f13, $f14
s1,s2,d1	$f12, $f13, $f14
n1,n2,n3,n4	$4,$5,$6,$7
n1,n2,n3,d1	$4,$5,$6,$f15
n1,n2,n3,s1	$4,$5,$6, $f15
s1,s2,s3,s4	$f12, $f13,$f14,$f15
s1,n1,s2,n2	$f12, $5,$f14,$7
n1,s1,n2,s2	$4,$f13,$6,$f15
n1,s1,n2,n3	$4,$f13,$6,$7
d1,d2,d3,d4,d5	$f12, $f13, $f14, $f15, $f16
d1,d2,d3,d4,d5,s1,s2,s3,s4	$f12, $f13, $f14, $f15, $f16, $f17, $f18,$f19,stack
d1,d2,d3,s1,s2,s3,n1,n2,n3	$f12, $f13, $f14, $f15, $f16, $f17, $10,$11, stack

6. Next, you must restore registers that were saved in step 4. To restore general purpose registers:

   ```
   lw reg,framesize+frameoffset-N($sp)
   ```

   To restore the floating-point registers:

   ```
   l.[sd] reg,framesize+frameoffset-N($sp)
   ```

   Refer to step 4 for a discussion of the value of *N*.)

7. Get the return address:

   ```
   lw $31,framesize+frameoffset($sp)
   ```

8. Clean up the stack:

   ```
   addu framesize
   ```

9. Return:

```
j $31
```

10. To end the procedure:

```
.end procedurename
```

The difference in stack frame usage for `-n32` and `-64` operations can be summarized as follows:

The portion of the argument structure beyond the initial eight doublewords is passed in memory on the stack, pointed to by the stack pointer at the time of call. The caller does not reserve space for the register arguments; the callee is responsible for reserving it if required (either adjacent to any caller-saved stack arguments if required, or elsewhere as appropriate). No requirement is placed on the callee either to allocate space and save the register parameters, or to save them in any particular place.

The Shape of Data

In most cases, high-level language routine and assembly routines communicate via simple variables: pointers, integers, booleans, and single- and double-precision real numbers. Describing the details of the various high-level data structures (arrays, records, sets, and so on) is beyond the scope of this book. If you need to access such a structure as an argument or as a shared global variable, refer to the *MIPSpro N32/64 Compiling and Performance Tuning Guide*.

Examples

This section contains the examples that illustrate program design rules. Each example shows a procedure written and C and its equivalent written in assembly language.

Example 7-1 Non-leaf procedure

The following example shows a non-leaf procedure. Notice that it creates a stackframe, and also saves its return address since it must put a new return address into register $31 when it invokes its callee:

```
float
nonleaf(int i, int *j)
     {
     double atof();
```

```
      int temp;

      temp = i - *j;
      if (i < *j) temp = -temp;
      return atof(temp);
      }
            .globl       nonleaf
   #   1    float
   #   2    nonleaf(i, j)
   #   3    int i, *j;
   #   4    {
            .ent        nonleaf 2
   nonleaf;
            .cpload $25              ## Load $gp
            subu        $sp, 32      ## Create stackframe
            sw          $31, 20($sp) ## Save the return
                                     ## address
            sw          $sp, 24($sp) ## Save gp
            .mask       0x80000000, -4
            .frame    $sp, 32, $31
   #  5    double atof();
   #  6    int temp;
   #  7
   #  8    temp = i - *j;
            lw      $2, 0($5)        ## Arguments are in
                                     ## $4 and $5
            subu     $3, $4, $2
   #  9    if (i < *j) temp = -temp;
            bge      $4, $2, $32     ## Note: $32 is a label,
                                     ##  not a reg
            negu     $3, $3
$32:
   #  10   return atof(temp);
            move     $4, $3
            jal      atof
            cvt.s.   $f0, $f0       ## Return value goes in $f0
            lw       $gp, 24($sp)   ## Restore gp
            lw       $31, 20($sp)   ## Restore return address
            addu     $sp, 32        ## Delete stackframe
            j        $31            ## Return to caller
            .end     nonleaf
```

The `-n32` code for the previous example is shown below. Note that this code is under `.set noreorder`, so be aware of delay slots.

```
  .set          noreorder
        # Program Unit: nonleaf
  .ent          nonleaf
  .globl        nonleaf
nonleaf:        # 0x0
  .frame        $sp, 32, $31
  .mask         0x80000000, -32
  lw $7,0($5)             # load *j
  addiu $sp,$sp,-32       #.frame.len.nonleaf
  sd $gp,8($sp)           # save $gp
  sd $31,0($sp)           # save $ra
  lui $31,%hi(%neg(%gp_rel(nonleaf+0))) #load new $gp
  addiu $31,$31,%lo(%neg(%gp_rel(nonleaf +0))) #
  addu $gp,$25,$31        #
  slt $1,$4,$7            # compare i to *j
  beq $1,$0,.L.1.1.temp   #
  subu $7,$4,$7           # i-*j, in delay slot of branch
  subu $7,$0,$7           # temp = -temp
.L.1.1.temp:     # 0x2c
  lw $25,%call16(atof)($gp)#
  jalr $25                #atof
  or $4,$7,$0             # delay slot of jalr loads arg
  ld $31,0($sp)           # restore $ra
  cvt.s.d $f0,$f0         #
  ld $tp,8($sp)           # restore $gp
  jr $31                  #
  addiu $sp,$sp,32        # .frame.len.nonleaf
  .end   nonleaf
```

