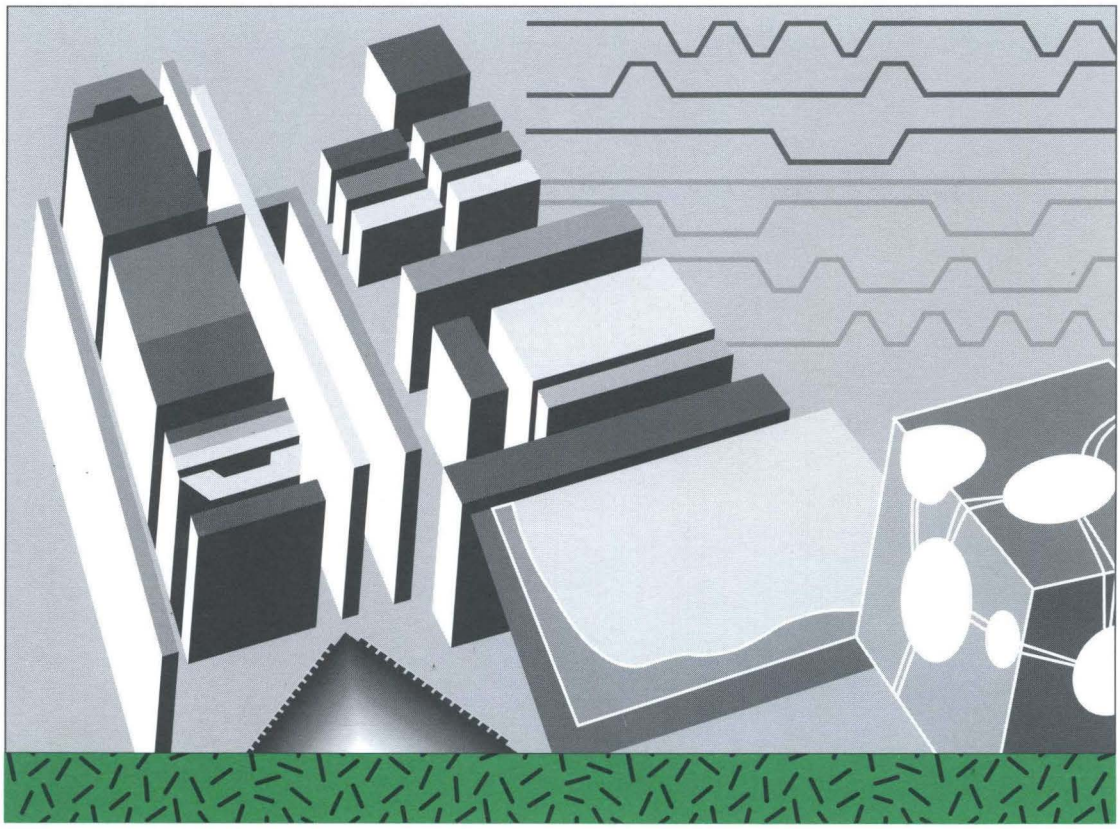
V_R Series™

Programmer's Guide



Assembly Language

November 1995

NEC

MIPS Assembly Language Programmer's Guide

November 1995
Document No. 50776

About This Book

This book describes the assembly language supported by the RISCompiler system, its syntax rules, and how to write some assembly programs. For information about assembling and linking a program written in assembly language, see the *MIPS RISCompiler and C Programmer's Guide*.

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. See **Appendix B** for more information about assembler instructions that generate multiple machine instructions.

Who Should Read This Book?

This book assumes that you are an experienced assembly language programmer.

The assembler exists primarily to produce object modules from the assembly instructions that the C, Fortran 77, and Pascal compilers generate. It therefore lacks many functions normally present in assemblers. Therefore, we recommend that you use the assembler only when you need to:

- Maximize the efficiency of a routine, which might not be possible in C, Fortran 77, Pascal, or another high-level language—for example, to write low-level I/O drivers.

- Access machine functions unavailable from high-level languages or satisfy special constraints such as restricted register usage.
- Change the operating system.
- Change the compiler system.

What Does This Book Cover?

This book has these chapters:

Chapter 1—Registers describes the format for the general registers, the special registers, and the floating point registers.

Chapter 2—Addressing describes how addressing works.

Chapter 3—Exceptions describes exceptions you might encounter with assembly programs.

Chapter 4—Lexical Conventions describes the lexical conventions that the assembler follows.

Chapter 5—Instruction Set describes the main processor's instruction set, including notation, load and store instructions, computational instructions, and jump and branch instructions.

Chapter 6—Coprocessor Instruction Set describes the coprocessor instruction sets.

Chapter 7—Linkage Conventions describes linkage conventions for all supported high-level languages. It also discusses memory allocation and register use.

Chapter 8—Pseudo-Op-Codes describes the assembler's pseudo-operations (directives).

Chapter 9—Object File Format provides an overview of the components comprising the object file and describes the headers and sections of the object file.

Chapter 10—Symbol Table describes the purpose of the Symbol Table and the format of entries in the table. This chapter also lists the symbol table routines that are supplied.

Chapter 11—Execution and Linking Format describes the Execution and Linking Format (ELF) for object files. This chapter also describes the components of an elf object file, symbol table format, global data area, register infomration, and relocation.

Chapter 12— Program Loading and Dynamic Linking describes the object file structures that relate to program execution. This chapter also describes how the process image is created from executable files and object files.

Appendix A—Instruction Summaries summarizes all assembler instructions.

Appendix B—Basic Machine Definition describes instructions that generate more than one machine instruction.

For More Information

As you use this manual, consult the following book(s):

- *RISCompiler and C Programmer's Guide* (Order number CMP-01-DOC)
- *MIPS RISC Architecture* (Order number SYS-02-DOC)

Contents

5
Instruction Set

6
Coprocessor Instruction Set

7
Linkage Conventions

8
Pseudo Op-Codes

9
MIPS Object File Format

10
Symbol Table

11
Execution and Linking Format

12
Program Loading and Dynamic Linking

A
Instruction Summary

B
Basic Machine Definition

Registers

1

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 for information regarding register use and linkage.

Register Format

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.

The byte ordering is configurable (configuration occurs during hardware reset) into either big-endian or little-endian byte ordering. 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).

Figure 1.1 and Figure 1.2 illustrate the ordering of bytes within words and the ordering of halfwords for big and little endian systems.

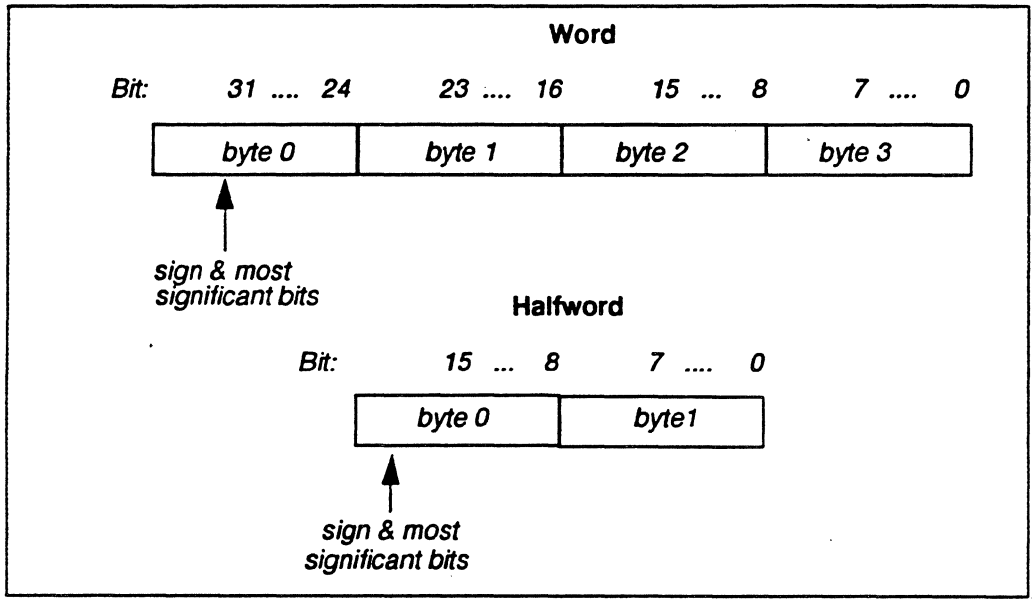


Figure 1.1 Big–endian Byte Ordering

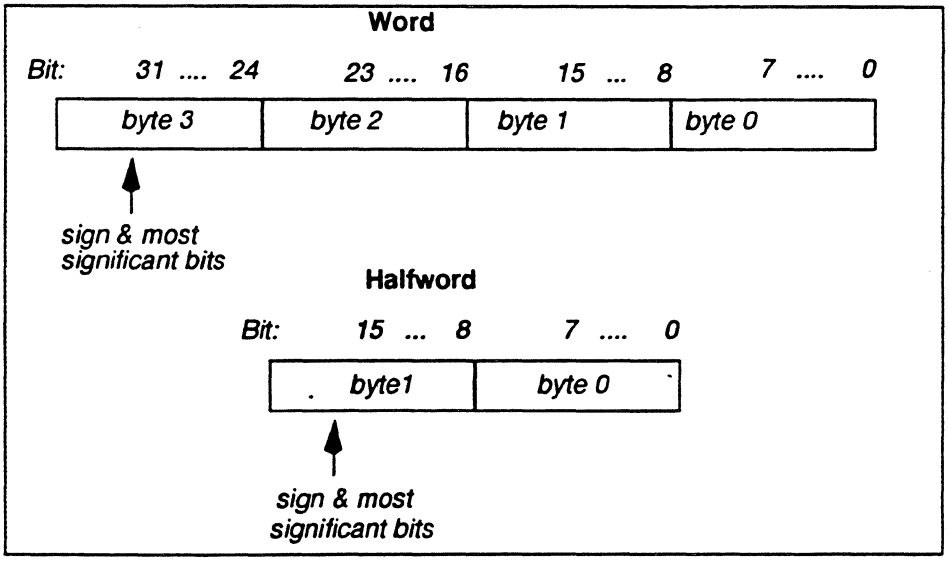


Figure 1.2 Little–endian Byte Ordering

General Registers

The CPU has thirty-two 32-bit registers. Table 1.1 summarizes the assembler's usage and 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.

Note: Attempts to use these general registers in other ways can produce unexpected results.) If a program uses the names $1, $26, $27, $28, $29 rather than the names $at, $kt0, $kt1, $gp, $sp respectively, the assembler issues warning messages.

Table 1.1 General (Integer) Registers

Register Name	Software Name (from regdef.h)	Use and Linkage
$0		Always has the value 0.
$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	Used to pass the first 4 words of integer type actual arguments, their values are not preserved across procedure calls.
$8..$15	t0-t7	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.
$30 or $fp	fp	Contains the frame pointer (if needed); otherwise a saved register (like s0-s7).
$31	ra	Contains the return address and 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 for a description of register assignments.

Special Registers

The CPU defines three 32 bit special registers: PC (program counter), HI and LO. 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 1.2 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.

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. All references to these registers use an even register number (e.g., $f4). Table 1.3 summarizes the assembler's usage conventions and restrictions for these registers.

Table 1.3 Floating Point Registers

Register Name	Use and Linkage
$f0..f2	Used to hold floating-point type function results ($f0) and complex type function results ($f0 has the real part, $f2 has the imaginary part.)
$f4..f10	Temporary registers, used for expression evaluation, whose values are not preserved across procedure calls.
$f12..$f14	Used to pass the first two single or double precision actual arguments, whose values are not preserved across procedure calls.
$f16..$f18	Temporary registers, used for expression evaluation, whose values are not preserved across procedure calls.
$f20..$f30	Saved registers, whose values must be preserved across procedure calls.

Addressing

2

This chapter describes the formats that you can use to specify addresses. The machine uses 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. Any attempt to address a data item that does not have the proper alignment causes an alignment exception.

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 word left (lwl)
- Load word right (lwr)
- Store word left (swl)
- Store word right (swr)
- Unaligned load word (ulw)
- Unaligned load halfword (ulh)
- Unaligned load halfword unsigned (ulhu)
- Unaligned store word (usw)
- Unaligned store halfword (ush)

These instructions load and store aligned data

- Load word (lw)
- Load halfword (lh)
- Load halfword unsigned (lhu)

- Load byte (lb)
- Load byte unsigned (lbu)
- Store word (sw)
- Store halfword (sh)
- Store byte (sb)

Address Formats

The assembler accepts these formats for addresses:

Table 2.1: Address Formats

Format	Address
(base register)	Base address (zero Offset assumed).
expression	Absolute address.
expression (base register)	Based address.
relocatable-symbol	Relocatable 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's 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 machine adds the value of the expression to the contents of the base–register.
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.
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 machine adds the index–register to 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 (index register)	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

3

This chapter describes the exceptions that you can encounter while running assembly programs. The machine 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 the machine references a data item 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).
  - 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—these 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

4

This chapter discusses lexical conventions for these topics:

- Tokens
- Comments
- Identifiers
- Constants
- Multiple lines per physical line
- Sections and location counters
- Statements
- Expressions.

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 #s 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)

Identifiers can be up to 31 characters long, and the first character 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). 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. Scalar constants can be any of the digits *0123456789abcdefABCDEF*.

Scalar constants can be one of these constants:

- 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, and in the floating point Load Immediate instructions, see Chapter 6. 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*.

.float and *.double* directives may optionally 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 denormalized. 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 could be confused with the octal number. For hexadecimal notation, the backslash conventions require two characters when the next character could be confused with the hexadecimal number (i.e., 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).
\t	horizontal tab (0x09).
\v	Vertical feed (0x0b).
\\	Backslash (0x5c).
\"	Quotation mark (0x22).
\'	Single quote (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 /*).

Sections and Location Counters

Assembled code and data fall in one of the sections shown in Figure 4.1.

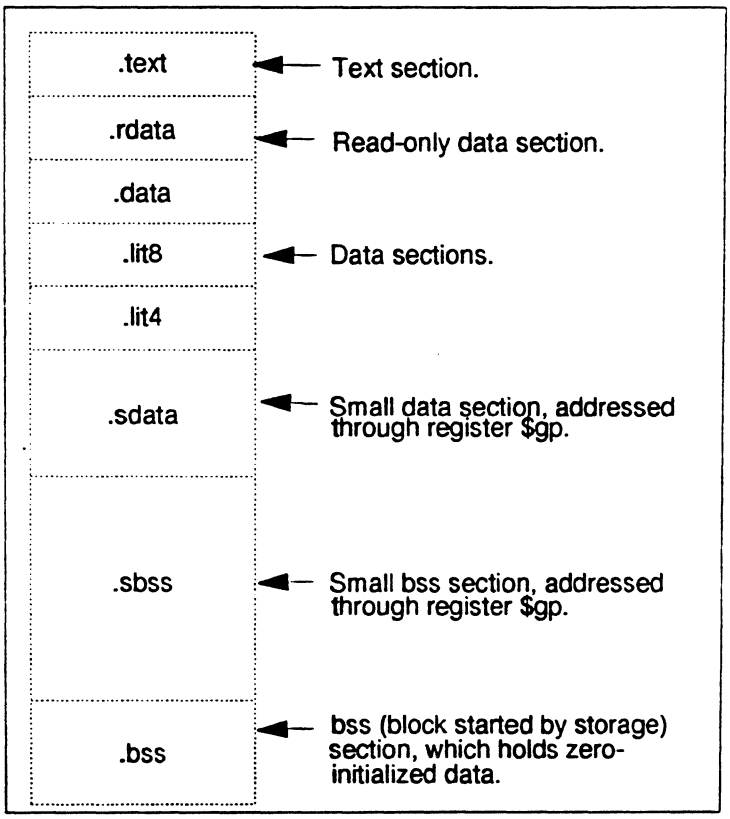


Figure 4.1: Section and location counters

(For more information on section data, see Chapter 9 of this manual.)

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), 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 64K bytes.

Statements

Each statement consists of an optional label, an operation code, and the operand(s). The machine 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 exists.

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 complement integers with 32 bits of precision. 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	*, /, %, <<, >>, ^, &, |
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 has these operators:

Figure 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 4.2:

Table 4.2: 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 64k byte 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 which 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 addressing to access these symbols.
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 64k byte region pointed to by the $gp register, so that the assembler can use economical 16–bit addressing to access it.
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 64k byte 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 addressing. 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.

Instruction Set

5

This chapter describes instruction notation and discusses assembler instructions for the main processor. Chapter 6 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.
- **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. In writing assembly language instructions, you must be aware of the rules to achieve this goal. This information is given in *MIPS RISC Architecture* (published by Prentice Hall). You should refer to the following sections in this book for more information:

Chapter	Section Title
1	Cycles/Instruction
1	Instruction Pipelines
1	Instruction Operation Time
1	Instruction Access Time
3	The Delayed Instruction Slot
3	Delayed Loads
3	Delayed Jumps and Branches
C	Filling the Branch Delay Slot

Refer also to Figure 7.4 FPA Instruction Execution Times in Chapter 7 of the same book.

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*.

Load and Store Instructions

The machine has general-purpose load and store instructions.

Load and Store Formats

The operands in Table 5.1 have the following meanings:

Operand	Description
destination	The destination register.
address	A symbolic expression (see Chapter 2).
source	The source register.
expression	An absolute value.

Table 5.1: Load and Store Formats

Description	Op-code	Operands
Load Address	la	destination, address
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 Double	ld	
Unaligned Load Halfword	ulh	
Unaligned Load Halfword Unsigned	ulhu	
Unaligned Load Word	ulw	
Load Immediate	li	destination, expression
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	

* Not valid in mips1 architectures.

Load Instruction Descriptions

For all machine 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 address of the specified data item.
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 machine 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 machine treats the loaded byte as an unsigned value, it fills the three most significant bytes of the destination register with zeros.
Load Double (ld)	ld is a machine instructon in the mips3 architecture. For the -mips3 option: Loads the destination register with the contents of the double word thta is at the memory location. The machine replaces all bytes of the register with the contents of the loaded double word. The machine signals an address error exception when the effective address is not divisible by eight. 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 machine signals an address error exception. **Note:** This is retained for use with the -mips1 and -mips2 options to provide backward compatibility only.

Table 5.2: Load Instruction Descriptions (continued)

Instruction Name	Description
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 machine treats the loaded halfword as a signed value. If the effective address is not even, the machine 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 machine 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 machine signals an address error exception.
Load Immediate (li)	Loads the destination register with the 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 Linked (ll)	Loads the destination register with the contents of the word that is at the memory location. This instruction implicitly perfomrs a SYNC operation; all loads and stores to shared memory fetched prior to the ll must access memory before the ll, and loads and stores to shared memroy fetched subsequent to the ll must access memory after the ll. Load Linked and Store Conditional can be use to automically update memory locations. This instruction is not valid in the mips1 architectures. The machine signals an address exception when the effective address is not divisible by four.
Load Upper Immediate (lui)	Loads the most significant half of a register with the expression's value, The machine fills the least significant half of the register with zeros. The expression's value must be in the range –32768...65535.

Table 5.2: Load Instruction Descriptions (continued)

Instruction Name	Description
Load Word (lw)	Loads the destination register with the contents of the word that is at the memory location. The machine replaces all bytes of the register with the contents of the loaded word. The machine 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 machine, the effective address specifies the lowest numbered byte, and in a little–endian machine 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 machine, the effective address specifies the highest numbered byte, and in a little–endian machine 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.
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.

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 5.3: Store Instruction Description

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 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.
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.
Store Double (sd)	sd is a machine instruction in the mips3 architecture. For the -mips3 option: Stores the contents of a double word 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. 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 pari, 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.

Table 5.3: Store Instruction Description (continued)

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.
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.
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 ;memrpy fetched prior to the sc must access memory before the sc, and laods and stores to shared memory fetched subsequent to teh 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 teh source register after execution of the instruction. A successful store sets it to 1; and failed store sets it to 0. This instruction is not valid in the mips1 architectures. The machine signals an address exception when the effective address is not divisible by four.

Computational Instructions

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

Computational Formats

In the Table 5.4, operands have the following meanings:

Operand	Description
destination/src1	The destination register is also source register 1.
destination	The destination register.
immediate	the immediate value.
src1,src2	The source registers.