Example 7-2 Leaf Procedure

This example shows a leaf procedure that does not require stack space for local variables. Notice that it creates no stackframe, and saves no return address.

```
int
leaf(p1, p2)
    int p1, p2;
```

```
    {
    return (p1 > p2) ? p1 : p2;
    }
                .globl        leaf
   #    1       int
   #    2       leaf(p1, p2)
   #    3         int p1, p2;
   #    4         {
                .ent          leaf2
leaf:
                .frame        $sp, 0, $31
   #    5         return (p1 > p2) ? p1 : p2;
                 ble          $4, $5, $32    ## Arguments in
                                             ##  $4 and $5
                 move         $3, $4
                 b            $33
$32:
                 move         $3, $5
$33:
                 move         $2, $3         ## Return value
                                             ##  goes in $2
                 j            $31            ## Return to
                                             ##  caller
   #    6          }
                 .end    leaf
```

The `-n32` code for the previous example looks like this:

```
   .set    noreorder
   .ent    leaf
   .globl  leaf
leaf:   #0x0
   .fram$sp, 0, $31
   slt $2,$5,$4           # compare p1 and p2
   beq $2, $0,.L.1.2.temp #
   or $9,$4,$0            # delay slot
   b .L.1.1.temp          #
   or $2,$9,$0            # delay slot, return pl
.L.1.2.temp:  # 0x14
   or $2,$5,$0            # return p2
.L.1.1.temp:  # 0x18
   jr $31                 #
```

```
nop                     # delay slot
.end     leaf
```

Interfaces Between Assembly Routines and Other Languages

The rules and parameter requirements that exist between assembly language and other languages are varied and complex. The simplest approach to coding an interface between an assembly routine and a routine written in a high-level language is to do the following:

- Use the high-level language to write a skeletal version of the routine that you plan to code in assembly language.
- Compile the program using the –S option, which creates an assembly language (`.s`) version of the compiled source file (the `-O` option, though not required, reduces the amount of code generated, making the listing easier to read).
- Study the assembly-language listing and then, imitating the rules and conventions used by the compiler, write your assembly language code.

Using the `.s` Assembly Language File

The MIPSpro compilers can produce a `.s` file rather than a `.o` file. The file is produced by specifying the `-S` option on the command line instead of the `-c` option.

The assembly language file that is produced contains exactly the same set of instructions that would have been produced in the `.o` object file, and inputting the `.s` file to the assembler produces an object file with the same instructions that the compiler would have produced. The `.s` file is a listing of the instructions, but does not contain all the object information that a `.o` file contains. Therefore, a .o file generated by a `.s` file will not be exactly the same as one generated directly by the compiler and they are not guaranteed to work identically (for example, `reorg_common` information is lost).

In addition to the program's instructions, the `.s` file contains comments indicating the effects of various optimization transformations that were made by the compiler.

Most of these comments are self-explanatory or contain easily understood information, while other comments require a detailed knowledge of the compiler's internal workings. The following information is intended to describe the more useful,

non-obvious, features of the file without getting into the details of optimization theory. For more detailed information about optimization see the *MIPSpro N32/64 Compiling and Performance Tuning Guide*.

The following subsections describe the different elements of the `.s` file.

Program Header

The file begins with comments that indicate the name of the source file and the compiler that was used to produce the `.s` file. The options that were used by the compiler are also listed. It is often important to know the target machine that the instructions were intended for; this is discussed in the following subsections. By default, only a select set of options are included in the file. More detail can be obtained by including the `-LIST:options` flag on the compiler's command line.

Instruction Alignment

One of the first pseudo-instructions in the file is similar to the following example:

```
.section .text, 1, 0x00000006, 4, 64
```

or

```
.section .text, 1, 0x00000006, 4, 16
```

This directive is used by the loader to align the start of the program's instructions at particular byte-address boundaries. The rightmost field is `16` if quad word alignment is required, or is `64` if cache line alignment is needed. The proper number is determined by the target processor type and the optimization level that was used because some optimizations require an exact knowledge of the I-Cache placement of each instruction while others do not benefit from this level of control.

Label Offset Comments

A comment is attached to each label definition (recognized by the colon (`:`) following the name). This comment provides the byte offset of the label's location relative to the start of the `.section .text` directive. The first label, which usually corresponds to the first entry point of the first function, is `0x0`.

The remaining labels have addresses that are increased by 4 bytes for each instruction that is placed between successive labels. These offsets are the same for both the `.s`

file and the related `.o` files, although the loader can choose to place the start of the program (the `0x0` location) anywhere in the machine's address space. The start is subject only to the alignment restriction placed on the `.section` directive (see "Instruction Alignment", page 101).

This is useful to note when you are using a debugger and trying to correlate the assembly file to the executed instructions. The machine addresses are sometimes difficult to translate to the file's relative offsets when only quad word alignment was requested.

The following is an example of this comment:

```
.BB1.kernel_:  # 0x0
```

Source Code Comments

To help associate the compiler-generated code to the original source code, the line number and source code are inserted into comment lines that are interspersed with the assembly instructions. The comments usually appear ahead of the machine instructions that are generated for it. However, various optimizations may cause instructions to be moved or reordered and it is sometimes difficult to understand where they appear.

A further difficulty can arise if inline code expansion occurs. In these cases, the line number (`503` in the following example) may refer to the line of the module that contained the inlined routine, and not to the original source code module of the compiled program. This can be especially confusing if the `-ipa` option was requested, and if several source code files were intermixed.

To determine the original file that contains a particular source code line, search for the immediately preceding `.loc` directive. This directive contains the line number and an index to a previous `.file` directive that identifies the file that the source code was read from. See Chapter 8, "Pseudo Op-Codes (Directives)", page 109 for information about the `.loc` directive.

The following is an example of this comment:

```
# 503              x[k] = q + y[k]*( r*z[k+10] + t*z[k+11] )
```

Relative Instruction Issue Times

When any level of optimization greater than `-O0` is requested on the command line, comments are added to the right of machine instructions that indicate the compiler's knowledge of the relative issue time for the particular instruction. These comments consist of an integer between square brackets, as shown in the following example:

```
mul.d $f2,$f2,$f10              # [11]
```

In this example, the `[11]` indicates the clock cycle (relative to the start of the block) in which this instruction will be issued by the processor.

The assembly files targeted for processors that can only issue a single instruction in a clock period have unique times for each instruction in the block, while target processors that can issue multiple instructions may show that several instructions have the same integer in the issue time comment.

The times for processors that support Out-Of-Order issue of instructions may sometimes appear unusual because an instruction may be issued before other instructions that precede it in the block. This is common processor behavior. The compiler attempts to model the queuing mechanisms contained by the hardware and it uses knowledge of the details to arrive at meaningful times to place in these comment fields. The times are accurate to the limit that the machine is modeled.

Several simplifying assumptions are made to calculate these times, which make it difficult to estimate the performance of the code by using these comments alone. The most important point to make is that program flow is not taken into account. The actual performance of a program is influenced by the path taken into a particular block of code, which often determines when the inputs that an instruction needs will be ready. It would be difficult to model an entire program and take into account all possible paths into a block, so it is assumed that all inputs computed outside a block are available at the start of the block, and that all functional units are initially free to accept new operations.

Even with these restrictions, it is difficult to accurately model the behavior of load and store instructions. The compiler attempts to recognize accesses that will be satisfied from a data cache and use an appropriate latency. Although performance data suggests that most data references are to a cache, this can be very program-dependent. With the additional complexity introduced when multiple levels of cache are available, the compiler can never be certain that it is using the correct memory latency to produce the issue time comments. Because of these uncertainties, the compiler uses times that match what happens in the average program.

A limitation on the use of these times is illustrated with the following program example run on a machine with an R10000 processor:

```
.Lt.0.224:      # 0x508
        .loc    1 589 17
# 589                   temp -= x[j]*y[j];
         ldc1 $f9,24024($3)             # [0]
         ldc1 $f10,32064($5)            # [1]
         addiu $2,$2,-1                 # [0]
         addiu $3,$3,8                  # [0]
         addiu $5,$5,8                  # [1]
         bne $2,$0,.Lt.0.224            # [1,1]
         nmsub.d $f4,$f4,$f9,$f10       # [4]
```

With just a glance at the times, you might conclude that a `nmsub` instruction will only be issued every 5 clock periods. However, as long as execution stays within the loop, the processor will prefetch instructions faster than it can execute them, resulting in an average issue of 1 `nmsub` instruction every 2 clock periods, limited by the 2 memory accesses that take 2 clock periods to issue.

There are occasions when the first instruction is not considered to be part of the block and no instruction issue time is computed for it. This happens when the block is frequently branched to using a `label+4` address specification. The following example code illustrates this:

```
.Lt.0.274:      # 0x9ac
         or $3,$9,$0                    #
         or $6,$10,$0                   # [0]
```

The most frequent transfer to the label is the following instruction:

```
     bne $8,$30,.Lt.0.274+4                  # [0,1]
```

Relative Branch Prediction Times

If the target processor is an Out-Of-Order processor, the cycle when the hardware will predict the direction of a conditional branch is estimated. This happens at the time the instruction is first read into the instruction decode buffer and is independent of the time that the instruction actually issues.