Table 5.4: Computational Instruction Formats

Description	Op-code	Operand
Add (with overflow)	add	destination,src1,src2
Add (without overflow)	addu	destination/src1,src2
AND	and	destination,src1,
Divide (signed)	div	immediate
Divide (unsigned)	divu	destination/src1,
EXCLUSIVE OR	xor	immediate
Multiply	mul	
Multiply with Overflow	mulo	
Multiply with Overflow Unsigned	mulou	
NOT OR	nor	
OR	or	
Set Equal	seq	
Set Greater	sgt	
Set Greater/Equal	sge	
Set Greater/Equal Unsigned	sgeu	
Set Greater Unsigned	sgtu	
Set Less	slt	
Set Less/Equal	sle	
Set Less/Equal Unsigned	sleu	
Set Less Unsigned	sltu	
Set Not Equal	sne	
Subtract (with overflow)	sub	
Subtract (without overflow)	subu	

Table 5.4: Computational Instruction Formats (continued)

Description	Op-code	Operand
Remainder (signed) Remainder (unsigned) Rotate Left Rotate Right Shift Right Arithmetic Shift Left Logical Shift Right Logical	rem remu rol ror sra sll srl	destination,src1, src2 destination/src1,src2 destination,src1, immediate destination/src1, immediate
Absolute Value Negate (with overflow) Negate (without overflow) NOT Move	abs neg negu not move	destination,src1 destination/src1 destination,src1
Multiply Multiply (unsigned)	mult multu	src1,src2
Trap if Equal Trap if not Equal Trap if Less Than Trap if Less than, Unsigned Trap if Greater Than or Equal Trap if Greater than or Equal, Unsigned	teq tne tlt tltu tge tgeu	src1, src2 src1, immediate

Computational Instruction Descriptions

Table 5.5: 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.

Table 5.5 Computational Instruction Descriptions (continued)

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.
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
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.

Table 5.5 Computational Instruction Descriptions (continued)

Instruction Name	Description
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 a 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 produce; 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.
NOT OR (nor)	Computes the NOT OR of two values. This instruction combines the 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.

Table 5.5 Computational Instruction Descriptions (continued)

Instruction Name	Description
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 · 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 · 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. RotateLeft 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 the contents of src2 (or the immediate value) specify the amount to shift. RotateRight puts the result in the destination register. If src2 (or the immediate value) is greater than 32, src1 shifts by src2 MOD 32.
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.

Table 5.5 Computational Instruction Descriptions (continued)

Instruction Name	Description
Set Greater (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 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 (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 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.

Table 5.5 Computational Instruction Descriptions (continued)

Instruction Name	Description
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.
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 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.

Table 5.5: Computational Instruction Descriptions (continued)

Instruction Name	Description
Trap if Equal (teq)	Compares two 2-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 thanl the immediate value), a trap exception occurs.
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 thanl 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 thanl the contents of src2 (or src1 is greater than the immediate value), a trap exception occurs.

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 Formats

In Table 5.6 below, the operands have the following meanings:

Operand	Description
address	An expression.
src1,src2	The source registers.
target	Register containing the target.
label	A symbol label.
return	Register containing the return address.
immediate	An expression with an absolutevalue.

Table 5.6: Jump and Branch Instruction Formats

Description	Op-code	Operand
Jump	j	address
Jump and Link	jal	address target return,target
Branch on Equal Branch on Greater Branch on Greater/Equal Branch on Greater/Equal Unsigned Branch on Greater Unsigned Branch on Less Branch on Less/Equal Branch on Less/Equal Unsigned Branch on Less Unsigned Branch on Not Equal	beq bgt bge bgeu bgtu blt ble bleu bltu bne	src1,src2,label src1, immediate,label
Branch on Equal Likely* Branch on Greater Likely* Branch on Greater/Equal* Branch on Greater/Equal Unsigned Likely* Branch on Greater Unsigned Likely* Branch on Less Likely* Branch on Less/Equal Likely * Branch on Less/Equal Unsigned Likely* Branch on Less Unsigned Likely* Branch on Not Equal Likely*	beql bgtl bgel bgeul bgtul bltl blel bleul bltul bnel	src1,src2,label src1, immediate,label
Branch Branch and Link	b bal	label

* Not valid in mips1 architecture.

Table 5.6: Jump and Branch Instruction Format

Description	Op-code	Operand
Branch on Equal to Zero	beqz	src1,label
Branch on Greater/Equal Zero	bgez	
Branch on Greater Than Zero	bgtz	
Branch on Greater or Equalto 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 in mips1 architecture.

Jump and Branch Instruction Descriptions

In the following branch instructions, branch destinations must be defined in the source being assembled.

Table 5.7: 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.
Branch on Equal (beq)	Branches to the specified label when the contents of src1 equal the contents of src2, or it can branch 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 (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 RISC Architecture* book 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.

Table 5.7 Jump and Branch Instruction Descriptions (continued)

Instruction Name	Description
Branch on Greater Than Zero (bgtz)	Branches to the specified label when the contents of src1 are greater than zero.
Branch on Less (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.
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 (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 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 RISC Architecture* book 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.

Table 5.7 Jump and Branch Instruction Descriptions (continued)

Instruction Name	Description
Jump (j)	Unconditionally jumps to a specified location. A symbolic address or a general register specifies the destination. The instruction j $31 returns from the a jal call instruction.
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.
* Likely	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.

Special Formats

In *Table 5.8*, operands have the following meanings:

Operand	Description
register	Destination or source register.
breakcode	Value that determines the break type.

Table 5.8: Special Instruction Formats

Description	Op-code	Operand
Break	break	breakcode
Exception Return Restore From Exception Syscall	eret** rfe* syscall	
Move From HI Register Move To HI Register Move From LO Register Move To LO Register	mfhi mthi mflo mtlo	register

*Not available in R4000. Use the *eret* instruction.
** Not available in mips1 and mips2 architectures.

Special Instruction Descriptions

Table 5.9: Special Instruction Descriptions

Instruction Name	Description
Break (break)	Unconditionally transfers control to the exception handler. The breakcode operand is interpreted by software conventions.
Exception Return (eret)	Returns from an interrupt, exception or error trap. Similar to a branch or jump insturction, eret executes the next instruction before taking effect. Use this on R4000 processor machines in place of rfe.
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 callee 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 the machine's coprocessors.

Coprocessor Interface Formats

In Table 5.10, the operands have the following meanings:

Operand	Description
z	A coprocessor number in the range 0...3 (2 in mips3).
destination	The destination coprocessor register.
dest-gpr	The destination general register.
address	A symbolic expression.
source	A coprocessor register from which values are assigned.
src-gpr	A general register from which values are assigned.
operation	The coprocessor specific operation.
label	A symbolic label.

Table 5.10: Coprocessor Interface Instruction Formats

Description	Op-code	Operand
Load Word Coprocessor z Load Double Coprocessor z*	lwcz ldcz	destination,address
Store Word Coprocessor z Store Double Coprocessor z*	swcz sdcz	source, address
Move From Coprocessor z Move To Coprocessor z	mfcz mtcz	dest-gpr, source src-gpr, destination
Branch Coprocessor z False Branch Coprocessor z True Branch Coprocessor z FalseLikely* Branch Coprocessor z TrueLikely*	bczf bczt bczfl bcztl	label
Coprocessor z Operation	cz	expression
Control From Coprocessor z Control To Coprocessor z	cfcz ctcz	dest-gpr, source src-gpr, destination

* Not valid in mips1 architectures.

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

Coprocessor Interface Instruction Descriptions

Table 5.11: 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 inthe branch delay slot is nullified.
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 inthe branch delay slot is nullified.
Control From Coprocessor z (cfcz)	Stores the contents of the coprocessor control register specified by the source in the general register specified by dest–gpr.
Control To Coprocessor z (ctcz)	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.

Note: The branch likely insturcions should be used only within a .set noreorder block. The assembler does not attempt to schedule the delay slot of a branch likely instruction.

Table 5.11: Coprocessor Interface Instruction Descriptions (continued)

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 insturction 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.
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 insturction 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

6

This chapter describes the coprocessor instructions for these coprocessors:

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

See Chapter 5 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 below.

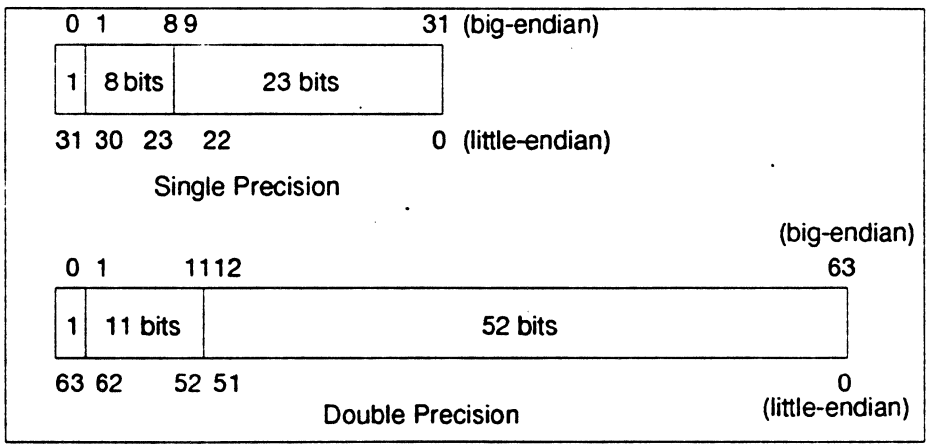


Figure 6-1: Floating Point Formats

Floating Point Load and Store Formats

Floating point load and store instructions must use even registers. The operands in Table 6.1 have the following meanings:

Operand	Meaning
destination	The destination register.
address	Offset (base).
source	The 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 Immediate Fp		
Double	li.d	destination, floating point constant
Single	li.s	
Store Fp		
Double	s.d	source, address
Single	s.s	

Floating Point Load and Store Descriptions

This part of Chapter 6 groups the instructions by function. Please consult Table 6.1 for the op–codes.

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. The bytes must be word aligned. **Note:** We recommend that you use double word alignment for double precision operands. It is required in the mips2 architecture (R4000 & R6000).
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. **Note:** We recommend that you use double word alignment for double precision operands. It is required in the mips2 architecture (R4000 & R6000).

Floating Point Computational Formats

This part of Chapter 6 describes floating point computational instructions. The operands in Table 6.3 below have the following meaning:

Operand	Meaning
destination	The destination register.
source	The source register.
gpr	General purpose register.

Table 6.3: Floating Point Computational Instruction Formats

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	
Divide Fp		
Double	div.d	
Single	div.s	
Multiply Fp		
Double	mul.d	
Single	mul.s	
Subtract Fp		
Double	sub.d	
Single	sub.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	
Double to Fixed Point Fp	cvt.w.d	

Table 6.3 Floating Point Computational Instruction Formats (continued)

Description	Op-code	Operand
Truncate and Round Operations		
Truncate to Single Fp	trunc.w.s	
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	destination, src, gpr
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 Fp, 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	

Floating Point Computational Instruction Descriptions

This part of Chapter 6 groups the instructions by function. Refer to Table 6.3 for the op–code names.

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 Single 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.
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.

Table 6.4 Floating Point Computational Instruction Descriptions (continued)

Instruction	Description
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.
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.

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 to logically 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

Condition		Relations				Invalid Operation Exception if Unordered
Mnemonic		Greater Than	Less Than	Equal	Unordered	
True	False					
F	T	F	F	F	F	no
UN	OR	F	F	F	T	no
EQ	NEQ	F	F	T	F	no
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 in have the following meanings:

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
OLE	Ordered Less Than or Equal	UGT	Unordered or Greater Than
ULE	Unorderd or Less Than or Equal	OGT	Ordered Greater Than
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 Than
LT	Less Than	NLT	Not Less Than
NGE	Not Greater Than or Equal	GE	Greater Than or Equal
LE	Less Than or Equal	NLE	Not Less Than or Equal
NG	Not Greater Than	G	Greater Than

To branch on the result of a relational:

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

c.eq.s $f1,$f2 /* compare the single precision values */
bclt true /* if $f1 equals $f2, branch to true */
bclf false /* if $f1 does not equal $f2, branch to */
/* false */
```

Floating Point Relational Instruction Formats

In the table below, *src1* and *src2* refer to the source registers.

Note: These are the most common *Compare* instructions. The machine provides other Compare instructions for IEEE compatibility.

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	
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	

Table 6.6 Floating Point Relational Instruction Formats (continued)

Description	Op-code	Operand
Compare NGLE		
Double	c.ngle.d	src1, src2
Single	c.ngle.s	
Compare SEQ		
Double	c.seq.d	
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	

Floating Point Relational Instruction Descriptions

This part of Chapter 6 describes the relational instruction descriptions by function. Refer to Chapter 1 for information regarding registers. Please consult Table 6.6 for the op–code names.

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 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 signals an exception for unordered values.
Compare LT 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 signals an exception for unordered values.
Compare NGE 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 signals an exception for unordered values.
Compare NGL Instructions	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 Instructions	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 Instructions	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.

Table 6.7: Floating Point Relational Instruction Descriptions (continued)

Instruction	Description
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.
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 coprocessor's *move* instructions move data from source to destination registers (only floating point registers are allowed).

Table 6.8: Floating Point Move Instruction Formats

Description	Op-code	Operand
Move Fp Double Single	 mov.d mov.s	destination, src1

Floating Point Move Instruction Descriptions

This part of Chapter 6 describes the floating point *move* instructions. Please consult Table 6.8 for the op-code names.

Table 6.9: 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.

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:

Table 6.10: System Control Instruction Formats.

Description	Op-code
Cache**	cache
Translation Lookaside Buffer Probe	tlbp
Translation Lookaside Buffer Read	tlbr
Translation Lookaside Buffer Write Random	tlbwr
Translation Lookaside Write Index	tlbwi
Synchronize*	sync

* Not valid in mips1 architectures.
* Not valid in mips1 and mips2 architectures..

System Control Coprocessor Instruction Descriptions

This part of Chapter 6 describes the system control coprocessor instructions.

Table 6.11: 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.
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. This instruction is not valid in the mips1 architecture.

Control and Status Register

Floating-point coprocessor control register 31 contains status and control information. 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.

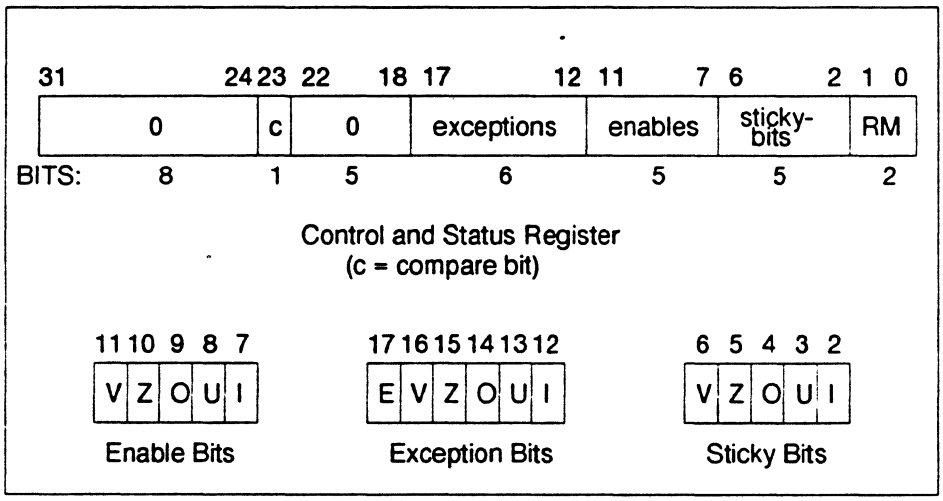


Figure 6-2: Floating Control and Status Register 31

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.
V	Invalid Operation.
Z	Division by Zero.
I	Inexact Exception.
O	Overflow Exception.
U	Underflow Exception.

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
V	Invalid Operation.
Z	Division by Zero.
I	Inexact Exception.
O	Overflow Exception.
U	Underflow Exception.

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: $\sqrt{x}$, 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 trancendental 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 expressable 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 properly handle the condition. 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 = 0x0
RoundZero = 0x1
RoundPosInf = 0x2
RoundNegInf = 0x3
      cfc1 rt2, $31       # move from coprocessor 1
      and rt, 0xfffffffc  # zero the round mode bits
      or rt, RoundZero    # set mask as round to zero
      ctc1 rt, $f31       # move to coprocessor 1
```

Linkage Conventions

7

This chapter gives rules and examples to follow when designing an assembly language program. The chapter concludes with a "learn by doing" technique that you can use if you still have any doubts about how a particular calling sequence should 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.

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 three general areas of concern to the assembly language programmer:

- Usable and restricted registers.
- 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.

Register Use and Linkage

The main processor has 32 32-bit integer registers. The uses and restrictions of these registers are described in Table 1.1 in Chapter 1.

The floating point coprocessor has 16 floating point registers. Each register can hold either a single precision (32 bit) or a double precision (64 bit) value. All references to these registers uses an even register number (e.g., $f4). Refer to Table 7.1 for details.

Table 7.1: Floating Point Registers

Floating Point Registers	
register name	**use and linkage**
$f0..f3	Used to hold floating point type function results ($f0) and complex type function results ($f0 has the real part, $f2 has the imaginary part).
$f4..f10	Temporary registers, used for expression evaluation, whose values are not preserved across procedure calls.
$f12..$f14	Used to pass the first 2 single or double precision actual arguments, whose values are not preserved across procedure calls.
$f16..$f18	Temporary registers, used for expression evaluations, whose values are not preserved across procedure calls.
$f20..$f30	Saved registers, whose values mustbe preserved across procedure calls.

The Stack Frame

The compilers classify each routine into one of 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 8. 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 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). Figure 7.1 illustrates the stack components.

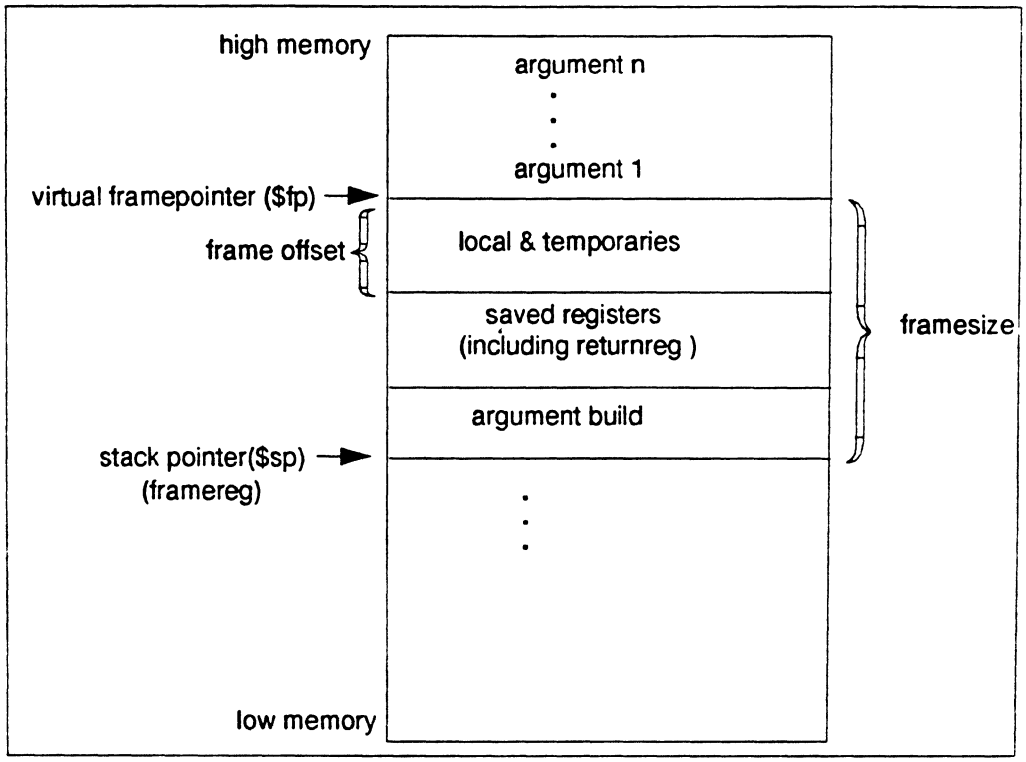


Figure 7.1: Stack Organization

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.

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          $16,framesize+frameoffset-8($sp)
```

Figure 7.2 illustrates this example.

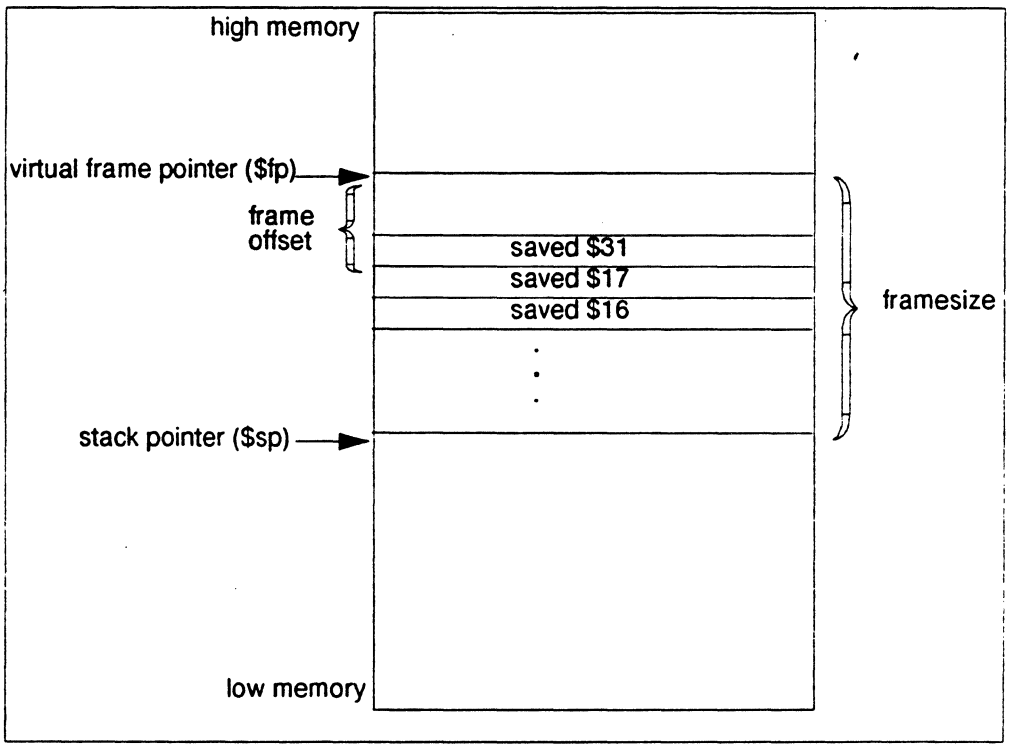


Figure 7.2: Stack Example

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 8 for doubles. The stack framesize *must* be a multiple of 8.

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 Chapter 4 of the *RISCompiler and C Programmer's 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 $4–$7 and float registers $f12, $f14 must be 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.

   In the table below, the' 'fN' arguments are considered single and double precision floating point arguments, and'nN' 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.

Arguments	Register Assignments
(f1, f2, ...)	f1 -> $f12, f2 -> $f14
(f1, n1, f2, ...)	f1 -> $f12, n1 -> $6, f2 ->stack
(f1, n1, n2, ...)	f1 -> $f12, n1 -> $6, n2 -> $7
(n1, n2, n3, n4, ...)	n1 -> $4, n2 -> $5, n3 -> $6, n4 -> $7
(n1, n2, n3, f1, ...)	n1 -> $4, n2 -> $5, n3 -> $6, f1 -> stack
(n1, n2, f1, ...)	n1 -> $4, n2 -> $5, f1 -> ($6, $6)
(n1, f1, ...)	n1 -> $4, f1 -> ($6, $7)

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 $sp, framesize
```

9. Return:

```
j $31
```

10. To end the procedure:

```
.end procedurename
```


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 our scope here. If you need to access such a structure as an argument or as a shared global variable, refer to Chapter 4 of the *RISCompiler and C Programmer's Guide*, and the "Learn by Doing" technique described at the end of this section.

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.

Figure 7.3 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(i, j)
  int i, *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:
             subu            $sp, 24          ## Create stackframe
             sw              $31, 20($sp)     ## Save the return address
             .mask           0x80000000, -4
             .frame          $sp, 24, $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 registe
             negu            $3, $3
$32:
  #   10       return atof(temp);
             move            $4, $3
             jal             atof
             cvt.s.d         $f0, $f0         ## Returnvalue goes in $f0
             lw              $31, 20($sp)     ## Restore return address
             addu            $sp, 24          ## Delete stackframe
             j               $31              ## Return to caller
             .end            nonleaf
```

Figure 7.3: Non-Leaf Procedure

Figure 7.4 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            leaf 2
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
```

Figure 7.4: Leaf Procedure Without Stack Space for Local Variables

Figure 7.5 shows a leaf procedure that requires stack space for local variables. Notice that it creates a stack frame, but does not save a return address.

```
char
leaf_storage(i)
  int i;
  {
  char a[16];
  int j;

  for (j = 0; j < 10; j++)
    a[j] = '0' + j;
  for (j = 10; j < 16; j++)
    a[j] = 'a' + j;
  return a[i];
  }

                .globl          leaf_storage
 #   1          char
 #   2          leaf_storage(i)
 #   3            int i;
 #   4            {
                .ent            leaf_storage 2 ## "2" is the lexical level of the
                                               ## procedure.  You may omit it.
leaf_storage:
                subu            $sp, 24          ## Create stackframe.
                .frame          $sp, 24, $31
 #   5            char a[16];
 #   6            int j;
 #   7
 #   8            for (j = 0; j < 10; j++)
                sw              $0, 4($sp)
                addu            $3, $sp, 24
$32:
 #   9              a[j] = '0' + j;
                lw              $14, 4($sp)
                addu            $15, $14, 48
                addu            $24, $3, $14
                sb              $15, -16($24)
                lw              $25, 4($sp)
                addu            $8, $25, 1
                sw              $8, 4($sp)
                blt             $8, 10, $32
 #  10            for (j = 10; j < 16; j++)
                li              $9, 10
                sw              $9, 4($sp)
$33:
 #  11             a[j] = 'a' + j;
                lw              $10, 4($sp)
                addu            $11, $10, 97
                addu            $12, $3, $10
                sb              $11, -16($12)
                lw              $13, 4($sp)
                addu            $14, $13, 1
                sw              $14, 4($sp)
                blt             $14, 16, $33
 #  12            return a[i];
                addu            $15, $3, $4          ## Argument is in $4.
                lbu             $2, -16($15)         ## Return value goes in $2.
                addu            $sp, 24              ## Delete stackframe.
                j               $31                  ## Return to caller.
                .end            leaf_storage
```

Figure 7.5: Leaf Procedure With Stack Space for Local Variables

Learning by Doing

The rules and parameter requirements required 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.
- Study the assembly–language listing and then, imitating the rules and conventions used by the compiler, write your assembly language code.

The next two sections illustrate techniques to use in creating an interface between assembly language and high–level language routines. The examples shown are merely to illustrate what to look for in creating your interface. Details such as register numbers will vary according to the number, order, and data types of the arguments. You should write and compile realistic examples of your own code in writing your particular interface.

Calling a High–Level Language Routine

The following steps show a technique to follow in writing an assembly language routine that calls *atof*, a routine written in C that converts ASCII characters to numbers; for more information, see the *atof*(3) in the *RISC/os Programmer's Reference Manual.*

1. Write a C program that calls *atof*. Pass global rather than local variables; this makes them to recognize in the assembly language version of the C program (and ensures that optimization doesn't remove any of the code on the grounds that it has no effect).

   Below is an example of a C program that calls *atof*

```
char c[] = "3.1415";
double d, atof();
float f;
caller()
  {
  d = atof(c);
  f = (float) atof(c);
  }
```

c is declared as a global variable.

2. Compile the program using the compiler options shown below:

   ```
   cc -S -O caller.c
   ```

   The —S option causes the compiler to produce the assembly-language listing; the —O option, though not required, reduces the amount of code generated, making the listing easier to read.

   After compilation, look at the file caller.s (shown below). The highlighted section of the listing shows how the parameters are passed, the execution of the call, and how the returned values are retrieved.

```
        .globl      c
        .align 2
c:
        .word       875638323 : 1
        .word       13617 : 1
        .comm       d 8
        .comm       f 4
        .globl      caller
        .text
        .ent        caller 2
caller:
        subu        $sp, 24
        sw          $31, 20($sp)
        .mask       0x80000000, -4
        .frame      $sp, 24, $31
 #   1          char c[] = "3.1415";
 #   2          double d, atof();
 #   3          float f;
 #   4          caller()
 #   5            {
 #   6            d = atof(c);
        la          $4, c           ## load address of c
        jal         atof            ## call atof
        s.d         $f0, d          ## store result in d
 #   7
        la          $4, c           ## load address of c
        jal         atof            ## call atof
        cvt.s.d     $f4, $f0        ## convert double result to float
        s.s         $f4, f          ## store float result in f
        lw          $31, 20($sp)
        addu        $sp, 24
        j           $31
        .end        caller
```

Calling an Assembly Language Routine

This section shows a technique to follow in writing an assembly language routine that calls a routine written in a high–level language (Pascal is used in this example).

1. Write a facsimile of the assembly language routine you wish to call. In the body of the routine, write statements that use the same arguments you intend to use in the final assembly language routine. Copy the arguments to global variables rather than local variables to make it easy for you to read the resulting assembly language listing.

   Below is the Pascal facsimile of the assembly language program.

```
type
  str = packed array [1 .. 10] of char;
  subr = 2 .. 5;
var
  global_r: real;
  global_c: subr;
  global_s: str;
  global_b: boolean;

function callee(var r: real; c: subr; s: str): boolean;
  begin
  global_r := r;
  global_c := c;
  global_s := s;
  callee := c = 3;
  end;
```

2. Compile the program using the compiler options shown below:

```
cc -S -O caller.c
```

   The —S option causes the compiler to produce the assembly–language listing; the —O option, though not required, reduces the amount of code generated, making the listing easier to read.

3. After compilation, look at the file caller.s (shown below). The highlighted section of the listing shows how the parameters are passed, the execution of the call, and how the returned values are retrieved.

```
        .lcomm          $dat 0
        .comm           global_r 4
        .comm           global_c 1
        .comm           global_s 10
        .comm           global_b 1
        .text
        .globl          callee
#  10     function callee(var r: real; c: subr; s: str): boolean;
        .ent            callee 2
callee:
        .frame          $sp, 0, $31
        sw              $5, 4($sp)
        sw              $6, 8($sp)
        lbu             $3, 4($sp)      ## Get subrange c, masking it to 8 bits
        and             $3, $3, 255
#  11       begin
#  12       global_r := r;
        l.s             $f4, 0($4)      ## The pointer to "r" is in 0($4)
        s.s             $f4, global_r
#  13       global_c := c;
        sb              $3, global_c
#  14       global_s := s;
        la              $14, global_s   ## For array "s", the caller gives you a
        addu            $15, $sp, 8     ## pointer at 8(Ssp). If youwant to use
        .set             noat           ## it as a call-by-value argument just as
        addu            $24, $15, 10    ## Pascal does (that is, if you want to
$32:                                    ## be able to modify a local copy without
        lbu             $1, 0($15)      ## affecting the global copy) then you
        addu            $15, $15, 2     ## must copy it into your stack frame as
        sb              $1, 0($14)      ## shown here (the code enclosed in ".set
        lbu         31  $1, -1($15)     ## noat" is a tight byte-copying loop).
        addu            $14, $14, 2     ## Otherwise, you may simply use the
        sb              $1, -1($14)     ## pointer provided to you.
        bne             $15, $24, $32
        .set             at
#  15       callee := c = 3;
        seq             $5, $3, 3
        and             $5, $5, 255
#  16       end;
        and             $2, $5, 255     ## Return the boolean by leaving it in $2
        j               $31
        .end
```

Memory Allocation

The machine's default memory allocation scheme gives every process two storage areas, that can grow without bound. A process exceeds virtual storage only when the sum of the two areas exceeds virtual storage space. The link editor and assembler use the scheme shown in Figure 7.6. An explanation of each area in the allocation scheme follows the figure.

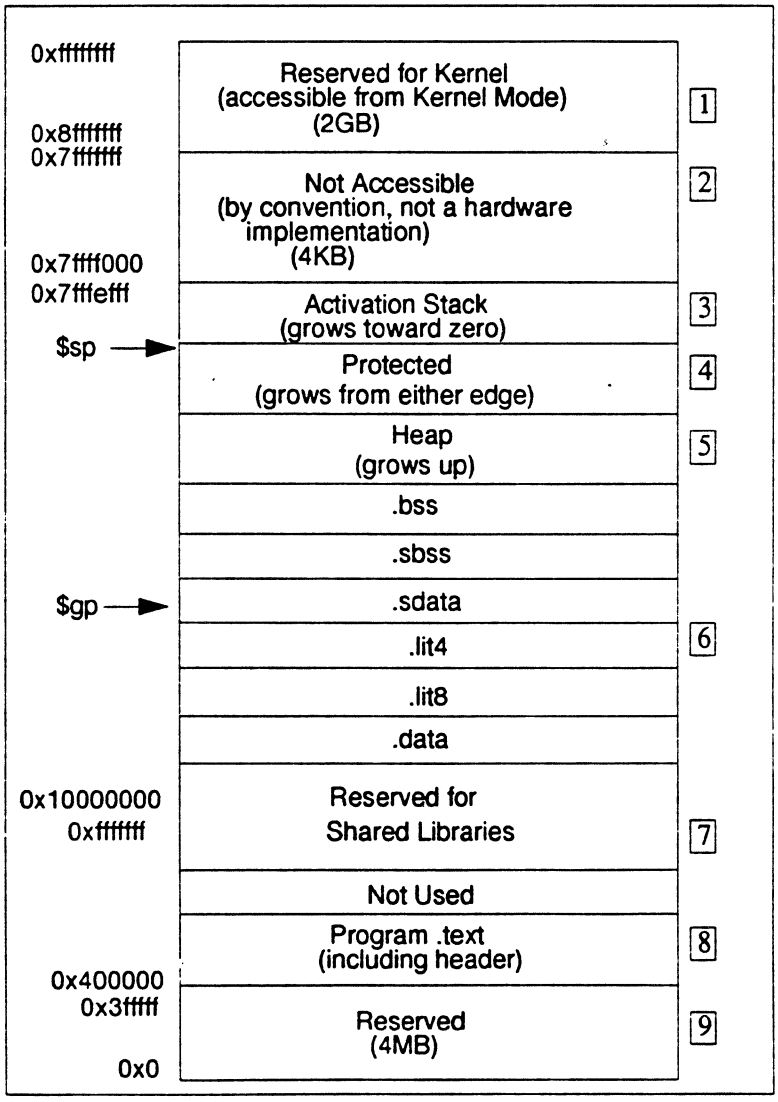


Figure 7.6: Layout of memory (User Program View)

1. Reserved for kernel operations.
2. Reserved for operating system use.
3. Used for local data in C programs.
4. Not allocated until a user requests it, as in System V shared memory regions.
5. The heap is reserved for sbrk and break system calls, and it not always present.
6. The machine divides all data into one of five sections:
   - bss - Uninitialized data with a size greater than the value specified by the –G command line option.
   - sbss - Data less than or equal to the –G command line option. (512 is the default value for the –G option.)
   - sdata (small data) - Data initialized and specified for the sdata section.
   - data (data) - Data initialized and specified for the data section.
7. Reserved for any shared libraries.
8. Contains the .text section, .rdata section and all dynamic tables.
9. Reserved.

Pseudo Op–Codes

8

This chapter describes pseudo op–codes (directives). These pseudo op–codes influence the assembler's later behavior. In the text, boldface type specifies a keyword and italics represents an operand that you define.

The assembler has these pseudo op–codes:

Pseudo-Op	Description
.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.
.alias *reg1, reg2*	Indicates that memory reference through the two registers (*reg1, reg2*) will overlap. The compiler uses this form to improve instruction scheduling.
.align *expression*	Advance the location counter to make the *expression* low order bits of the counter zero.

Pseudo-Op	Description
	Normally, the .half, .word, .float, and .double directives automatically align their data appropriately. For example, *.word* does an implicit .align 2 (.double does a .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: .word0 .
.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.
.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.
.asm0	Tells the assembler's second pass that this assembly came from the first pass. (For use by compilers.)

Pseudo-Op	Description
.bgnb *symno*	(For use by compilers.) Sets the beginning of a language block. The *.bgnb* and *.endb* directives delimit the scope of a variable set. The scope can be an entire procedure, or it can be a nested scope (for example a "{}" block in the C language). The symbol number *symno* refers to a dense number in a **.T** file. For an explanation of **.T** files, see the *RISCompiler and C Programmer's Guide*. To set the end of a language block, see *.endb*.
.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*	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*.
.data	Tells the assembler to add all subsequent data to the data section.

Pseudo-Op	Description
.double *expression [, expression2] ...[, expressionN]*	Initializes memory to 64–bit floating point numbers. The operands can optionally 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 automatically aligns its data and any preceding labels to a double–word boundary. You can disable this feature by using *.align* 0.
.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*.
.endb *symno*	Sets the end of a language block. To set the beginning of a language block, see *.bgnb*.
.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*.

Pseudo-Op	Description
.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.
.err	Signals an error. Any compiler front–end that detects an error condition puts this directive in the input stream. When the assembler encounters a *.err*, it quietly ceases to assemble the source file. This prevents the assembler from continuing to process a program that is incorrect. (For use by compilers.)
.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.

Pseudo-Op	Description
.float *expression1 [, expression2]... [, expressionN]*	Initializes memory to single precision 32–bit floating point numbers. The operands can optionally have the form: *expression1* [: *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 automatically aligns its data and preceding labels to a word boundary. You can disable this feature by using *.align* 0.
.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*.

Pseudo-Op	Description
.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 otherwise defined (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).
.gjaldef int_bitmask fp_bitmask	For use by compilers. Sets the masks defining the registers whose value is preserved during a procedure call. See Table 1.1 and Table 7.1 for the default for integer saved registers.
.gjallive int_bitmask fp_bitmask	For use by compilers. Sets the default masks for live registers before a procedure call (A JAL instruction).
.gjrlive int_bitmask fp_bitmask	For use by compilers. Sets the default masks for live registers before a procedure's return (A JR instruction).

Pseudo-Op	Description
.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 can optionally 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 *.align* 0.
.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.

Pseudo-Op	Description
.livereg *int_bitmask fp_bitmask*	For use by compilers. Affects the next jump instruction even if it is not the successive instruction. The *.livereg* directive may come before any of the following instructions: JAL, JR, and SYSCALL. By default, external J instructions and JR instructions through a register other than $ra, are treated as external calls; that is; all registers are assumed live. The directive *.livereg* cannot appear before an external J (it will affect the next JR, JAL, or SYSCALL instead of the J instruction). *.livereg* may appear before a JR instruction through a register other than $ra. The directive can't be used before a BREAK instruction. For BREAK instructions, the assembler also assumes all registers are live. *.livereg* notes to the assembler which registers are live before a jump, in order to avoid unsafe optimizations by the reorganizer. The directive *.livereg* takes two arguments, *int_bitmask*, and *fp_bitmask*, which are 32 bit bitmasks with a bit turned on for each register that is live before a jump. The most significant bit corresponds to register $0 (which is opposite to that used in other assembly directives, *.mask*, *.fmask*). The first bitmap indicates live integer registers and the second indicates live FPs.

Pseudo-Op	Description
.loc *file_number line_number*	Specifies the source file and the line within that file that corresponds to the assembly instructions that follow. 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. When a *.loc* directive appears in the binary assembly language .G file, the file number is a dense number pointing at a file symbol in the symbol table .T file. For more information about .G and .T files, see the *RISCompilers and C Programmer's Guide*. (For use by compilers).
.mask *mask, offset*	Sets a mask with a bit turned on for each general purpose register that the current routine saved. 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 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 use by compilers).

Pseudo-Op	Description
.noalias *reg1, reg2*	Register1 and register2, when used as indexed registers to memory will never point to the same memory. The assembler will use this as a hint to make more liberal assumptions about resource dependency in the program. To disable this assumption, see *.alias.*
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.

Pseudo-Op	Description
.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.
.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*.
.rdata	Tells the assembler to add subsequent data into the rdata section.
.sdata	Tells the assembler to add subsequent data to the sdata section.
.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.

Pseudo-Op	Description
	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 *bopt/nobopt* option lets the assembler perform branch optimization. This involves moving an instruction that is the target of a branch or jump instruction into the delay slot; this is perofrmed only if no unpredictable side effects can occur.
	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.

Pseudo-Op	Description
	The *nomove* options 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*.
.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.

Pseudo-Op	Description
(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.)
.verstamp *major minor*	Specifies the major and minor version numbers (for example, version 0.15 would be *.verstamp* 0 15).
.vreg *register offset symno*	(For use by compilers). Describes a register variable by giving the offset from the virtual frame pointer and the symbol number *symno* (the dense number) of the surrounding procedure.
.word *expression1 [, expression2]... [, expressionN]*	Truncates the *expressions* in the comma–separated list to 32–bits 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. This directive automatically aligns its data and preceding labels to a word boundary. You can disable this feature by using *.align* 0.