This time is reported as the first of a pair of integers, in square brackets, in the comment field of the instruction. The second field is the issue time. In the preceding

example, branch prediction happens in cycle 0, but the instruction will not issue until cycle 1 (because it has to wait for an input).

The compiler attempts to move inputs to conditional branches as far previously as possible so that both the branch prediction and the issue times are identical. However, there are conditions that prevent the compiler from doing so; this is done to minimize the number of instructions that are speculatively executed after the prediction and before the direction of the branch can be determined with certainty. It is only when the instruction completes execution that the hardware is certain which branch direction is correct.

If the wrong direction was predicted, all speculatively executed instructions will need to be aborted, wasting time that could have been devoted to completing the program.

nop Instructions

A `nop` instruction is a real operation that does not change the contents of any registers. There are several that could be used, but the preferred one is `sll $0,$0,0`, which means "shift left by 0 bits the contents of register `$0` and store the result into register `$0`".

`nop` instructions usually waste space and should be deleted by the compiler, but there are situations where they are necessary for the correctness of the executed code and cases where they can improve the performance of the executed code. They are most often encountered as a placeholder for the delay slot of a branch instruction, when no other instruction can be found. The following code sequence illustrates this:

```
addiu $5,$5,1                   # [0]
bne $5,$30,.Lt.0.460            # [0,1]
nop                             # [0]
```

Other than their use in the delay slot of conditional branches, `nop` instructions are used to optimize the fetch and decode performance of processor types that can read, decode and execute multiple instructions in each clock period. These processors cannot group together instructions when a cache line boundary occurs between them, resulting in a delay that can be avoided by inserting one or more `nop` instructions ahead of a label.

The optimization that attempts this alignment depends on the processor type and the optimization levels selected. In the common case, the first block of each loop is forced to start on a quad word boundary. This is simple and fast although it sometimes causes `nop`s to be added in the middle of a cache line, where they are not useful.

For the highest level of optimization, and only for Out-Of-Order issue processors, closer track is kept of cache line boundaries. This requires that the start of the module (that is, the address of the first text label) be aligned on a cache line boundary, increasing the size of the generated executable but allowing the compiler to avoid unnecessary instructions.

Along with optimally aligning instructions on Out-Of-Order processors, attention is paid to a timing "hiccup" that can occur if a branch instruction is separated from its delay slot instruction by a cache line break. The insertion of a `nop` before the branch can improve performance slightly. The following is an example of this. The `nop` forces the `bne` instruction to start in the next cache line, as can be determined by the address comment in the label field of the next block.

```
        nop                             # [1]
        bne $0,$1,.Lt.0.550             # [3,5]
        xori $1,$1,1                    # [5]
.BB307.kernel_: # 0x2408
```

Loop Information Comments

Comments are added at the start of loops to indicate the transformations that were applied to the loop. The meanings of most of these are obvious, but some need some explanation:

- The occurrence of comments that start with `<swpf>` or `<loop>Not unrolled:` indicate that software pipelining failed to optimize the loop. There is usually a reason given, although the meaning can be obscure and refer to details of software theory.
- Comments that look like `<swps> xx cycles per iteration` may not contain an accurate count of the number of cycles for Out-Of-Order processors. This is because the exact cycle times are determined much later in the compiler process than when this cycle count is estimated and the comment is constructed. These inaccuracies also affect the numbers that precede `% of peak` comments.
- Similarly, for Out-Of-Order processors, the cycle count in comments similar to `<sched> Loop schedule length:  xxx cycles (ignoring nested loops)` is sometimes wrong.

Block Information

A block is a sequence of instructions between 2 labels. Blocks are usually identified in the assembly file by a comment between the starting label and the first instruction with a comment that contains `BB:xxx`. The block number that follows the `BB:` is used to identify each unique block of the program. Comments that start with `<freq> BB:xxx frequency = yyy.yy` indicate how often the compiler believes the block is executed for each invocation of the function where the block is located.

The comment is followed by `(heuristic)` or `(feedback)` to indicate how that average was arrived at. Because many optimizations utilize this information, incorrect information can result in sub-optimal compiler output. It is important that the feedback data be generated by tests that truly represent the expected behavior of the final program so that accurate decisions can be made by the compiler.

Blocks that end with conditional branches also contain comments similar to `<freq> BB:xxx => BB:yyy probability = 0.zzzzz`. These indicate the compiler's estimation for the direction of each possible branch. Again, it is important for optimal performance that feedback be generated by test cases that are representative of the actual workload.

Pseudo Op-Codes (Directives)

This chapter describes pseudo op-codes (directives). These pseudo op-codes influence the assembler's later behavior.