MIPS Object File Format

9

This chapter provides information on the object file format and has the following major topics:

- An overview of the components that make up the object file, and the differences between the MIPS object-file format and the UNIX System V common object file format (COFF).
- A description of the headers and sections of the object file. Detailed information is given on the logic followed by the assembler and link editor in handling relocation entries.
- The format of object files (OMAGIC, NMAGIC, ZMAGIC, and LIBMAGIC), and information used by the system loader in loading object files at run-time.
- Archive files and link editor defined symbols.

Overview

The assembler and the link editor generate object files that have sections ordered as shown in Figure 9.1. Any areas empty of data are omitted, except that the File Header, Optional Header, and Section Header are always present.

The sections of the Symbol table portion (indicated in Figure 9.1) that appear in the final object file format vary, as follows:

- The Line Numbers, Optimization Symbols, and Auxiliary Symbols tables appear only when debugging is on (when the user specifies one of the compiler -g1, -g2 or -g3 options).

- When the user specifies the -x option (strip non-globals) for the link edit phase, the link editor strips the Line Number, Local Symbols, Optimization Symbols, Auxiliary Symbols, Local Strings, and Relative File Descriptor tables from the object file, and updates the Procedure Descriptor table.
- The link editor strips the entire Symbol table from the object file when the user specifies the -s option (strip) for the link edit phase.

Any new assembler or link editor designed to work with the compiler system should lay out the object file sections in the order shown in Figure 9.1. The link editor can process object files that are ordered differently, but performance may be degraded.

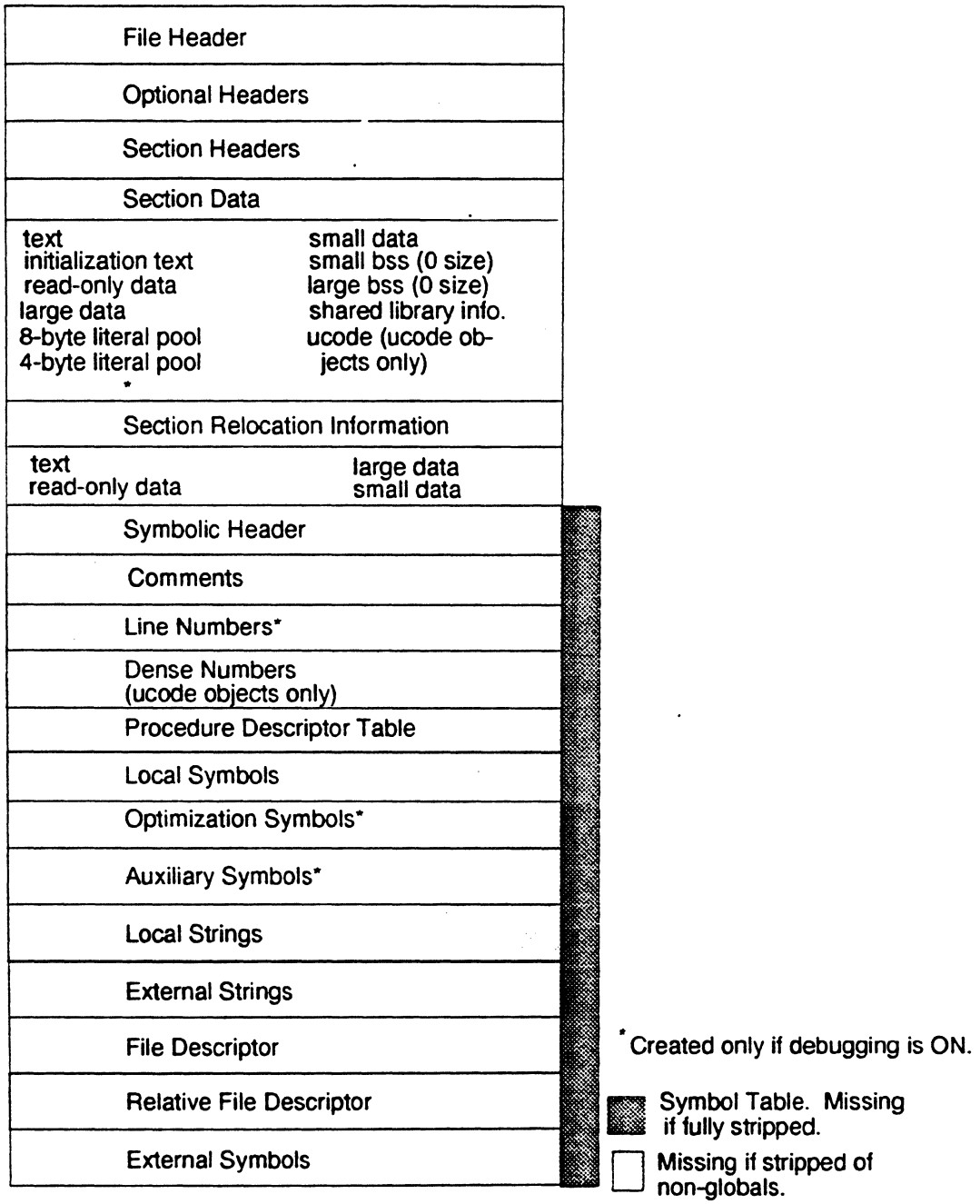


Figure 9.1: Object File Format

Readers already familiar with standard UNIX System V COFF (common object file format) may be interested in the differences between it and the MIPS compiler system format, as described next.

The compiler system File Header definition is based on UNIX System V header file *filehdr.h* with the following modifications.

- The symbol table file pointer and the number of symbol table entries now specify the file pointer and the size of the Symbolic Header respectively (described in Chapter 10).
- All tables that specify symbolic information have their file pointers and number of entries in this Symbolic Header.

The Optional Header definition has the same format as specified in the UNIX System V header file *aouthdr.h*, except the following fields have been added: *bss_start*, *gprmask*, *cprmask*, and *gp_value*. See Table 9.4.

The Section Header definition has the same format as the UNIX System V's header file *scnhdr.h*. except the line number fields are used for global pointers. See Table 9.6.

The relocation information definition is similar to UNIX 4.3 BSD, which has local relocation types; however, you should read the Section Relocation Information section in this chapter for information on differences.

The File Header

The format of the File Header, defined in *filehdr.h*, is shown in Table 9.1.

Table 9.1: File Header Format

Declaration	Field	Description
unsigned short	f_magic;	Magic number.
unsigned short	f_nscns;	Number of sections.
long	f_timdat;	Time and date stamp.
long	f_symptr;	File pointer to symbolic header.
long	f_nsyms;	Size of symbolic header.
unsigned short	f_opthdr;	Size of optional header.
unsigned short	f_flags;	Flags.

f_symptr points to the Symbolic Header of the Symbol table, and *f_nsyms* gives the size of the header. For a description of the Symbolic Header, see Chapter 10.

File Header Magic Field (f_magic)

The magic number in the *f_magic* entry in the File Header specifies the target machine on which an object file can execute. Table 9.2 shows the values and mnemonics for the magic numbers; the header file *filehdr.h* contains the macro definitions.

Table 9.2: File Header Magic Numbers

Symbol	Value	Description
MIPSEBMAGIC MIPSEBMAGIC_2	0x0160 0x0163	Big–endian target (headers and tables have same byte order as host machine).
MIPSELMAGIC MIPSELMAGIC_2	0x0162 0x0166	Little–endian target (headers and tables have same byte order as host machine).
SMIPSEBMAGIC SMIPSEBMAGIC_2	0x6001 0x6301	Big–endian target (headers and tables have opposite byte order as host machine).
SMIPSELMAGIC SMIPSELMAGIC_2	0x6201 0x6601	Little–endian target (headers and tables have opposite byte order as host machine).
MIPSEBUMAGIC MIPSELUMAGIC	0x0180 0x0182	MIPS big–endian ucode object file. MIPS little–endian ucode object file.

Note: The "_2" magic numbers are defined for mips2 object files. They cannot be used on a MIPS I implementation.

Flags (f_flags)

The *f_flags* field describes the object file characteristics. Table 9.3 describes the flags and gives their hexadecimal values. The table notes those flags that do not apply to compiler system object files.

Table 9.3: File Header Flags

Symbol	Value	Description
F_RELF bLG	0x0001	Relocation information stripped from file
F_EXEC	0x0002	File is executable (i.e. no unresolved external references).
F_LNNO	0x0004	Line numbers stripped from file.
F_LSYMS	0x0008	Local symbols stripped from file.
F_MINMAL	0x0010	!Minimal object file (".m") output of *fextract*.
F_UPDATE	0x0020	!Fully bound update file, output of *ogen*.
F_SWABD	0x0040	!File whose bytes were *swabbed* (in names).
F_AR16WR	0x0080	!File has the byte ordering of an AR16WR. (e.g.11/70) machine (it was created there, or was produced by *conv*).
F_AR32WR	0x0100	!File has the byte ordering of an AR32WR machine (e.g. vax).
F_AR32W	0x0200	!File has the byte ordering of an AR32W machine (e.g. 3b,maxi,MC68000).
F_PATCH	0x0400	!File contains "patch" list in Optional Header.
F_NODF	0x0400	!(Minimal file only) no decision functions for replaced functions.
F_MIPS_NO_SHARED	010000	Cannot be dynamically shared.
F_MIPS_SHARABLE	020000	A dynamically shared object.
F_MIPS_CALL_SHARED	030000	Dynamic executable.
F_MIPS_NO_REORG	040000	Do not reorder sections.

!Not used by compiler system object modules.

Optional Header

The link editor and the assembler fill in the Optional Header, and the system (kernel) loader (or other program that loads the object module at run–time) uses the information it contains, as described in the section Loading Object Files in this chapter.

Table 9.4 shows the format of the Optional Header (defined in the header file *aouthdr.h*).

Table 9.4: Optional Header Definition

Declaration	Field	Description
short	magic;	See Table 9.5.
short	vstamp;	Version stamp.
long	tsize;	Text size in bytes, padded to 16–byte boundary.
long	dsize;	Initialized data in bytes, padded to 16–byte boundary.
long	bsize;	Uninitialized data in bytes, padded to 16–byte boundary.
long	entry;	Entry point.
long	text_start;	Base of *text* used for this file.
long	data_start;	Base of *data* used for this file.
long	bss_start;	Base of *bss* used for this file.
long	gprmask;	General purpose register mask.
long	cprmask[4];	Co–processor register masks.
long	gp_value;	The gp value used for this object.

Optional Header Magic Field (magic)

Table 9.5 shows the values of the *magic* field for the Optional Header; the header file *aouthdr.h* contains the macro definitions.

Table 9.5: RISC/os Magic Numbers

Symbol	Value	Description
OMAGIC	0407	Impure Format. The text is not write-protected or sharable; the data segment is contiguous with the text segment.
NMAGIC	0410	Shared Text. The data segment starts at the next page following the text segment and the text segment is write-protected.
ZMAGIC	0413	The object file is to be demand loaded and has a special format; the text and data segments are separated. Text segment is also write protected. (The MIPS default). The object may be either dynamic or static.
LIBMAGIC	0443	The object file is a target shared library to be demand loaded and file has a special format like that of a ZMAGIC file.

See the Object Files section in this chapter for information on the format of OMAGIC, NMAGIC, ZMAGIC, and LIBMAGIC files.

Section Headers

Table 9.6 shows the format of the Section Header (defined in the header file *scnhdr.h*).

Table 9.6: Section Header Format

Declaration	Field	Description
char	s_name[8];	Section name.
long	s_paddr;	Physical address.
long	s_vaddr;	Virtual address.
long	s_size;	Section size.
long	s_scnptr;	File pointer to raw data for section.
long	s_relptr;	File pointer to relocation.
long	s_lnnoptr;	File pointer to gp (global pointer) tables.
unsigned short	s_nreloc;	Number of relocation entries.
unsigned short	s_nlnno;	Number of gp tables.
long	s_flags;	Flags.

Section Name (s_name)

Table 9.7 shows the defined section names for the *s_name* field of the Section Header; the header file *scnhdr.h* contains the macro definitions.

Table 9.7: Section Header Constants for Section Names

Declaration	Field	Description
_TEXT	".text"	Text section.
_INIT	".init"	Initialization text section for shared libraries.
_FINI	".fini"	Cleanup text section.
_RDATA	".rdata"	Read only data section.
_DATA	".data"	Large data section.
_LIT8	".lit8"	8 byte literal pool section.
_LIT4	".lit4"	4 byte literal pool section.
_SDATA	".sdata"	Small data section.
_BSS	".bss"	Large bss section.
_SBSS	".sbss"	Small bss section.
_LIB	".lib"	Shared library information section
_UCODE	".ucode"	ucode section.
_GOT	".got"	* Global offset table.
_DYNAMIC	".dynamic"	* Dynamic linking information.
_DYNSYM	".dynsym"	* Dynamic linking symbol table.
_REL_DYN	".rel.dyn"	* Relocation information.
_DYNSTR	".dynstr"	* Dynamic linking strings.
_HASH	".hash"	* Symbol hash table.
_DSOLIST	".dsolist"	* Dynamic shared object list table.
_CONFLICT	".conflict"	* Additional dynamic linking information.
_REGINFO	".reginfo"	* Register usage information.

* these sections exist only in ZMAGIC type files and are used during dynamic linking

Flags (s_flags)

Table 9.8 sbows the defined values for *s_flags*; the header file scnhdr.h contains the definitions (those flags that are not used by compiler system object files are noted).

Table 9.8: Format of s_flags Section Header Entry

Symbol	Value	Description
STYP_REG	0x00	Regular section; allocated, relocated, loaded.
STYP_DSECT	0x01	!Dummy; not allocated, relocated, not loaded.
STYP_NOLOAD	0x02	!Noload; allocated, relocated, not loaded.
STYP_GROUP	0x04	!Grouped; formed of input sections.
STYP_PAD	0x08	!Padding; not allocated, not relocated, loaded.
STYP_COPY	0x10	!Copy; for decision function used by field update; not allocated, not relocated, loaded; relocated, and line number entries processed normally.
STYP_TEXT	0x20	Text only.
STYP_DATA	0x40	Data only.
STYP_BSS	0x80	Contains bss only.
STYP_RDATA	0x100	Read only data only.
STYP_SDATA	0x200	Small data only.
STYP_SBSS	0x400	Contains small bss only.
STYP_UCODE	0x800	Section contains ucode only.
STYP_LIT4	0x10000000	Section 4 byte literals only.
S_NRELOC_OVFL	0x20000000	s_nreloc overflowed, the value is in r_vaddr of the first entry.
STYP_LIB	0x40000000	Section contains shared library information only.
STYP_INIT	0x80000000	Section initialization text only.
STYP_FINI	0x01000000	.fini section text.
STYP_COMMENT	0x02100000	Comment section.
STYP_LIT8	0x08000000	Section 8 byte literals only.
STYP_CONFLICT	0x00100000	Additional linking information.
STYP_DSOLIST	0x00040000	Dynamic shared object list table.
STYP_HASH	0x00020000	Symbol has table.
STYP_DYNSTR	0x00010000	Dynamic linking strings.
STYP_GOT	0x00001000	Global offset table.
STYP_DYNAMIC	0x00002000	Dynamic linking information section.
STYP_DYNSYM	0x00004000	Dynamic linking symbol table.
STYP_REL_DYN	0x0008000	Relocation information for runtime linker.

!Not used by compiler system object modules.

S_NRELOC_OVFL is used when the number of relocation entries in a section overflows the *s_nreloc* field of the section header. In this case, *s_nreloc* contains the value 0xffff and the *s_flags* field has the S_NRELOC_OVFL flag set; the value true is in the *r_vaddr* field of the first relocation entry for that section. That relocation entry has a type of R_ABS and all other fields are zero, causing it to be ignored under normal circumstances.

Note: For performance reasons, the link editor uses the *s_flags* entry instead of *s_name* to determine the type of section. However, the link editor does correctly fill in the *s_name* entry.

Global Pointer Tables

The gp (global pointer) tables are part of the object file that is produced by the assembler. These are used by the link editor in calculating the best –G *num* to compile the objects are specified as recompilable by the –count option. There is a gp table for the *.sdata* and *.bss* sections only.

The gp table gives the section size corresponding to each applicable value specified by the –G *num* option (always including 0), sorted by smallest size first. The *s_lnnoptr* field in the section header points to this value and the *s_nlnno* field contains the number of entries (including the header). If there is no *small* section, the related gp table is attached to the corresponding *large* section to provide the link editor with this information.

When an object does not contain a data and bss section, the –G *num* option specified for the object at compilation is unknown. Because the size of the literal pools cannot be known, this complicates the calculation of a best –G *num*. However, a reliable calculation can be made when there is an 8–byte literal pool, which ensures that the object was compiled with a –G of at least eight.

The global pointer table has the following format:

```
union gp_table {
struct {
        long current_g_value;/* actual value */
        long unused;
} header;
        struct {
        long g_value;       /* hypothetical value */
        long bytes;         /* section size corresponding */
                            /* to hypothetical value */
        } entry;
};
```

Shared Library Information

The *.lib* section contains the shared libraries used by executable objects. The absence of a *.lib* section header indicates that no shared libraries are used. Shared libraries are a feature of System V Release 3; thus, only objects compiled with *–systype sysv* should contain *.lib* sections. The field *s_nlib* in the section header is defined to be the same as *s_paddr* and contains the number of shared library entries in the *.lib* section. The shared library information definition shown below defines a compiler system *.lib* section entry. Note the size and offset are in *sizeof(long)*'s not bytes. The size (in bytes) of each entry must be a multiple of SCNROUND. The name the *offset* field refers to is a C-null–terminated string.

```
struct libscn {
    long size;        /* size of this entry (including */
                      /* target name)*/
    long offset;      /* offset from start of entry  */
                      /* to target name*/
    long tsize;       /* text size in bytes*/
    long dsize;       /* initialized data size in bytes */
    long bsize;       /* uninitialized data size in bytes */
    long text_start; /* base of text used for */
                      /* this library*/
    long data_start; /* base of data used for */
                      /* this library */
    long bss_start;  /* base of bss used for */
                      /* this library */
    /* pathname of target shared library */
};
```

Section Data

RISCompiler system files are represented by the following sections: *.dynamic*, *.liblist*, *.rel.dyn*, *.conflict*, *.dynstr*, *.dynsym*, *.hash*, *.rdata* (read–only data), *.text*, *.init* (shared library initialization text), *.fini* (process termination text), *.data* (data), *.lit8* (8–byte literal pool), *.lit4* (4–byte literal pool), *.sdata* (small data), *.sbss* (small block started by storage), *.bss* (block started by storage), *.lib* (shared library information), and *.ucode* (intermediate code). Figure 9.2 shows the layout of the sections.

The *.dynamic*, *.liblist*, *.rel.dyn*, *.conflict*, *.dynstr*, *.dynsym*, and *.hash* sections exist only in ZMAGIC files and are used during dynamic linking. These sections are described in more detail in Chapter 11. Dynamic linking is discussed in Chapter 12.

The *.text* section contains the machine instructions that are to be executed; the *.rdata*, *.data*, *.lit8*, *.lit4*, and *.sdata* contain initialized data, and the *.sbss* and *.bss* sections reserve space for uninitialized data that is created by the kernel loader for the program before execution and filled with zeros.