Op-Codes

The assembler has the following pseudo op-codes:

Table 8-1 Pseudo Op-Codes

Pseudo-Op	Description
.2byte *expression1* [, *expression2*]... [, *expressionN*]	Truncates the expressions in the comma-separated list to 16-bit values and assembles the values in successive locations. The expressions must be absolute or in the form of a label difference (*label1* - *label2*) if both labels are defined in the same section. This directive optionally can have the form *expression1* [: *expression2*]. The *expression2* replicates *expression1*'s value *expression2* times. This directive does no automatic alignment. (64–bit and N32 only)
.4byte *expression1* [, *expression2*]... [, *expressionN*]	Truncates the expressions in the comma-separated list to 32-bit values and assembles the values in successive locations. The expressions must be absolute or in the form of a label difference (*label1* - *label2*) if both labels are defined in the same section. This directive optionally can have the form *expression1* [: *expression2*]. The *expression2* replicates *expression1*'s value *expression2* times. This directive does no automatic alignment. (64-bit and N32 only)

.8byte *expression1* [, *expression2*]... [, *expressionN*]	Truncates the expressions in the comma-separated list to 64-bit values and assembles the values in successive locations. The expressions must be absolute or in the form of a label difference (*label1* - *label2*) if both labels are defined in the same section. This directive optionally can have the form *expression1* [: *expression2*]. The *expression2* replicates *expression1*'s value *expression2* times. This directive does no automatic alignment. (64-bit and N32 only)
.aent *name,* symno	Sets an alternate entry point for the current procedure. Use this information when you want to generate information for the debugger. It must appear inside an .ent/.end pair.
.align *expression*	Advances the location counter to make the expression low order bits of the counter zero. Normally, the .half, .word, .float, and .double directives automatically align their data appropriately. For example, .word does an implicit .align 2 (.double does an .align 3). You disable the automatic alignment feature with .align 0. The assembler reinstates automatic alignment at the next .text, .data, .rdata, or .sdata directive. Labels immediately preceding an automatic or explicit alignment are also realigned. For example, foo: .align 3; .word 0 is the same as .align 3; foo: .word 0.
.ascii *string* [, *string*]...	Assembles each string from the list into successive locations. The .ascii directive does not null pad the string. You **must** put quotation marks (") around each string. You can use the backslash escape characters. For a list of the backslash characters, see Chapter 4, "Lexical Conventions", page 15.
.asciiz *string* [, *string*]...	Assembles each string in the list into successive locations and adds a null. You can use the backslash escape characters. For a list of the backslash characters, see Chapter 4, "Lexical Conventions", page 15.

.byte *expression1* [, *expression2*]... [, *expressionN*]	Truncates the expressions from the comma-separated list to 8-bit values, and assembles the values in successive locations. The expressions must be absolute. The operands can optionally have the form: *expression1* [: *expression2*]. The *expression2* replicates *expression1*'s value *expression2* times.
.comm *name*, *expression* [*alignment*]	Unless defined elsewhere, *name* becomes a global common symbol at the head of a block of expression bytes of storage. The linker overlays like-named common blocks, using the maximum of the *expressions*. The 64-bit and N32 assembler also accepts an optional value which specifies the alignment of the symbol.
.cpadd *reg*	Emits code that adds the value of "_gp" to reg.
.cpload *reg*	Expands into the three instructions function prologue that sets up the $gp register. This directive is used by position-independent code.
.cplocal *reg*	Causes the assembler to use *reg* instead of *$gp* as the context pointer. This directive is used by position-independent code. (64-bit and N32 only)
.cprestore *offset*	Causes the assembler to emit the following at the point where it occurs: `sw $gp, offset ($sp)` Also, causes the assembler to generate `lw $gp, offset ($sp)` after every JAL or BAL operation. Offset should point to the saved register area as described in Chapter 7, "Writing Assembly Language Code", page 87. This directive is used by position-independent code following the caller-saved gp convention.
.cpreturn	Causes the assembler to emit the following at the point where it occurs: `ld $gp, offset ($sp)` The *offset* is obtained from the previous .cpsetup pseudo-op.

	(64-bit and N32 only)

.cpsetup *reg1*, {*offset* | *reg2*}, *label*	Causes the assembler to emit the following at the point where it occurs:
	`sd $gp, offset ($sp)`
	`lui $gp, 0 {label}`
	`daddiu $gp, $gp, 0 { label }`
	`daddu $gp, $gp, reg1`
	`ld $gp, offset ($sp)`
	This sequence is used by position-independent code following the callee-saved `gp` convention. It stores `$gp` in the saved register area and calculates the virtual address of *label* and places it in *reg1*. By convention, *reg1* is `$25` (`t9`).
	If *reg2* is used instead of offset, `$gp` is saved and restored to and from this register.
	(64-bit and N32 only)