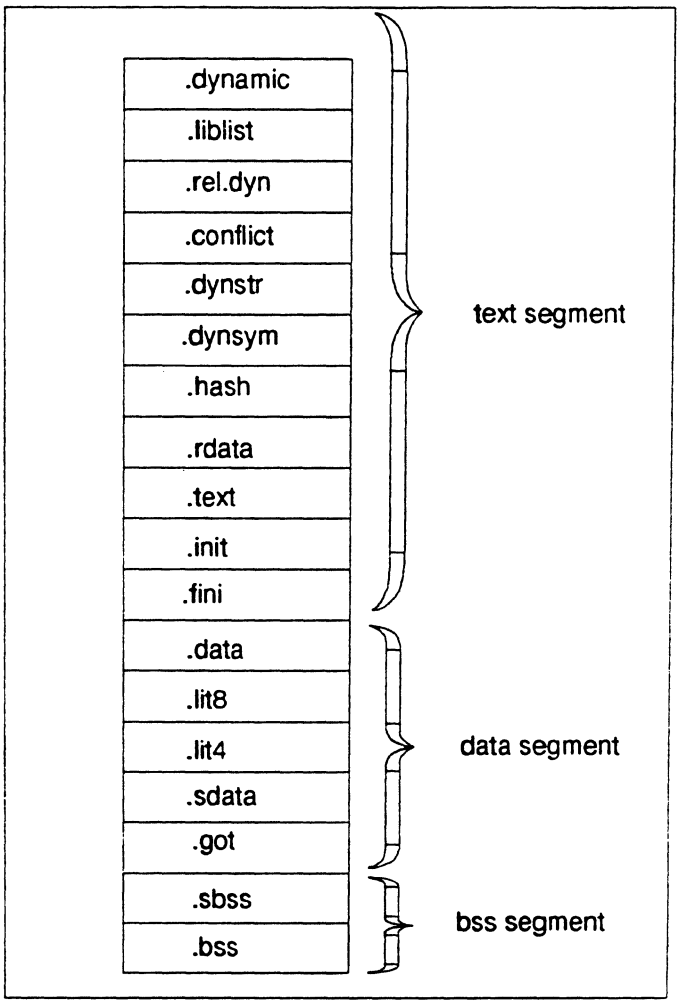


Figure 9.2: Organization of Section Data

As noted in Figure 9.2, the sections are grouped into the text segment (containing the *.text*, *.init*, and *.fini* sections), the data segment (*.rdata*, *.data*, *.lit8*, *.lit4*, and *.sdata*), and the bss segment (*.sbss* and *.bss*). A section is described by and referenced through the Section Header; the Optional Header provides the same information for segments.

The link editor references the data shown in Figure 9.2 both as sections and segments, through the Section Header and Optional Header respectively. However, the system (kernel) loader, when loading the object file at run-time, references the same data only by segment, through the Optional Header.

Section Relocation Information

Relocation Table Entry

Table 9.9 shows the format of an entry in the Relocation Table (defined in the header file *reloc.h).*

Table 9.1: Format of a Relocation Table Entry

Declaration	Field	Description
long	r_vaddr;	(Virtual) address of an item to be relocated.
unsigned	r_symndx:24,	Index into external symbols or section numbers; see r_extern below.
	r_reserved:3,	
	r_type:4,	Relocation type.
	r_extern:1;	= 1 for an external relocation entry; r_symndx is an index into External Symbols. = 0 for a local, relocation entry; r_symndx is the number of the section containing the symbol.

Symbol Index (r_symndx) and Extern Field (r_extern)

For external relocation entries, *r_extern* is set to 1 and *r_symndx* is the index into External Symbols for this entry. In this case, the value of the symbol is used as the value for relocation.

For local relocation entries, *r_extern* is set to 0, and *r_symndx* contains a constant that refers to a section. In this case, the starting address of the section to which the constant refers is used as the value for relocation.

Table 9.10 gives the section numbers for *r_symndx*; the *reloc.h* file contains the macro definitions.

Table 9.2: Section Numbers for Local Relocation Entries

Symbol	Value	Description
R_SN_TEXT	1	.text section.
R_SN_INIT	7	.init section.
R_SN_RDATA	2	.rdata section.
R_SN_DATA	3	.data section.
R_SN_SDATA	4	.sdata section.
R_SN_SBSS	5	.sbss section.
R_SN_BSS	6	.bss section.
R_SN_LIT8	8	.lit8 section.
R_SN_LIT4	9	.lit4 section.
R_SN_FINI	12	.fini section.

Relocation Type (r_type)

Table 9.11shows valid symbolic entries for the relocation type (*r_type*) field (defined in the header file *reloc.h*)

Table 9.3: Relocation Types

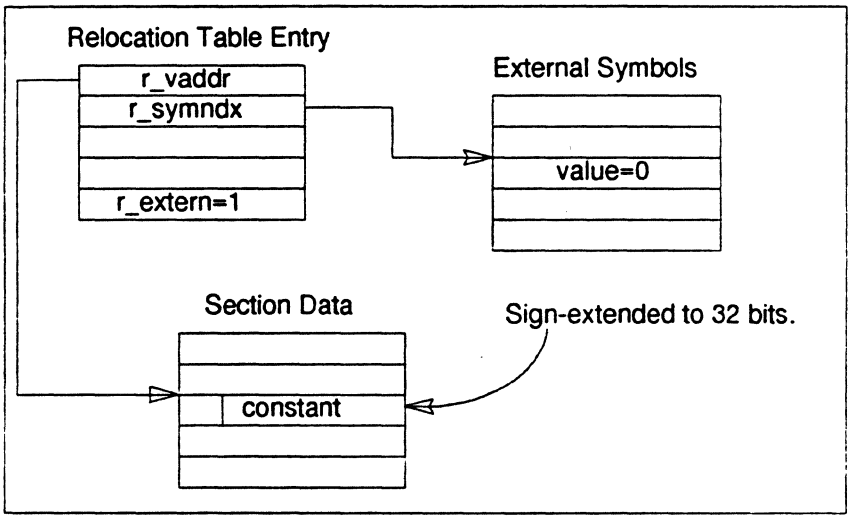
Symbol	Value	Description
R_ABS	0x0	Relocation already performed.
R_REFHALF	0x1	16–bit reference to the symbol's virtual address.
R_REFWORD	0x2	32–bit reference to the symbol's virtual address.
R_JMPADDR	0x3	26–bit jump reference to the symbol's virtual address.
R_REFHI	0x4	Reference to the high 16–bits of symbol's virtual address.
R_REFLO	0x5	Reference to the low 16–bits of symbol's virtual address.
R_GPREL	0x6	Reference to the offset from the global pointer to the symbol's virtual address.
R_LITERAL	0x7	Reference to a literal in a literal pool as an offset from the global pointer.
R_REL32	0x8	Reference to the distance from the offset to the global pointer.

Assembler and Link Editor Processing

Compiler system executable object modules with all external references defined have the same format as relocatable modules and are executable without re–link editing.

Local relocation entries must be used for symbols that are defined. Therefore, external relocations are used only for undefined symbols. Figure 9.3 gives an overview of the Relocation Table entry for an undefined external symbol.

Figure 9.3: Relocation Table Entry for Undefined External Symbols



The assembler creates this entry as follows:

Sets *r_vaddr* to point to the item to be relocated.

Places a constant to be added to the value for relocation at the address for the item to be relocated (*r_vaddr*).

Sets *r_symndx* to the index of the External Symbols entry that contains the symbol value.

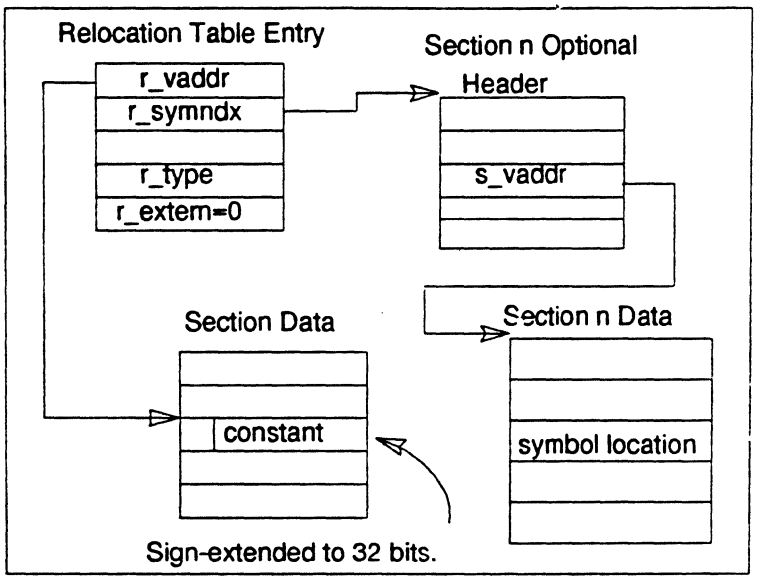
Sets *r_type* to the constant for the type of relocation types. Table 9.11 shows the valid constants for the relocation type.

Sets *r_extern* to 1.

Note: The assembler always sets the value of the undefined entry in External Symbols to 0. It may assign a constant value to be added to the relocated value at the address where the location is to be done. If the width of the constant is less than a full word, and an overflow occurs after relocation, the link editor flags this as an error.

When the link editor determines that an external symbol is defined, it changes the Relocation Table entry for the symbol to a local relocation entry. Figure 9.4 gives an overview of the new entry.

Figure 9.4: Relocation Table Entry for a Local Relocation Entry



To change this entry from an external relocation entry to a local relocation entry, the link editor:

- Picks up the constant from the address to be relocated (*r_vaddr*).
- If the width of the constant is less than 32 bits, sign-extends the constant to 32 bits.
- Adds the value for relocation (the value of the symbol) to the constant and places it back in the address to be relocated.
- Sets *r_symndx* to the section number that contains the external symbol.
- Sets *r_extern* to 0.

Examples

The examples that follow use external relocation entries.

Example 1: 32-Bit Reference—R_REFWORD. This example shows assembly statements that set the value at location *b* to the global data value *y*.

```
    .globl y
    .data
b: .word y #R_REFWORD relocation type at address b for symbol y
```

In processing this statement, the assembler generates a relocation entry of type R_REFWORD for the address *b* and the symbol *y*. After determining the address for the symbol *y*, the loader adds the 32-bit address of y to the 32-bit value at location *b*, and places the sum in location *b*. The loader handles 16-bit addresses (R_REFHALF) in the same manner, except it checks for overflow after determining the relocation value.

Example 2: 26-Bit Jump—R_JMPADDR. This example shows assembly statements that call routine *x* from location *c*.

```
    .text
x: #routine x
    ...
c: jal x #R_JMPADDR relocation type at address c for symbol x
```

In processing these statements, the assembler generates a relocation entry of type R_JMPADDR for the address and the symbol *x*. After determining the address for the routine, the loader shifts the address right two bits, adds the low 26 bits of the result to the low 26 bits of the instruction at address *c* (after sign-extending it), and places the results back into the low 26 bits at address *c*.

R_JMPADDR relocation entries are produced for the assembler's *j* (Jump) and *jal* (Jump and Link) instructions. These instructions take the high four bits of the target address from the address of the delay slot of their instruction. The link editor makes sure that the same four bits are in the target address after relocation; if not, it generates an error message.

If the entry is a local relocation type, the target of the Jump instruction is assembled in the instruction at the address to be relocated. The high four bits of the jump target are taken from the high 4 bits of the address of the delay slot of the instruction to be relocated.

Example 3: High/Low Reference-R_REFHI/R_REFLO. This example shows an assembler macro that loads the absolute address *y*, plus a constant, into Register 6:

In processing this statement, the assembler generates a 0 as the value *y*, and the following machine language statements:

```
f: lui $at,constant>>16            #R_REFHI relocation type at address
f
                                   for symbol y
g: addiu $r6,constant&0xffff($at)   #R_REFLO relocation type at address
```

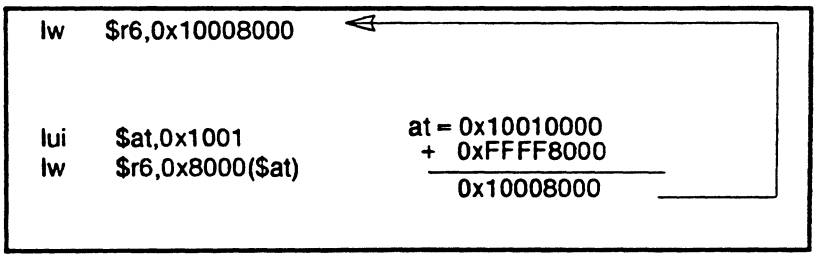
In this example, the assembler produces two relocation entries.

Note: When a R_REFHI relocation entry appears, the next relocation entry must always be the corresponding R_REFLO entry. This is required in order to reconstruct the constant that is to be added to the value for relocation.

In determining the final constant values for the two instructions, the link editor must take into account that the *addiu* instruction of the R_REFLO relocation entry sign–extends the immediate value of the constant.

In determining the sum of the address for the symbol *y* and the constant, the link editor does the following:

- It uses the low 16 bits of this sum for the immediate value of the R_REFLO relocation address.
- Because all instructions that are marked with a R_REFLO perform a signed operation, the assembler adjusts the high portion of the sum if Bit 15 is set. Then it uses the high 16 bits of the sum for the immediate value of the R_REFHI instruction at the relocation address. For example:



Example 4: Offset Reference—R_GPREL. This example shows an assembly macro that loads a global pointer relative value *y* into Register 6:

```
lw $r6,y
```

In processing this statement, the assembler generates a 0 as the value *y* and the following machine language statement:

```
h: lw $r6,0($gp)          #R_GPREL relocation type at
                          address h for symbol y
```

and a R_GPREL relocation entry would be produced. The assembler then uses the difference between the address for the symbol y and the address of the global pointer, as the immediate value for the instruction. The link editor gets the value of the global pointer used by the assembler from gp_value in the Optional Header (Table 9.4).

Example 4: Example of the R_LITERAL. This example shows of an R_LITERAL uses a floating-point literal. The assembler macro:

```
li.s $f0,1.234
```

is translated into the following machine instruction:

```
h:      lwc1 $f0,-32752(gp) # R_LITERAL relocation
                            type at address h for the
                            literal 1.234
```

and a R_LITERAL relocation entry is produced; the value of the literal is put into the *.lit4* section. The link editor places only one of all like literal constants in the literal pool. The difference between the virtual address of the literal and the address of the global pointer is used as the immediate value for the instruction. The link editor handles 8-byte literal constants similarly, except it places each unique constant in the .lit8 section. The value of the -G num option used when compiling determines if the literal pools are used.

Object Files

This section describes the object-file formats created by the link editor, namely the Impure (OMAGIC), Shared Text (NMAGIC), Demand Paged (ZMAGIC), and target-shared libraries (LIBMAGIC) formats. Before reading this section, you should be familiar in the format and contents of the text, data, and bss segments as described in the Section Data section of this chapter.

Note: This chapter discusses the creation of LIBMAGIC files (shared libraries). These are not to be confused with dynamic shared objects that have type ZMAGIC. Dynamic shared objects are discussed in Chapters 11 and 12.

The following constraints are imposed on the address at which an object can be loaded and the boundaries of its segments. The operating system can dictate additional constraints.

- Segnments must not overlap and all addresses must be less than 0x80000000.
- Space should be reserved for the stack, which starts below 0x80000000 and grows through lower addresses; that is, the value of each subsequent address is less than that of the previous address.
- The default text segment address for ZMAGIC and NMAGIC files is 0x0n0400000 and the default data segment address is 0x10000000.
- The default text segment address for OMAGIC files is 0x10000000 with the data segment following the text segment.
- The -B num option (specifying a bss segment origin) cannot be specified for OMAGIC files; the default, which specifies that the bss segment follow the data segment, must be used.
- RISC/os requires a 2-megabyte boundary for segments.

Impure Format (OMAGIC) Files

An OMAGIC file has the format shown in Figure 9.5.

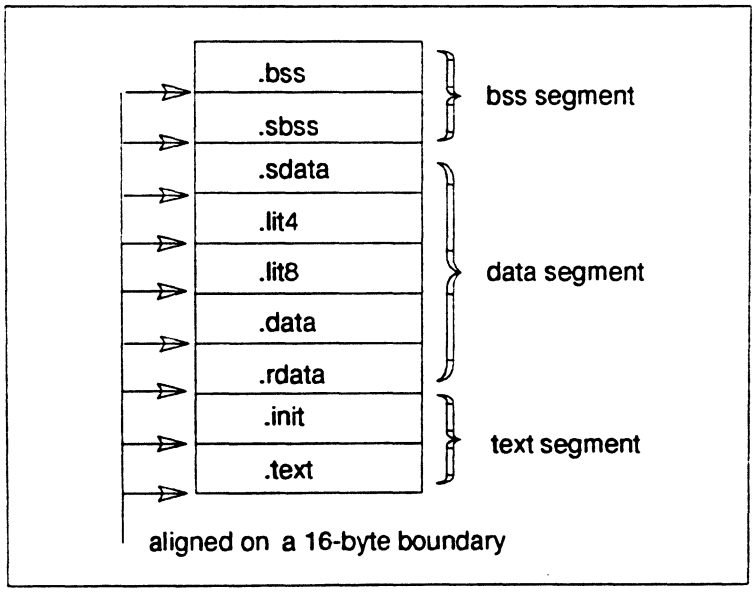


Figure 9.5: Layout of OMAGIC Files in Virtual Memory

The OMAGIC format hasb the following characteristics:

- Each section follows the other in virtual address space aligned on an 16–byte boundary.
- No blocking of sections.
- Text, data and bss segments can be placed anywhere in the virtual address space using the link editor's —T, —D and —B options.
- The addresses specified for the segments must be rounded to 16–byte boundaries.
- The text segment contains the *.text*, and *.init* sections.
- The sections in the data segment are ordered as follows: *.rdata*, *.data*, *.lit8*, *.lit4*, and *.sdata*.
- The sections in the bss segment are ordered as follows: *.sbss* and *.bss*.

Shared Text (NMAGIC) Files

An NMAGIC file has the format shown in Figure 9.6.

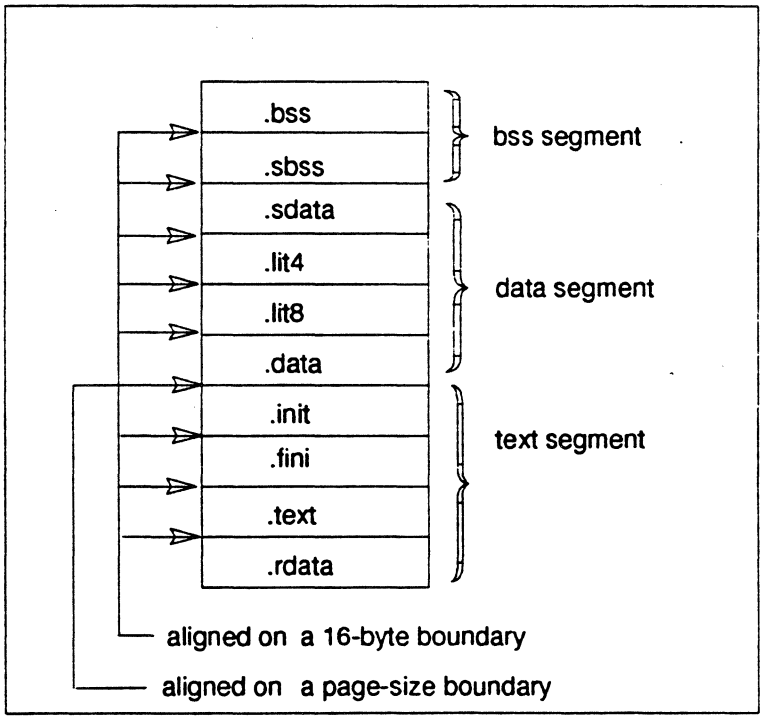


Figure 9.6: Layout of NMAGIC Files in Virtual Memory

An NMAGIC file has the following characteristics:

- The virtual address of the data segment is on a *pagesize* boundary.
- No blocking of sections.
- Each section follows the other in virtual address space aligned on an 16–byte boundary.
- Only the start of the text and data segments, using the link editor's –T and –D options, can be specified for a shared text format file; the start of the text and data segments must be a multiple of the *pagesize*.

Demand Paged (ZMAGIC) Files

A ZMAGIC file is a demand paged file in the format shown in Figure 9.7.

A ZMAGIC file has the following characteristics:

- The text segment and the data segment are blocked, with *pagesize* as the blocking factor. Blocking reduces the complexity of paging in the files.
- The size of the sum of the of the File, Optional, and Sections Headers (Tables 9.1, 9.4, and 9.6) rounded to 16 bytes is included in blocking of the text segment.
- The text segment starts by default at 0x400000 (4 Mbyte) , plus the size of the sum of the headers again rounded to 16 bytes. With the standard software, the text segment starts at 0x400000 + header size.

Note: This is required because the first 32K bytes of memory are reserved for future use by the compiler system to allow data access relative to the constant register 0.

- Only the start of the text and data segments, using the link editor's –T and –D options can be specified for a demand paged format file and must be a multiple of the pagesize.