`.data`	Tells the assembler to add all subsequent data to the data section.
.double *expression* [, *expression2*] ...[, *expressionN*]	Initializes memory to 64-bit floating point numbers. The operands optionally can have the form: *expression1* [: *expression2*]. The *expression1* is the floating point value. The optional *expression2* is a non-negative expression that specifies a repetition count. The *expression2* replicates *expression1*'s value *expression2* times. This directive aligns its data and any preceding labels automatically to a double-word boundary. You can disable this feature by using `.align0`.
`.dword` *expression* [, *expression2*]...[, *expressionN*]	Truncates the expressions in the comma-separated list to 64-bits and assembles the values in successive locations. The expressions must be absolute. The operands optionally can have the form: *expression1* [:*expression2*]. The *expresssion2* replicates *expression1*'s value *expression2* number of times. The directive aligns its data and preceding labels automatically to a

	doubleword boundary. You can disable this feature by using .align0.
.dynsym *name value*	Specifies an ELF st_other value for the object denoted by *name* (64-bit and N32 only).
.end [*proc_name*]	Sets the end of a procedure. Use this directive when you want to generate information for the debugger. To set the beginning of a procedure, see .ent.
.endr	Signals the end of a repeat block. To start a repeat block, see .repeat.
.ent *proc_name*	Sets the beginning of the procedure *proc_name*. Use this directive when you want to generate information for the debugger. To set the end of a procedure, see .end.
.extern *name expression*	*name* is a global undefined symbol whose size is assumed to be *expression* bytes. The advantage of using this directive, instead of permitting an undefined symbol to become global by default, is that the assembler can decide whether to use the economical $gp-relative addressing mode, depending on the value of the –G option. As a special case, if expression is zero, the assembler refrains from using $gp to address this symbol regardless of the size specified by –G.
.file *file_number file_name_string*	Specifies the source file corresponding to the assembly instructions that follow. For use only by compilers, not by programmers; when the assembler sees this, it refrains from generating line numbers for dbx to use unless it also sees .loc directives.
.float *expression1* [, *expression*2]... [, *expressionN*]	Initializes memory to single precision 32-bit floating point numbers. The operands optionally can have the form: *expression1*<_newline>[: *expression2*]. The optional *expression2* is a non-negative expression that specifies a repetition count. This optional form replicates *expression1*'s value *expression2* times. This directive aligns its data and preceding labels automatically to a word boundary. You can disable this feature by using .align0.
.fmask *mask offset*	Sets a mask with a bit turned on for each floating point register that the current routine saved. The

	least-significant bit corresponds to register `$f0`. The offset is the distance in bytes from the virtual frame pointer at which the floating point registers are saved. The assembler saves higher register numbers closer to the virtual frame pointer. You must use `.ent` before `.fmask` and only one `.fmask` may be used per `.ent`. Space should be allocated for those registers specified in the `.fmask`.
`.frame` *frame-register offset return_pc_register*	Describes a stack frame. The first register is the frame-register, the offset is the distance from the frame register to the virtual frame pointer, and the second register is the return program counter (or, if the first register is `$0`, this directive shows that the return program counter is saved four bytes from the virtual frame pointer). You must use .`ent` before .`frame` and only one .`frame` may be used per .`ent`. No stack traces can be done in the debugger without .`frame`.
.`globl` *name*	Makes the *name* external. If the *name* is defined otherwise (by its appearance as a label), the assembler will export the symbol; otherwise it will import the symbol. In general, the assembler imports undefined symbols (that is, it gives them the UNIX storage class "global undefined" and requires the linker to resolve them).
`.gpvalue` *number*	Sets the offset to use in gp_rel relocations; 0 by default. (64–bit and N32 only).
`.gpword` *local-sym*	This directive is similar to `.word` except that the relocation entry for *local-sym* has the `R_MIPS_GPREL32` type. After linkage, this results in a 32-bit value that is the distance between *local-sym* and `gp`. *local-sym* must be local. This directive is used by the code generator for PIC switch tables.
.`half` *expression1* [, *expression2*]... {, *expressionN*]	Truncates the expressions in the comma-separated list to 16-bit values and assembles the values in successive locations. The *expressions* must be absolute. This directive optionally can have the form: *expression1* [: *expression2*]. The *expression2* replicates *expression1*'s value *expression2* times. This directive automatically

	aligns its data appropriately. You can disable this feature by using `.align0`.
`.lab` *label_name*	Associates a named label with the current location in the program text. For use by compilers.
`.lcomm` *name, expression*	Makes the *name*'s data type `bss`. The assembler allocates the named symbol to the `bss` area, and the *expression* defines the named symbol's length. If a `.globl` directive also specifies the name, the assembler allocates the named symbol to external `bss`. The assembler puts `bss` symbols in one of two `bss` areas. If the defined size is smaller than (or equal to) the size specified by the assembler or compiler's `–G` command line option, the assembler puts the symbols in the `sbss` area and uses `$gp` to address the data.
`.loc` *file_number line_number [column]*	Specifies the source file and the line within that file that corresponds to the assembly instructions that follow. For use by compilers. The assembler ignores the file number when this directive appears in the assembly source file. Then, the assembler assumes that the directive refers to the most recent `.file` directive. The 64-bit and N32 assembler also supports an optional value that specifies the column number.
`.mask` *mask, offset*	Sets a mask with a bit turned on for each general purpose register that the current routine saved. For use by compilers. Bit one corresponds to register `$1`. The offset is the distance in bytes from the virtual frame pointer where the registers are saved. The assembler saves higher register numbers closer to the virtual frame pointer. Space should be allocated for those registers appearing in the mask. If bit zero is set it is assumed that space is allocated for all 31 registers regardless of whether they appear in the mask. For N32/64, only bit 31 for `$ra` is utilized; the rest are ignored.
`.nada`	Tells the assembler to put in an instruction that has no effect on the machine state. It has the same effect as `nop` (described below), but it produces more efficient code on an R8000.