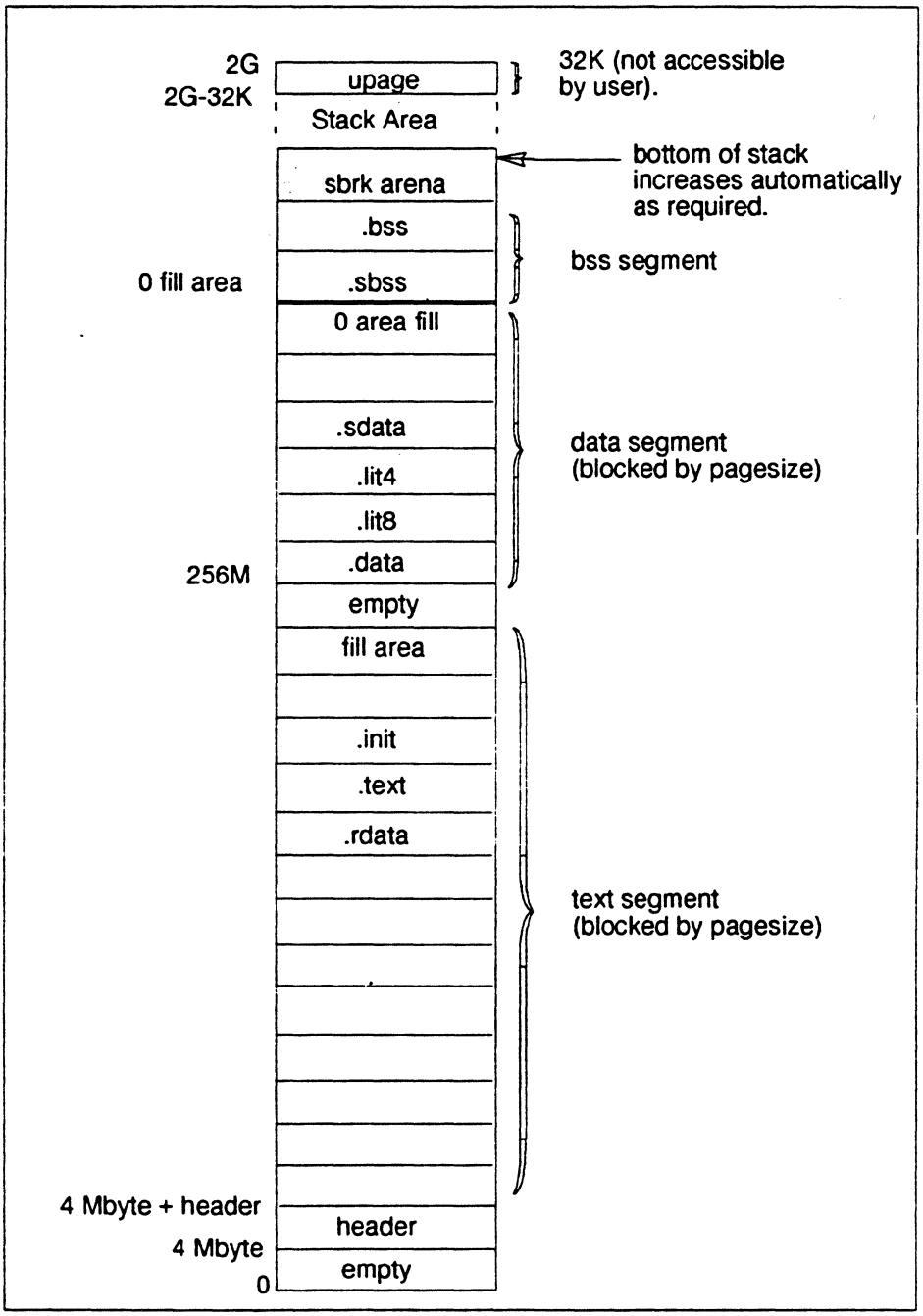


Figure 9.7: Layout of ZMAGIC Files in Virtual Memory

Figure 9.8 shows a ZMAGIC file as it appears in a disk file.

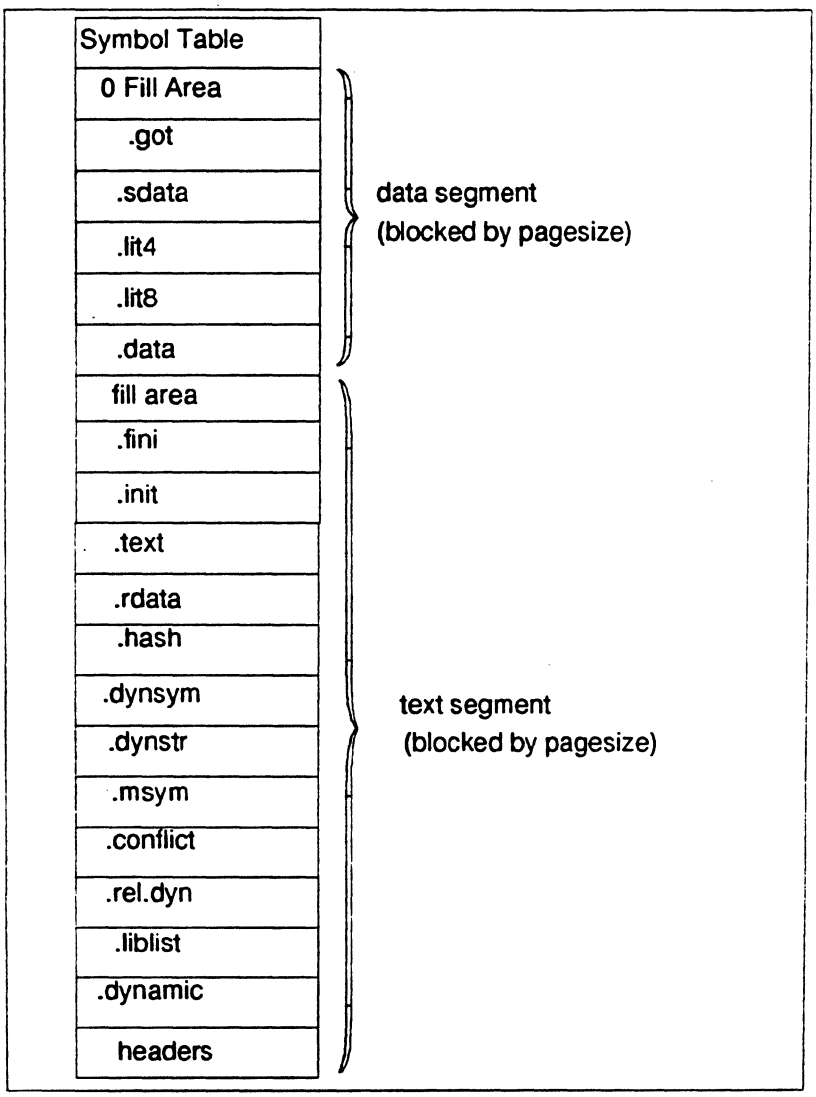


Figure 9.8: Layout of a ZMAGIC File on Disk.

Target Shared Library (LIBMAGIC) Files

Typically, *mkshlib*(1) creates target shared libraries. The link editor creates such libraries when its –c option is specified (each shared library file name is displayed during the link if the –v option is supplied).

LIBMAGIC files are demand paged and have the same format as ZMAGIC file except as follows:

- Headers are put on their own page
- The text section starts on the next page from the value of the –T *num* option. This prevents the number and size of headers from affecting the start of the first real text. The first real text is the branch table and must stay at the same address.

Both the –T and –D options should be specified, because the defaults would cause the target shared library to overlay the ZMAGIC files and cause an execution failure. The link editor –c option requires that the files to be linked are compiled with the –G 0 option (which sets the link editor –G 0 option).

Note: Shared library refers to System V Release 3 type shared libraries. Elsewhere, we use "shared objects" or "dynamically linked executable" for shared libraries in the sense similar to System V Release 4 type shared libraries.

Objects Using Shared Libraries

Object files that use shared libraries contain a *.lib* section following the data segment (including the zero fill area created by blocking it to a pagesize). All object file contain an *.init* section used by shared library initialization code. Shared library initialization instructions are generated by *mkshlib*(1) from the *#init* directive in the library specification file. This following code from the shared library specification

```
#init                    bar.o
_libfoo_ext              ext
```

generates these instructions generated in the *.init* section:

```
la      $2,ext
sw      $2,_libfoo_ext
```

Initialization instructions are not bounded by any procedure; the initialization instructions from each *.init* section are concatenated and the runtime startup (*crt1.o*) branches to its label in its *.init* section. Then the execution falls through all the concatenated *.init* sections until reaching *crtn.o* (the last object with a *.init* section) which contains the RETURN instruction.

Object files without shared libraries contain a small *.init* section that executes, producing no significant results.

Ucode objects

Ucode objects contain only a file header, the ucode section header, the ucode section and all of the symbolic information. A ucode section never appears in a machine code object file.

Loading Object Files

The link editor produces object files with their sections in a fixed order similar to UNIX system object files that existed before COFF. See Figure 9.1 for the a description of the sections and how they are formatted.

The sections are grouped into segments, which are described in the Optional Header. In loading the object module at run–time, the system (kernel) loader needs only the magic number in the File Header and the Optional Header to load an object file for execution.

The starting addresses and sizes of the segments for all types of object files are specified similarly, and they are loaded in the same manner.

After reading in the File Header and the Optional Header, the system (kernel) loader must examine the file magic number to determine if the program can be loaded. Then, the system (kernel) loader loads the text and data segments.

The starting offset in the file for the text segment is given by the macro

```
N_TXTOFF(f,a)
```

in the header file *a.out.h*, where *f* is the File Header structure and *a* is the option header structure for the object file to be loaded. The *tsize* field in the Optional Header (Table 9.4) contains the size of the text segment and *text_start* contains the address at which it is to be loaded.

The starting offset of the data segment follows the text segment. The *dsize* field in the Section Header (Table 9.6) contains the size of the data segment; *data_start* contains the address at which it is to be loaded.

The system (kernel) loader must fill the *.bss* segment with zeros. The *bss_start* field in the Optional Header specifies the starting address; *bsize* specifies the number of bytes to be filled with zeros. In ZMAGIC files, the link editor adjusts *bsize* to account for the zero filled area it created in the data segment that is part of of the *.sbss* or *.bss* sections.

If the object file itself does not load the global pointer register it must be set to the *gp_value* field in the Optional Header (Table 9.4).

The other fields in the Optional Header are *gprmask* and *cprmask[4]*, whose bits show the registers used in the *.text*, *.init* , and *.fini* sections. They can be used by the operating system, if desired, to avoid save register relocations on context–switch.

Archive files

The link editor can link object files in archives created by the archiver. The archiver and the format of the archives are based on the UNIX System V portable archive format. To improve performance, the format of the archives symbol table was changed so that it is a hash table, not a linear list.

The archive hash table is accessed through the *ranhashinit()* and *ranlookup()* library routines in *libmld.a*, which are documented in the manual page *ranhash(3x)*. The archive format definition is in the header file *ar.h*.

Link Editor Defined Symbols

Certain symbols are reserved and their values are defined by the link editor. A user program can reference these symbols, but can not define one; an error is generated if a user program attempts to define one of these symbols. Table 9.12 lists the names and values of these symbols; the header file *sym.h* contains their preprocessor macro definitions.

Table 9.12: Link Editor Defined Symbols

Symbol	Value	Description
_ETEXT	"etext"	1st location after .text
_EDATA	"edata"	1st location after .sdata (all initialized data)
_END	"end"	1st location after .bss (all data)
_FTEXT	"_ftext"	!1st location of .text
_FDATA	"_fdata"	!1st location of .data
_FBSS	"_fbss"	!1st location of the .bss
_PROCEDURE_TABLE	"_procedure_table"	runtime procedure table
_PROCEDURE_TABLE_SIZE	"_procedure_table_size"	runtime procedure table size
_PROCEDURE_STRING_TABLE	"_procedure_string_table"	string table for runtime proc.
_COBOL_MAIN	"_cobol_main"	1st cobol main symbol
_GP	"_gp"	!the value of the global pointer
__UNWIND	"__unwind"	Unwinds the stack.
__PC_NLC_GOTO	"__pc_nlc_goto"	Handles Pascal non local goto commands.
__FIND_RDP	"__find_rdp"	Finds runtime procedure tables.

!compiler system only.

The dynamic linker also reserves and defines certain symbols; see Chapters 11 and 12 for more information.

The first three symbols come from the standard UNIX system link editors and the rest are compiler system specific. The last symbol is used by the start up routine to set the value of the global pointer, as shown in the following assembly language statements:

```
globl       _GP
la          $gp,_GP
```

The assembler generates the following machine instructions for these statements:

```
a:   lui   gp,0  # R_REFHI relocation type at address a for symbol _GP
b:   add   gp,0  # R_REFLO relocation type at address b for symbol _GP
```

which would cause the correct value of the global pointer to be loaded.

The link editor symbol _COBOL_MAIN is set to the symbol value of the first external symbol with the *cobol_main* bit set. COBOL objects uses this symbol to determine the the main routine.

Runtime Procedure Table Symbols

The five link editor defined symbols, _PROCEDURE_TABLE, __FIND_RDP __PC_NLC_GOTO,, __UNWIND, _PROCEDURE_TABLE_SIZE and _PROCEDURE_STRING_TABLE, relate to the runtime procedure table. The Runtime Procedure Table is used by the exception systems in ADA, PL/I and COBOL. Its description is found in the header file *sym.h*. The table is a subset of the Procedure Descriptor Table portion of the Symbol Table with one additional field, *exception_info*.

When the procedure table entry is for an external procedure, and an External Symbol Table exists, the link editor fills in *exception_info* with the address of the external table. Otherwise, its fill in *exception_info* with zeros.

The name of the External Symbol Table is the procedure name concatenated with the string *_exception_info* (actually, the preprocessor macro EXCEPTION_SUFFIX as defined in the header file *exception.h*).

The Runtime Procedure Table provides enough information to allow a program to unwind its stack. It is typically used by the routines in *libexc.a*.

The comments in the header file *exception.h* describes the routines in that library.

The Runtime Procedure Table is sorted by procedure address and always has a dummy entry with a zero address and a 0xffffffff address. When required, the table is padded with an extra zero entry to ensure that the total number of entries is an uneven (odd) number.

The Runtime Procedure Table and String Table for the runtime procedure table are placed at then end of the *.data* section in the object file.

Symbol Table

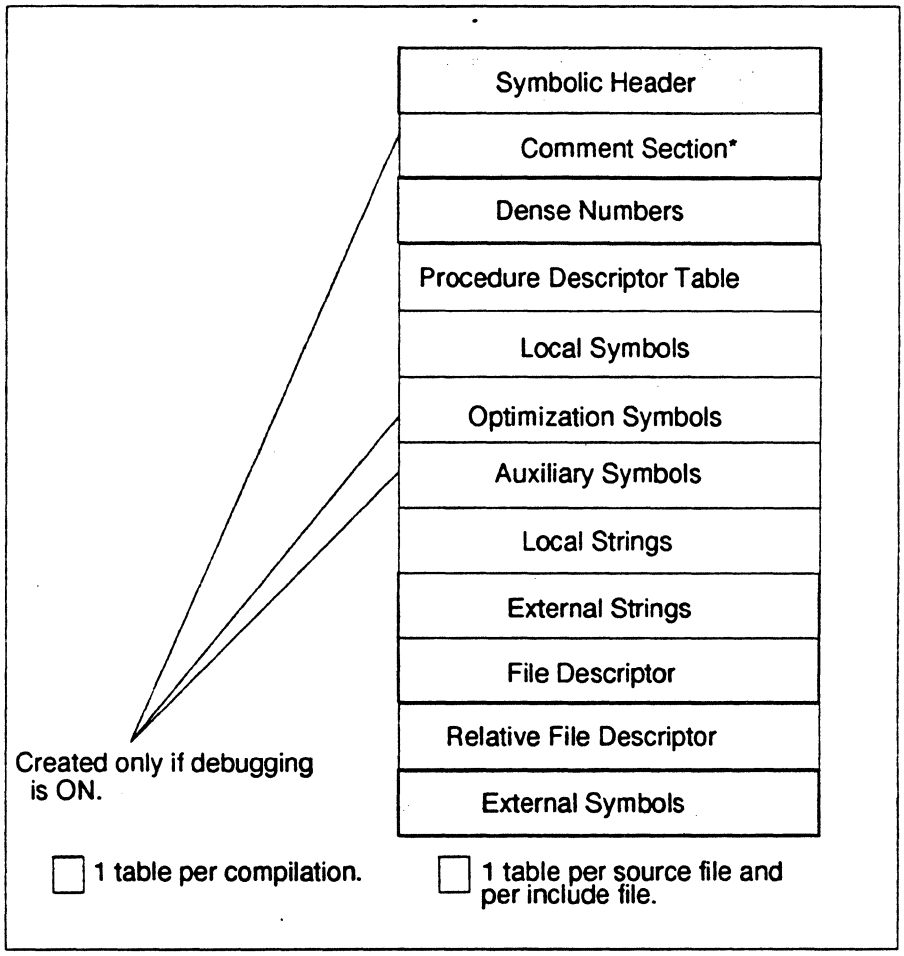
10

This chapter describes the symbol table and symbol table routines used to create and make entries in the table. The chapter contains the following major sections:

- Overview, which gives the purpose of the Symbol table, a summary of its components, and their relationship to each other.
- Format of Symbol Table Entries, which shows the structures of Symbol table entries and the values you assign them through the Symbol Table routines.
- Symbol Table Routine Reference, which lists the symbol table routines supplied with the compiler and summarizes the function of each.

Overview

The symbol table in created by the compiler front–end as a stand–alone file. The purpose of the table is to provide information to the link editor and the debugger in performing their respective functions. At the option of the user, the link editor includes information from the Symbol table in the final object file for use by the debugger. See Figure 9.1 in Chapter 9 for details.



* This section holds compacted form of relocation entries for Pixie. Currently only dynamically linked executables and shared objects have this section.

Figure 10.1: The Symbol Table - Overview

The elements that make up the Symbol table are shown in Figure 10.1. The front–end creates one group of tables (the shaded areas in Figure 10.1) that

contain global information relative to the entire compilation. It also creates a unique group of tables (the unshaded areas in the figure) for the source file and each of its include files.

Compiler front–ends, the assembler, and the link editor interact with the symbol table as summarized below:

- The front–end, using calls to routines supplied with the compiler system, enters symbols and their descriptions in the table.
- The assembler fills in line numbers, optimization symbols, updates Local Symbols and External Symbols, and updates the Procedure Descriptor table.
- The link editor eliminates duplicate information in the External Symbols and the External Strings tables, removes tables with duplicate information, updates Local Symbols with relocation information, and creates the Relative File Descriptor table.

The major elements of the table are summarized in the paragraphs that follow. Some of these elements are explored in more detail later in the chapter.

Symbolic Header. The Symbolic Header (HDRR for HeadDeR Record) contains the sizes and locations (as an offset from the beginning of the file) of the subtables that make up the Symbol Table. Figure 10.2 shows the symbolic relationship of the header to the other tables.

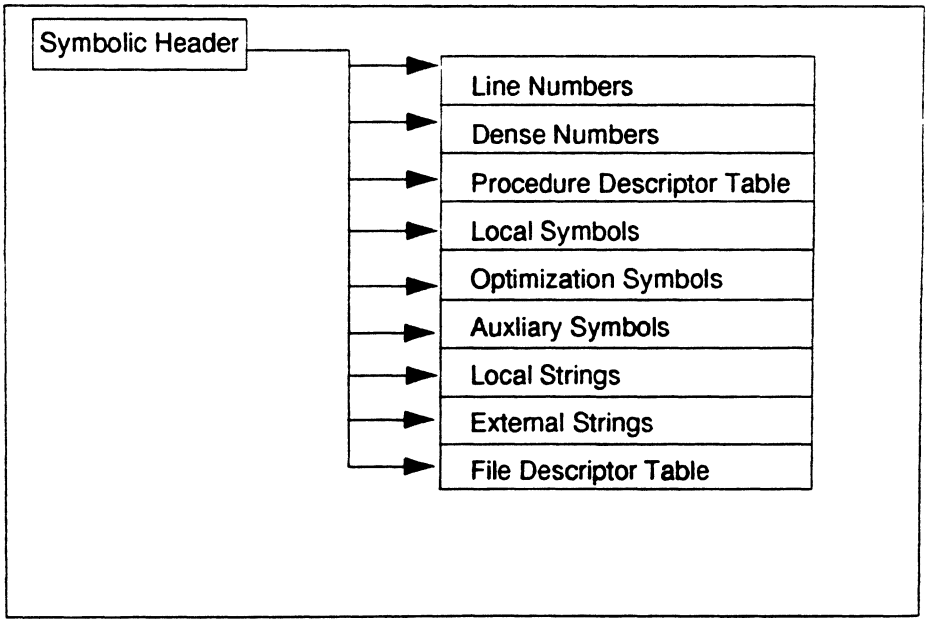


Figure 10.2: Functional Overview of the Symbolic Header

Line Numbers. The assembler creates the Line Number table. It creates an entry for every instruction. Internally, the information is stored in an encoded form. The debugger uses the entries to map instruction to the source lines and vice versa.