(64-bit and N32 only)

`.nop`

Tells the assembler to put in an instruction that has no effect on the machine state. While several instructions cause no-operation, the assembler only considers the ones generated by the `nop` directive to be wait instructions. This directive puts an explicit delay in the instruction stream.

Note: Unless you use `.set noreorder`, the reorganizer may eliminate unnecessary `nop` instructions.

`.option` *options*

Tells the assembler that certain options were in effect during compilation. (These options can, for example, limit the assembler's freedom to perform branch optimizations.) This *option* is intended for compiler-generated `.s` files rather than for hand-coded ones.

`.origin` *expression*

Specifies the current offset in a section to the value of *expression*.

(64-bit and N32 only)

`.repeat` *expression*

Repeats all instructions or data between the `.repeat` directive and the `.endr` directive. The expression defines how many times the data repeats. With the `.repeat` directive, you cannot use labels, branch instructions, or values that require relocation in the block. To end a `.repeat`, see `.endr`.

`.restore` *reg*

The following instruction restores the value in *reg*. This is used to provide correct information for stack unwinding. There must be an earlier `.save` *reg* that this is following.

(64-bit and N32 only)

`.rdata`

Tells the assembler to add subsequent data into the `rdata` section.

`.save` *reg*	The following instruction saves the value in *reg*. This is used to provide correct information for stack unwinding. (64-bit and N32 only)
`.sdata`	Tells the assembler to add subsequent data to the `sdata` section.
`.section` *name* [, *section type, section flags, section entry size, section alignment*]	Instructs the assembler to create a section with the given name and optional attributes. Legal *section type* values are denoted by variables prefixed by `SHT_ in` <*elf.h*>. Legal *section flags* values are denoted by variables prefixed by `SHF_ in` <*elf.h*>. The *section entry size* specifies the size of each entry in the section. For example, it is 4 for `.text` sections. The *section alignment* specifies the byte boundary requirement for the section. For example, it is 16 for `.text` sections. (64-bit and N32 only)
`.set` *option*	Instructs the assembler to enable or to disable certain options. Use `.set` options only for hand-crafted assembly routines. The assembler has these default options: `reorder`, `macro`, and `at`. You can specify only one option for each `.set` directive. You can specify these `.set` options: The `reorder` option lets the assembler reorder machine language instructions to improve performance. The `noreorder` option prevents the assembler from reordering machine language instructions. If a machine language instruction violates the hardware pipeline constraints, the assembler issues a warning message. The `macro` option lets the assembler generate multiple machine instructions from a single assembler instruction.

The `nomacro` option causes the assembler to print a warning whenever an assembler operation generates more than one machine language instruction. You must select the `noreorder` option before using the `nomacro` option; otherwise, an error results.

The `at` option lets the assembler use the $at register for macros, but generates warnings if the source program uses `$at`. When you use the `noat` option and an assembler operation requires the `$at` register, the assembler issues a warning message; however, the `noat` option does let source programs use `$at` without issuing warnings.

The `nomove` option tells the assembler to mark each subsequent instruction so that it cannot be moved during reorganization. Because the assembler can still insert `nop` instructions where necessary for pipeline constraints, this option is less stringent than `noreorder`. The assembler can still move instructions from below the nomove region to fill delay slots above the region or vice versa. The `nomove` option has part of the effect of the "volatile" C declaration; it prevents otherwise independent loads or stores from occurring in a different order than intended.

The `move` option cancels the effect of `nomove`.

The `notransform` option tells the assembler to mark each subsequent instruction so that it cannot be transformed by `pixie`(1) into an equivalent set of instructions. There are restrictions on the use of this option in order to guarantee `pixie`'s ability to still produce code that will execute correctly with the preceding/following transformed code. The sequence of instructions marked `notransform` must behave like a single basic block (i.e., there is only one entry and exit from the sequences and they are the first and last instructions, respectively). If this restriction cannot be met, correct transformed execution can still be guaranteed if the sequence of instructions does not use any of the saved registers `$16..$23, $30` (s0–s8).

	The `transform` option cancels the effect of `notransform`.
`.size` *name, expression*	Specifies the size of an object denoted by *name* to the value of *expression.*
`.space` *expression*	Advances the location counter by the value of the specified *expression* bytes. The assembler fills the space with zeros.
.`struct` *expression*	This permits you to lay out a structure using labels plus directives like `.word`, .`byte`, and so forth. It ends at the next segment directive (.`data`, .`text`, etc.). It does not emit any code or data, but defines the labels within it to have values which are the sum of *expression* plus their offsets from the .`struct` itself.
(*symbolic equate*)	Takes one of these forms: name = *expression* or *name* = *register*. You must define the name only once in the assembly, and you **cannot** redefine the name. The expression must be computable when you assemble the program, and the expression must involve operators, constants, and equated symbols. You can use the name as a constant in any later statement.
`.text`	Tells the assembler to add subsequent code to the text section. (This is the default.)
.`type` *name, value*	Specifies the `elf` type of an object denoted by name to value. Legal `elf` type values are denoted by variables prefixed by `STT_ in` <*elf.h*>.
	(64-bit and N32 only)
.`verstamp` *major minor*	Specifies the major and minor version numbers (for example, version 0.15 would be .`verstamp` 0 15).
`.weakext` *weak_name* [*strong_name*]	Defines a weak external name and optionally associates it with the *strong_name*.
`.word` *expression1* [, *expression2*]... [, *expressionN*]	Truncates the *expression*s in the comma-separated list to 32-bits and assembles the values in successive locations. The *expression*s must be absolute. The operands optionally can have the form: *expression1* `<_newline>`[: *expression2*]. The *expression2* replicates *expression1*'s value *expression2* times. This directive aligns its data

and preceding labels automatically to a word boundary. You can disable this feature by using .`align0`.

The directives listed below are only accepted in `-o32` compiles; they are only meant for compiler-generated code, and should not be used in hand-written assembly code.

```
.alias
.asm0
.bgnb
.endb
.err
.gjaldef
.gjallive
.gjrlive
.livereg
.noalias
.set bopt/nobopt
.vreg
```

PIC Assembly Code

By default, the assembler and compilers generate position-independent code (PIC). This is needed for programs that use dynamic shared libraries. For programs assembled with the `-n32` and `-64` options to `as`(1), see the *MIPSpro 64-Bit Porting and Transition Guide* for further information.

If you want to generate PIC with the assembler using the `-o32` option, assemble with the `-KPIC` and `-G0` options, which are enabled by default.

The following assembler directives support PIC generation: `.option pic2`, `.cpload`, `.cprestore`, `.gpword`, and `.cpadd`. See "Op-Codes", page 109 for information about the directives. The following example illustrates the use of these directives with the `-o32` option.

Example 8-1 `KPIC` directives example

Consider the following program:

```
     .option   pic2
     .data
     .align    2
$$5:
```

```
     .ascii    "hello world\X0A\X00"
     .text
     .align    2
main:
     .set   noreorder
     .cpload   $25
     .set   reorder
     subu $sp, 40
     sw   $31, 36($sp)
     .cprestore     32
     la   $4, $$5
     jal  printf
     move $2, $0
     lw   $31, 36($sp)
     addu $sp, 40
     j    $31
```

The actual instructions generated by the assembler are as follows:

```
     lui  gp,0     #
     addiu   gp,gp,0     #generated by .cpload
     addu gp,gp,t9 #
     lw   a0,0(g0) # gp-relative addressing used
     lw   t9,0(gp) # t9 is used for func. call
     addiu   sp,sp,-40
     sw   ra,36(sp)
     sw   gp,32(sp)# from .cprestore
     jalr ra,t9    # jal is changed to jalr
     addiu   a0,a0,0
     lw   ra,36(sp)
     lw   gp,32(sp)# activated by .cprestore
     move v0,zero
     jr   ra
     addiu   sp,sp,40
     nop
```

The ABI requires that register `t9` (`$25`) be used for indirect function calls, so `.cpload` should always use `$25`; no-reorder mode should also be used. Make sure that `t9` is dead before any function call, because the register is changed (and not restored) during the function call.

If your program uses an indirect jump (`jalr`), you must also use `t9` as the jump register.

If you have an unconditional jump to an external label (`j _cerror`), rewrite it into an indirect jump using `t9`, as in this example:

```
la t9,_cerror
j  t9
```

If you use branch-and-link (`bal`) instructions and if the target procedure begins with a `.cpload`, specify an alternate entry point, as in this example:

```
foo: .set noreorder   #callee
     .cpload   $25
     .set reorder
$$1: ...              # alternative entry point
     ...
     j   $31          # foo returns
bar: ...              # caller
     ...
     bal  $$1         # bypass the .cpload
     ...
```

The alternate entry point is important because `.cpload` assumes register `$25` contains the address of `foo`, but in this case, `$25` is not set up. Because both `foo` and `bar` reside in the same file, they must have the same value for `$gp`. The `.cpload` instructions can be and must be bypassed, but because `foo` can still be called from outside, the `.cpload` is still required.

If you don't want to have an alternate entry point, you can set up register `$25` before the `bal`, as in the following:

```
la   t9, foo
bal  foo
```

Entries of the address table created by `.gpword` are converted into displacement from the context pointer. To get the correct text address, `.cpadd` should be used to add the value of `gp` back to them. The `gp` is updated by the runtime linker, so the correct text address can be reconstructed regardless of the location of the DSO.

Index

J

K

L

M

N

U

V

W