Dense Numbers. The Dense Number table is an array of pairs. An index into this table is called a dense number. Each pair consists of a file table index (*ifd*) and an index (*isym*) into Local Symbols. The table facilitates symbol look–up for the assembler, optimizer, and code generator by allowing direct table access rather than hashing.

Procedure Descriptor Table. The Procedure Descriptor table contains register and frame information, and offsets into other tables that provide detailed information on the procedure. The front–end creates the table and links it to the Local Symbols table. The assembler enters information on registers and frames. The debugger uses the entries in determining the line numbers for procedures and frame information for stack traces.

Local Symbols. The Local Symbols table contains descriptions of program variables, types, and structures, which the debugger uses to locate and interpret runtime values. The table gives the symbol type, storage class, and offsets into other tables that further define the symbol.

A unique Local Symbols table exists for every source and include file; the compiler locates the table through an offset from the file descriptor entry that exists for every file. The entries in Local Symbols can reference related information in the Local Strings and Auxiliary Symbols subtables. This relationship is shown in Figure 10.3.

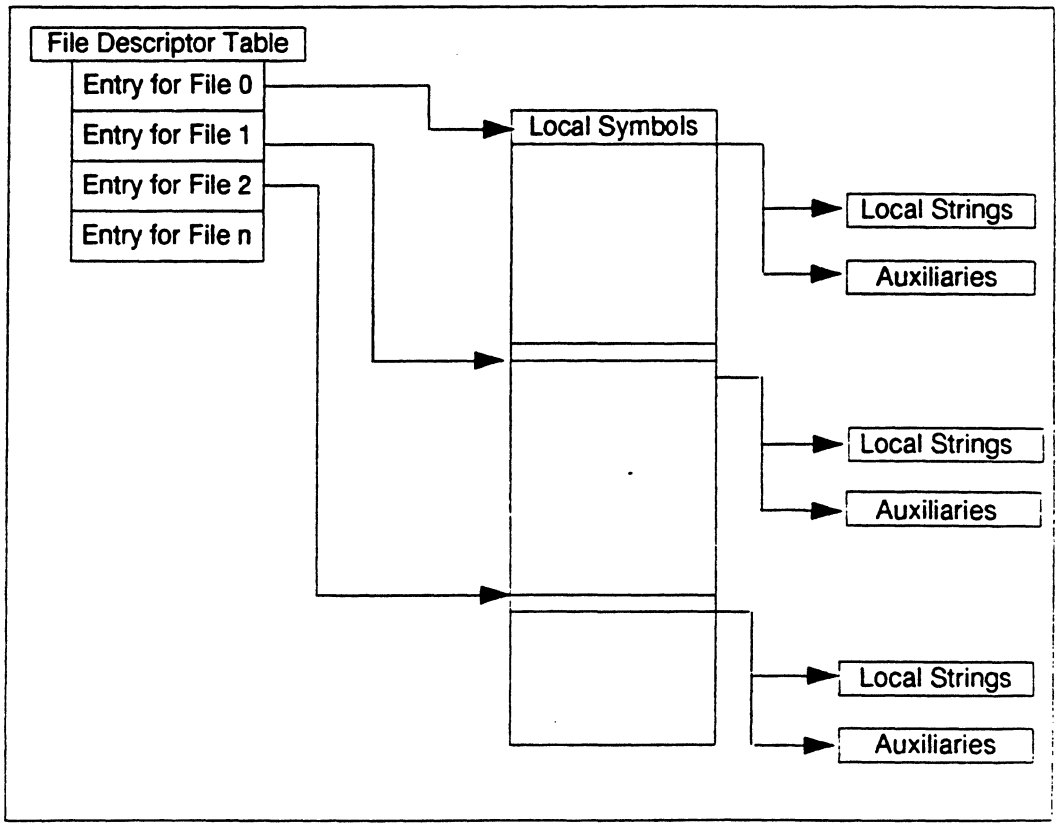


Figure 10.3: Logical Relationship between the File Descriptor Table and Local Symbols

Optimization Symbols. To be defined at a future date.

Auxiliary Symbols. The Auxiliary Symbols tables contain data type information specific to one language. Each entry is linked to an entry in Local Symbols. The entry in Local Symbols can have multiple, contiguous entries. The format of an auxiliary entry depends on the symbol type and storage class. Table entries are required only when the compiler debugging option is ON.

Local Strings. The Local Strings subtables contain the names of local symbols.

External Strings. The External Strings table contains the names of external symbols.

File Descriptor. The File Descriptor table contains one entry each for each source file and each of its include files. (The structure of an entry is given in Table 10.12 later in this chapter.) The entry is composed of pointers to a group of subtables related to the file. The physical layout of the subtables is shown in Figure 10.4.

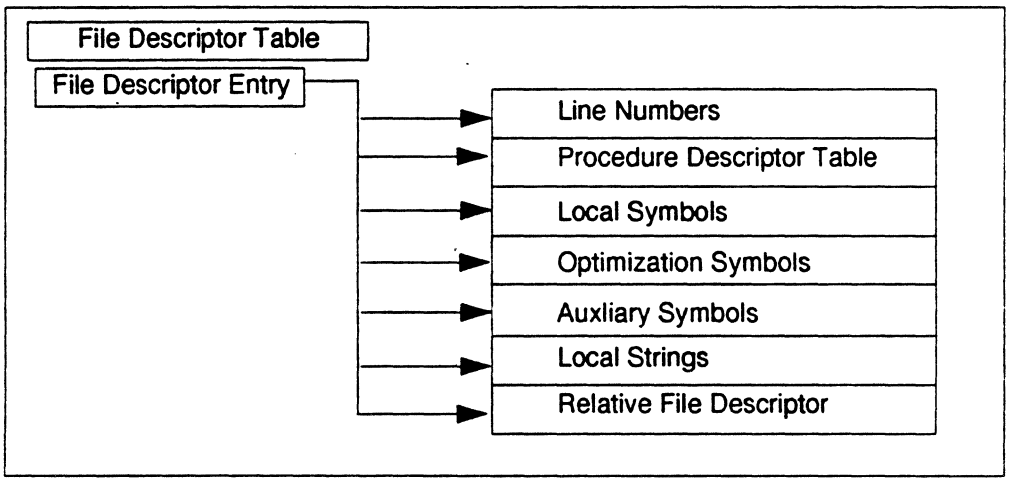


Figure 10.4: Physical Relationship of a File Descriptor Entry to Other Tables

The file descriptor entry allows the compiler to access a group of subtables unique to one file. The logical relationship between entries in this table and in its subtables is shown in Figure 10.5.

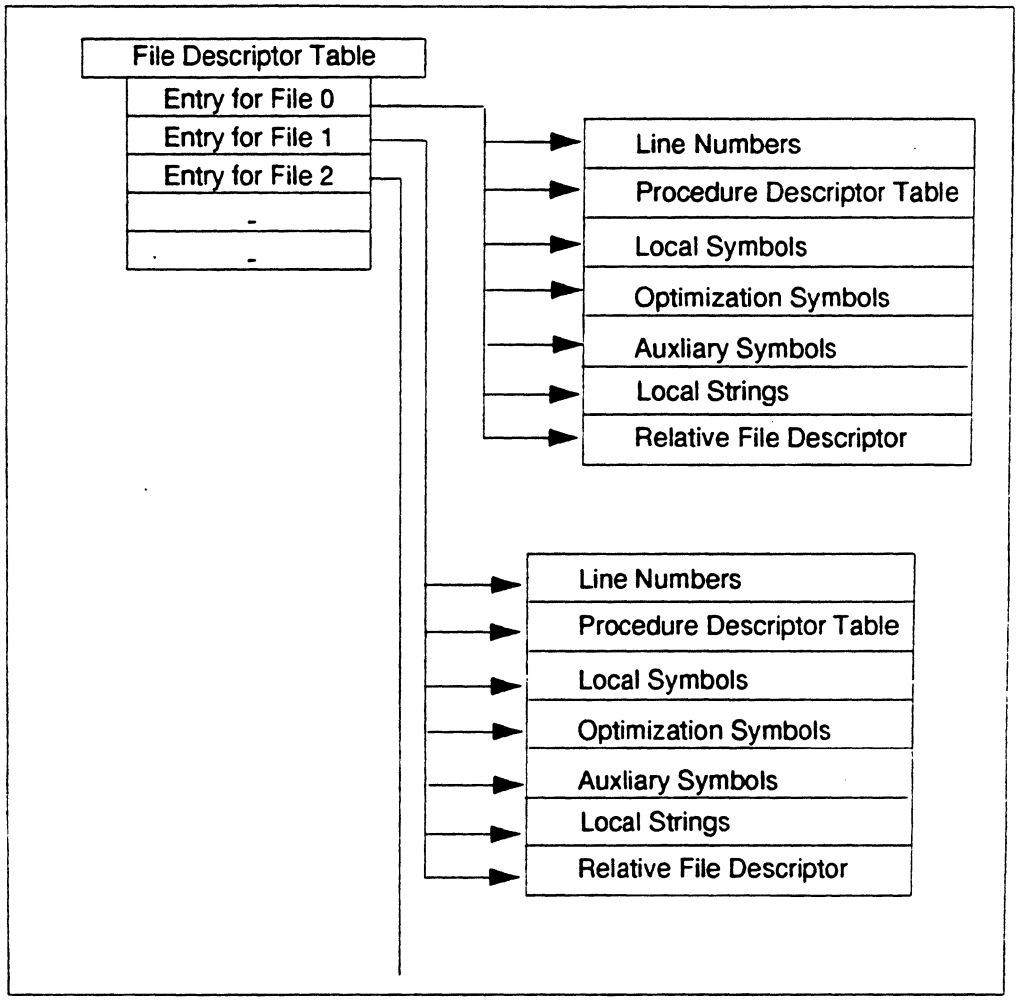


Figure 10.5: Logical Relationship between the File Descriptor Table and Other Tables

Relative File Descriptor. See the section Link Editor Processing later in this chapter.

External Symbols. The External Symbols contains global symbols entered by the front–end. The symbols are defined in one module and referenced in one or more other modules. The assembler updates the entries, and the link editor merges the symbols and resolves their addresses.

Format of Symbol Table Entries

Symbolic Header

The structure of the Symbolic Header is shown below in Table 10.1; the *sym.h* header file contains the header declaration.

Table 10.1: Format of the Symbolic Header

Declaration	Name	Description
short	magic	To verify validity of the table.
short	vstamp	Version stamp.
long	ilineMax	Number of line number entries.
long	cbLine	Number of bytes for line number entries.
long	cbLineOffset	Index to start of line numbers.
long	idnMax	Max index into dense numbers.
long	cbDnOffset	Index to start dense numbers.
long	ipdMax	Number of procedures.
long	cbPdOffset	Index to procedure descriptors.
long	isymMax	Number of local symbols.
long	cbSymOffset	Index to start of local symbols.
long	ioptMax	Maximum index into optimization entries.
long	cbOptOffset	Index to start of optimization entries.
long	iauxMax	Number of auxiliary symbols.
long	cbAuxOffset	Index to the start of auxiliary symbols.
long	issMax	Max index into local strings.
long	cbSsOffset	Index to start of local strings.
long	issExtMax	Max index into external strings.
long	cbSsExtOffset	Index to the start of external strings.
long	ifdMax	Number of file descriptors.
long	cbFdOffset	Index to file descriptor.
long	crfd	Number of relative file descriptors.
long	cbRfdOffset	Index to relative file descriptors.
long	iextMax	Maximum index into external symbols.
long	cbExtOffset	Index to the start of external symbols.

The lower byte of the vstamp field contains LS_STAMP and the upper byte MS_STAMP (see the *stamp.h* header file). These values are defined in the *stamp.h* file. The *iMax* fields and the *cbOffset* field must be set to 0 if one of the tables shown in Table 10.1 isn't present. The *magic* field must contain the constant *magicSym*, also defined in *longsymconst.h*.

Line Numbers

Table 10.2 shows the format of an entry in the Line Numbers table; the *sym.h* header file contains its declaration.

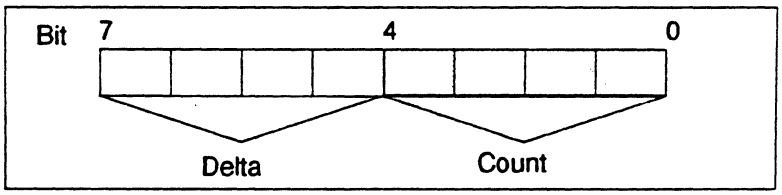
Table 10.2: Format of a Line Number Entry

Declaration	Name
typedef long	LINER, *pLINER

The line number section in the Symbol table is rounded to the nearest four-byte boundary.

Line numbers map executable instructions to source lines; one line number is stored for each instruction associated with a source line. It is stored as a long integer in memory and in packed format on disk.

The layout on disk is as follows:



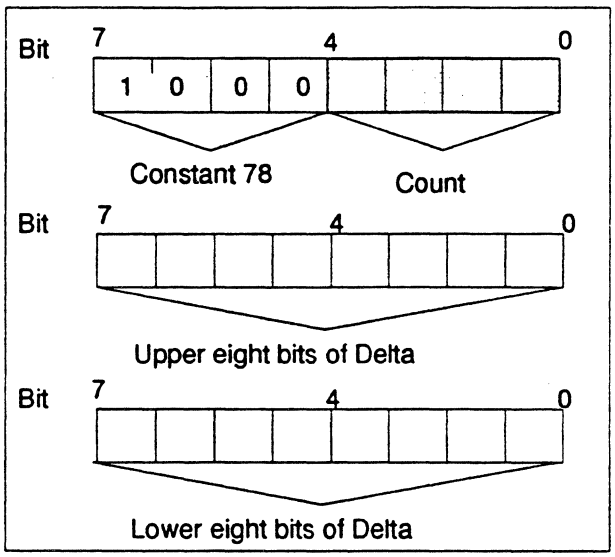
The compiler assigns a line number to only those lines of source code that generate one or more executable instructions.

Delta is a four-bit value in the range -7...7, defining the number of source lines between the current source line, and the previous line generating executable instructions. The Delta of the first line number entry is the displacement from the *lnLow* field in the Procedure Descriptor Table.

Count is a four-bit field with a value in the range 0...15 indicating the number (1...16) of executable instructions associated with a source line. If more than 16 instructions (15+1) are associated with a source line, new line number entries are generated with Delta = 0.

An extended format of the line number entry is used when Delta is outside the range of -7...7.

The layout of the extended field on disk is as follows:



Note: The compiler allows a maximum of 32,767 comment lines, blank lines, continuation lines and other lines *not* producing executable instructions, between two source lines that do produce executable instructions.

Line number example. This section gives an example of how the compiler assigns line numbers. For the source listing shown below, the compiler generates line numbers only for the highlighted lines (6, 7, 17, 18, and 19); the other lines are either blank or contain comments.

```
 1   #include <stdio.h>
 2   main ()
 3   {
 4       char c;
 5
 6       printf ("this program just prints its input\n");
 7       while ((c = getc(stdin)) != EOF) {
 8            /* this is a greater than a seven line comment
 9             * 1
10             * 2
11             * 3
12             * 4
13             * 5
14             * 6
15             * 7
16                  */
17                  printf("%c", c);
18       } /* end while */
19   } /* end main */
```

Figure 10.6: Source Listing for Line Number Example

Figure 10.8 (on the next page) shows the instructions generated for lines 3, 7, 17, 18, and 19. Figure 10.7 (below) shows the compiler–generated liner entries for each source line.

Source Line	Liner	
	Contents	Meaning
3	02	delta 0, count 2
6	31	delta 3, count 1
7	1f	delta 1, count 15
7	03	delta 0, count 3
17[1]	82 00 0a	-8[1], count 2, delta 10
18	1f	delta 1, count 15
18[2]	03	delta 0[2], count 3
19	15	delta 1, count 5

[1] Extended format (count is greater than seven lines).
[2] Continuation.

Figure 10.7: Source Listing for Line Number Example

```
[main:3, 0x4001a0]      addiu       sp,sp,-32
[main:3, 0x4001a4]      sw          r31,20(sp)          3 ,6
[main:3, 0x4001a8]      sw          r16,16(sp)
[main:6, 0x4001ac]      jal         printf
[main:6, 0x4001b0]      addiu       r4,gp,-32752
[main:7, 0x4001b4]      lw          r14,-32552(gp)
[main:7, 0x4001b8]      nop
[main:7, 0x4001bc]      addiu       r15,r14,-1
[main:7, 0x4001c0]      bltz        r15,0x4001e4
[main:7, 0x4001c4]      sw          r15,-32552(gp)
[main:7, 0x4001c8]      lw          r24,-32548(gp)
[main:7, 0x4001cc]      nop
[main:7, 0x4001d0]      lbu         r25,0(r24)          7
[main:7, 0x4001d4]      addiu       r8,r24,1
[main:7, 0x4001d8]      sb          r25,31(sp)
[main:7, 0x4001dc]      b           0x4001f4
[main:7, 0x4001e0]      sw          r8,-32548(gp)
[main:7, 0x4001e4]      jal         _filbuf
[main:7, 0x4001e8]      addiu       r4,gp,-32552
[main:7, 0x4001ec]      move        r16,r2
[main:7, 0x4001f0]      sb          r16,31(sp)
[main:7, 0x4001f4]      lbu         r9,31(sp)
[main:7, 0x4001f8]      li          r1,-1
[main:7, 0x4001fc]      beq         r9,r1,0x400260
[main:7, 0x400200]      nop
[main:17, 0x400204]     lbu         r5,31(sp)           17
[main:17, 0x400208]     jal         printf
[main:17, 0x40020c]     addiu       r4,gp,-32716
[main:18, 0x400210]     lw          r10,-32552(gp)
[main:18, 0x400214]     nop
[main:18, 0x400218]     addiu       r11,r10,-1
[main:18, 0x40021c]     bltz        r11,0x400240
[main:18, 0x400220]     sw          r11,-32552(gp)
[main:18, 0x400224]     lw          r12,-32548(gp)
[main:18, 0x400228]     nop
[main:18, 0x40022c]     lbu         r13,0(r12)          18
[main:18, 0x400230]     addiu       r14,r12,1
[main:18, 0x400234]     sb          r13,31(sp)
[main:18, 0x400238]     b           0x400250
[main:18, 0x40023c]     sw          r14,-32548(gp)
[main:18, 0x400240]     jal         _filbuf
[main:18, 0x400244]     addiu       r4,gp,-32552
[main:18, 0x400248]     move        r16,r2
[main:18, 0x40024c]     sb          r16,31(sp)
[main:18, 0x400250]     lbu         r15,31(sp)
[main:18, 0x400254]     li          r1,-1
[main:18, 0x400258]     bne         r15,r1,0x400204
[main:18, 0x40025c]     nop
[main:19, 0x400260]     b           0x400268
[main:19, 0x400264]     nop
[main:19, 0x400268]     lw          r31,20(sp)          19
[main:19, 0x40026c]     lw          r16,16(sp)
[main:19, 0x400270]     jr          r31
[main:19, 0x400274]     addiu       sp,sp,32
```

Figure 10.8: Source Listing for Line Number Example

Procedure Descriptor Table

Table 10.3 shows the format of an entry in the Procedure Descriptor table; the *sym.h* header file contains its declaration.

Table 10.1: Format of a Procedure Descriptor Table Entry

Declaration	Name	Description
unsigned,long	adr	Memory address of start of procedure.
long	isym	Start of local symbols.
long	iline	Procedure's line numbers. *
long	regmask	Saved register mask.
long	regoffset	Saved register offset.
long	iopt	Procedure's optimization symbol entries.
long	fregmask	Save floating point register mask.
long	fregoffset	Save floating point register offset.
long	frameoffset	Frame size.
long	framereg	Frame pointer register.
long	pcreg	Index or reg of return program counter.
long	lnLow	Lowest line in the procedure.
long	lnHigh	Highest line in the procedure.
long	cbLineOffset	Byte offset for this procedure from the base of the file descriptor entry.

*If NULL, and *cycm* field in file descriptor table = 0, then this field is indexed to the actual table.

Local Symbols

Table 10.4 shows the format of an entry in the Local Symbols table; the *sym.h* header file contains its declaration.

Table 10.2: Format of a Local Symbols Entry.

Declaration	Name	Description
long	iss	Index into local strings of symbol name.
long	value	Value of symbol. See Table 10.5.
unsigned	st : 6	Symbol type. See Table 10.6.
unsigned	sc : 5	Storage class. See Table 10.7.
unsigned	reserved : 1	
unsigned	index : 20	Index into local or auxiliary symbols See Table 3,5.

The meanings of the fields in a local symbol entry are explained in the following paragraphs.

iss. The *iss* (for index into string space) is an offset from the *issBase* field of an entry in the file descriptor table, to the name of the symbol.

value. An integer representing an address, size, offset from a frame pointer. The value is determined by the symbol type, as illustrated in Table 10.5.

st and *sc*. The symbol type (*st*) defines the symbol; the storage class (*sc*), where applicable explains how to access the symbol type in memory. The valid *st* and *sc* constants are given in Tables 10.6 and 10.7. These constants are defined in *symconst.h*.

index. The index is an offset into either Local Symbols or Auxiliary Symbols, depending of the storage type (st) as shown in Table10.5. The compiler uses *isymBase* in the file descriptor entry as the base for a Local Symbol entry and *iauxBase* for an Auxiliary Symbols entry.

Table 10.3: Index and Value as a Function of Symbol Type and Storage Class

Symbol Type	Storage Class	Index	Value
stFile	scText	isymMac	address
stLabel	scText	indexNil	address
stGlobal	scD/B[1]	iaux	address
stStatic	scD/B[1]	iaux	address
stParam	scAbs	iaux	frame offset[2]
	scRegister	iaux	register number
	scVar	iaux	frame offset[2]
	scVarRegister	iaux	register number
stLocal	scAbs	iaux	frame offset[2]
	scRegister	iaux	register number
stProc	scText	iaux	address
	scNil	iaux	address
	scUndefined	iaux	address
stStaticProc	scText	iaux	address
stMember			
enumeration	scInfo	indexNil	ordinal
structure	scInfo	iaux	bit offset[3]
union	scInfo	iaux	bit offset

[1] scD/B is the storage class determined by the assembler, either large/small or data/bss.

[2] frame offset is the offset from the virtual frame pointer.

[3] bit offset is computed from the beginning of the procedure.

Table 10.4: (continued)

Symbol Type	Storage Class	Index	Value
stBlock			
enumeration	scInfo	isymMac[1]	max enumeration
structure	scInfo	isymMac	size
text bock	scText	isymMac	relative address[2]
common block	scCommon	isymMac	size
variant	scVariant	isymMac	isymTag[3]
variant arm	scInfo	isymMac	iauxRanges[4]
union	scInfo	isymMac	size
stEnd			
enumeration	scInfo	isymStart[5]	0
file	scText	isymStart	relative address[2]
procedure	scText	isymStart	relative address[2]
structure	scInfo	isymStart	0
text block	scText	isymStart	relative address[2]
union	scInfo	isymStart	0
common block	scCommon	isymStart	0
variant	scVariant	isymStart	0
variant arm	scInfo	isymStart	0
stTypedef	scInfo	iaux	0

[1]isymMac is the isym of the corresponding stEnd symbol plus 1.

[2]relative address is the relative displacement from the beginning of the procedure.

[3]isymTab is the isym to the symbol that is the tag for the variant.

[4]iauxRanges is the iaux to ranges for the variant arm.

[5]isymStart is the isym of the correspodning begin block (stBlock, stFile, stProc, etc.)

The link editor ignores all symbols except the types *stProc*, *stStatic*, *stLabel*, *stStaticProc*, which it will relocate. Other symbols are used only by the debugger, and need be entered in the table only when the compiler debugger option is ON.

Symbol Type *(st).* Table 10.6 gives the allowable constants that can be specified in the *st* field of Local Symbols entries; the *symconst.h* header file contains the declaration for the constants.

Table 10.5: Symbol Type (st) Constants Supported by the Compiler

Constant	Value	Description
stNil	0	Dummy entry.
stGlobal	1	External symbol.
stStatic	2	Static.
stParam	3	Procedure argument.
stLocal	4	Local variable.
stLabel	5	Label.
stProc	6	Procedure.
stBlock	7	Start of block.
stEnd	8	End block, file, or procedures.
stMember	9	Member of structure, union, or enumeration.
stTypedef	10	Type definition.
stFile	11	File name.
stStaticProc	14	Load time only static procs.
stConstant	15	Const.

Storage Class (*st*) Constants. Table 10.7 gives the allowable constants that can be specified in the *sc* field of Local Symbols entries; the *symconst.h* header file contains the declaration for the constants.

Table 10.6: Table 10.7 Storage Class Constants Supported by the Compiler

Constant	Value	Description
scNil	0	Dummy entry.
scText	1	Text symbol.
scData	2	Initialized data symbol.
scBss	3	Un–initialized data symbol.
scRegister	4	Value of symbol is register number.
scAbs	5	Symbol value is absolute; not to be relocated.
scUndefined	6	Used but undefined in the current module.
reserved	7	
scBits	8	This is a bit field.
scDbx	9	Dbx internal use.
scRegImage	10	Register value saved on stack.
scInfo	11	Symbol contains debugger information.
scUserStruct	12	Address in struct user for current process.
scSData	13	(Load time only) small data.
scSBss	14	(Load time only) small common.
scRData	15	(Load time only) read only data.
scVar	16	Var parameter (Fortran or Pascal).
scCommon	17	Common variable.
scSCommon	18	Small common.
scVarRegister	19	Var parameter in a register.
scVariant	20	Variant records.
scUndefined	21	Small undefined.
scInit	22	Init section symbol.

Optimization Symbols

Reserved for future use.

Auxiliary Symbols

Table 10.8 shows the format of an entry, which is a union, in Auxiliary Symbols; the sym.h file contains its declaration.

Table 10.7: Storage Class Constants Supported by the Compiler

Declaration	Name	Description
TIR	ti	Type information record.
RNDXR	rndx	Relative index into local symbols.
long	dnLow	Low dimension.
long	dnHigh	High dimension.
long	isym	Index into local symbols for stEnd.
long	iss	Index into local strings (not used).
long	width	Width of a structure field not declared with the default value for size.
long	count	Count of ranges for variant arm.

All of the fields except the ti field are explained in the order they appear in the above layout. The ti field is explained last.

rndx. Relative File Index. The front-end fills this field in describing structures, enumerations, and other complex types. The relative file index is a pair of indexes. One index is an offset from the start of the File Descriptor table to one of its entries. The second is an offset from the file descriptor entry to an entry in the Local Symbols or Auxiliary Symbols table.

dnLow. Low Dimension of Array.

dnHigh. High Dimension of Array.

isym. Index into Local Symbols. This index is always an offset to an *stEnd* entry denoting the end of a procedure.

width. Width of Structured Fields.

count. Range Count. Used in describing case variants. Gives how many elements are separated by commas in a case variant.

ti. Type Information Record. Table 10.9 shows the format of a *ti* entry; the *sym.h* file contains its declaration.

Table 10.8: Format of an Type Information Record Entry

Declaration	Name	Description
unsigned	fBitfield : 1	Setif bit width is specified
unsigned	continued : 1	Next auxiliary entry has tq info
unsigned	bt : 6	Basic type
unsigned	tq4 : 4	Type qualifier.
unsigned	tq5 : 4	
unsigned	tq0 : 4	
unsigned	tq1 : 4	
unsigned	tq2 : 4	
unsigned	tq3 : 4	

All groups of auxiliary entries have a type information record with the following entries:

- *fbitfield* - Set if the basic type (bt) is of non–standard width.
- *bt* (for basic type) specifies if the symbol is integer, real complex, numbers , a structure, etc. The valid entries for this field are shown in Table 10.10; the *sym.h* file contains its declaration.
- *tq* (for type qualifier) defines whether the basic type (*bt*) has an array of, function returning, or pointer to qualifier. The valid entries for this field are shown in Table10.11; the sym.h file contains its declaration.

Table 10.9: Basic Type (bt) Constants

Constant	Value	Default Size*	Description
btNil	0	0	Undefined, void.
btAdr	1	32	Address – same size as pointer.
btChar	2	8	Symbol character.
btUChar	3	8	Unsigned character.
btShort	4	16	Short (16 bits).
btUShort	5	16	Unsigned short.
btInt	6	32	Integer.
btUInt	7	32	Unsigned integer.
btLong	8	32	Long (32 bits).
btULong	9	32	Unsigned long.
btFloat		10	32 floating point (real).
btDouble	11	64	Double–precision floating point real.
btStruct	12	n/a	Structure (Record).
btUnion		13	N/A union (variant).
btEnum	14	32	Enumerated.
btTypedef	15	n/a	Defined via a typedef; rndx points at a stTypedef symbol.
btRange	16	32	Subrange of integer.
btSet	17	32	Pascal sets.
btComplex	18	64	FORTRAN complex.
btDComplex	19	128	FORTRAN double complex.
btIndirect	20		Indirect definition;rndx points to TIR aux.
btMax	64		

*Size in bits.

Table 10.10: Type Qualifier (tq) Constants

Constant	Value	Description
tqNil	0	Place holder. No qualifier.
tqPtr	1	Pointer to.
tqProc	2	Function returning.
tqArray	3	Array of.
tqVol	5	Volatile.
tqMax	8	

File Descriptor Table

Table 10.12 shows the format of an entry in the File Descriptor table; the *sym.h* file contains its declaration.

Table 10.11: Format of File Descriptor Entry

Declaration	Name	Description
unsigned,long	adr	Memory address of start of file.
long	rss	Source file name.
long	issBase	Start of local strings.
long	cbSs	Number of bytes in local strings.
long	isymBase	Start of local symbol entries.
long	csym	Count of local symbol entries.
long	ilineBase	Start of line number entries.
long	cline	Count of line number entries.
long	ioptBase	Start of optimization symbol entries.
long	copt	Count of optimization symbol entries.
short	ipdFirst	Start of procedure descriptor table.
short	cpd	Count of procedures descriptors.
long	iauxBase	Start of auxiliary symbol entries.
long	caux	Count of auxiliary symbol entries.
long	rfdBase	Index into relative file descriptors.
long	crfd	Relative file descriptor count.
unsigned	lang : 5	Language for this file.
unsigned	fMerge : 1	Whether this file can be merged.
unsigned	fReadin : 1	True if it was read in (not just created).
unsigned	fBigendian : 1	If set, was compiled on big endian machine aux's is in compile host's sex.
unsigned	reserved : 22	Reserved for future use.
long	cbLineOffset	Byte offset from header or file ln's.
long	cbLine	

External Symbols

The External Symbols table has the same format as Local Symbols, except an offset (*ifd*) field into the File Descriptor table has been added. This field is used to locate information associated with the symbol in an Auxiliary Symbols table. Table 10.13 shows the format of an entry in External Symbols; the *sym.h* file contains its declaration.

Table 10.12: Format an Entry in External Symbols

Declaration	Name	Description
short	reserved	Reserved for future use.
short	ifd	Pointer to file descriptor entry.
SYMR	asym	Same as Local Symbols.

Execution and Linking Format

11

This chapter describes the Execution and Linking Format (ELF) for object files. The following topics are covered:

- The Components of an elf object file.
- Symbol Table Format.
- Global Data Area.
- Register Information.
- Relocation.

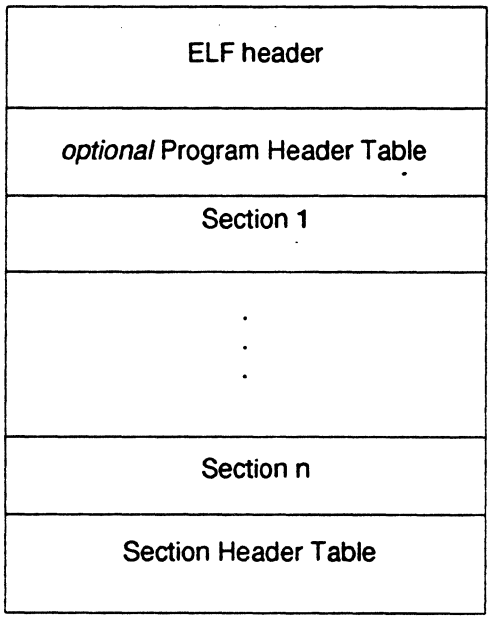
Program loading and dynamic linking are discussed in Chapter 12.

There are three types of object files:

- Relocatable files contain code and data and are linked with other object files to create an executable file or shared object file.
- Executable files contain a program that can be executed.
- Shared object files contain code and data that can be linked. These files may be linked with relocatable or shared object files to create other object files. They may also be linked with an executable file and other shared objects to create a process image.

Object File Format

An object file is organized as follows:



Each object file begins with an ELF header that describes the file. Sections contain information that is used when the file is linked with other objects (e.g. code, data, relocation information). The Section Header Table contains information describing the sections of the file and has an entry for each section. Files that will be linked with other objects must contain a Section Header Table.

If the Program Header Table is present, it contains information that is used to create a process image. Files used to build an executable program must have a Program Header Table; relocatable files do not need one.

ELF Header

The ELF header has the following format:

Declaration	Field
unsigned char	e_ident[EI_NIDENT];
Elf32_Half	e_type;
Elf32_Half	e_machine;
Elf32_Word	e_version;
Elf32_Addr	e_entry;
Elf32_Off	e_phoff;
Elf32_Off	e_shoff;
Elf32_Word	e_flags;
Elf32_Half	e_ehsize;
Elf32_Half	e_phentsize;
Elf32_Half	e_phnum;
Elf32_Half	e_shentsize;
Elf32_Half	e_shnum;
Elf32_Half	e_shstrndx;
#define EI_NIDENT	16

e_ident
contains machine-independent data concerning the file contents.

The index values for the *e_ident* member are:

EI_MAG0	0	File identification.
EI_MAG1	1	File identification.
EI_MAG2	2	File identification.
EI_MAG3	3	File identification.
EI_CLASS	4	File class.
EI_DATA	5	Byte order.
EI_VERSION	6	File version.
EI_PAD	7	Start of padding bytes.
EI_NIDENT	16	Size of e_ident[].

e_ident[EI_MAGO ... EI_MAG3]
contain a magic number identifying the file as an ELF object file

Position	Value	Name
e_ident[EI_MAG0]	0x7f	ELFMAG0
e_ident[EI_MAG1]	'E'	ELFMAG1
e_ident[EI_MAG2]	'L'	ELFMAG2
e_ident[EI_MAG3]	'F'	ELFMAG3

e_ident[EI_CLASS]
indicates the file class or capacity and must have the value ELFCLASS32.

e_ident[EI_DATA]
indicates the byte ordering of processor specific data in the object file and must be either ELFDATA2LSB (little endian byte order) or ELFDATA2MSB (big endian byte order).

e_ident[EI_VERSION]
indicates the version number of the ELF header and must be EV_CURRENT.

e_ident[PAD]
marks the beginning of unused bytes in the ELF header. These bytes are reserved and set to zero.

e_type
identifies the type of the object file and can have the following values:

ET_NONE	0	No file type.
ET_REL	1	Relocatable.
ET_EXEC	2	Executable.
ET_DYN	3	Shared object.
ET_CORE	4	Core file.
ET_LOPROC	0xff00	Processor specific.
ET_HIPROC	0xffff	Processor specific.

e_machine
indicates the required architecture and must have the value EM_MIPS.

e_version
indicates the object file version and must have the value EV_CURRENT. The value of EV_CURRENT is 1; in the future, this value will increase as extensions are added to the ELF header.

e_entry
contains the virtual address to which the system transfers control when the process is started. If the file has no entry point, this value is zero.

e_phoff
contains the offset in bytes of the Program Header Table and may be zero if the table is not present.

e_shoff
contains the offset in bytes of the Section Header Table. If the file has no Section Header Table, its value is zero.

e_flags
contains bit flags associated with the file. The following flags are defined:

```
EF_MIPS_NOREORDER       0x00000001
EF_MIPS_PIC             0x00000002
EF_MIPS_CPIC            0x00000004
EF_MIPS_ARCH            0xf0000000
```

This bit is asserted when at least one .noreorder directive in an assembly source program contributes to the object module.

If EF_MIPS_PIC is set, the file contains position–independent code that is relocatable.

If EF_MIPS_CPIC is set, the file contains code that conforms to the standard calling sequence rules for calling position–independent code. The code in this file is not necessarily position–independent.

The bits indicated by EF_MIPS_ARCH identify extensions to the MIPS1 architecture. AN ABI compliant file must have zero in these four bits.

e_ehsize
contains the size in bytes of the ELF header.

e_phentsize
contains the size in bytes of an entry in the file's Program Header Table.

e_phnum
indicates the number of entries in the Program Header Table. If a file has no Program Header Table, this value is zero. The product of *e_phnum* and *e_phentsize* gives the size in bytes of the Program Header Table.

e_shentsize
contains the size in bytes of an entry in the Section Header Table (also referred to as a Section Header).

e_shnum
indicates the number of entries in the Section Header Table. If a file has no Section Header Table, this value is zero. The product of *e_shnum* and *e_shentsize* gives the size in bytes of the Section Header Table.

e_shstrndx
contains the Section Header Table index of the entry associated with the Section Name String Table. If the table does not exist, this value is SHN_UNDEF.

Sections

Each section has a section header (an entry in the Section Header Table). There may be entries in the Section Header Table that have no associated section. Each section occupies a contiguous, possibly empty, sequence of bytes and may not overlap any other section.

Section Header Table

The Section Header Table is an array of structures that describe the sections of the object file. A Section Header Table Index is a subscript into this array of structures. Some of these indexes are reserved. An object file can not have a section that corresponds to a reserved index.

The following Special Section Indexes are defined:

Name	Value
SHN_UNDEF	0
SHN_LORESERVE	0xff00
SHN_LOPROC	0xff00
SHN_HIPROC	0xff1f
SHN_ABS	0xfff1
SHN_COMMON	0xfff2
SHN_HIRESERVE	0xffff
SHN_MIPS_ACCOMON	0xff00

The special section indexes have the following meanings:

SHN_UNDEF
marks an undefined, missing, or meaningless section reference. A symbol defined relative to section number SHN_UNDEF is an undefined symbol.

SHN_LORESERVE
specifies the lower bound of the reserved indexes.

SHN_LOPROC through SHN_HIPROC
are reserved for processor specific semantics.

SHN_ABS
specifies absolute values for the corresponding references. Symbols defined relative to this section number have absolute values and are not affected by relocation.

SHN_COMMON
indicates that the corresponding references are common symbols, such as FORTRAN COMMON or unallocated C external variables.

SHN_HIRESERVE
specifies the upper bound of the reserved indexes. The Section Header Table does not contain entries for the reserved indexes.

SHN_MIPS_ACOMMON
indicates that the corresponding references are common symbols. The *st_value* member for a common symbol contains its virtual address. If the section is relocated, the alignment indicated by the virtual address is preserved, up to modulo 65536.

Section Header

A section header (an entry in the Section Header Table) has the following structure:

Declaration	Field
Elf32_Word	sh_name;
Elf32_Word	sh_type;
Elf32_Word	sh_flags;
Elf32_Addr	sh_addr;
Elf32_Off	sh_offset;
Elf32_Word	sh_size;
Elf32_Word	sh_link;
Elf32_Word	sh_info;
Elf32_Word	sh_addralign;
Elf32_Word	sh_entsize;

sh_name
specifies the name of the section. Its value is an index into the section header string table section and gives the location of a null terminated string that is the name of the section.

sh_type
indicates the type of the section and may have the following values :

Name	Value
SHT_NULL	0
SHT_PROGBITS	1
SHT_SYMTAB	2
SHT_STRTAB	3
SHT_RELA	4
SHT_HASH	5
SHT_DYNAMIC	6
SHT_NOTE	7
SHT_NOBITS	8
SHT_REL	9
SHT_SHLIB	10
SHT_DYNSYM	11
SHT_LOPROC	0x70000000
SHT_HIPROC	0x7fffffff
SHT_LOUSER	0x80000000
SHT_HIUSER	0xffffffff
SHT_MIPS_LIBLIST	0x70000000
SHT_MIPS_CONFLICT	0x70000002
SHT_MIPS_GPTAB	0x70000003
SHT_MIPS_UCODE	0x70000004

SHT_NULL
marks the section header as inactive. There is no associated section. Other members of the section header have undefined values.

SHT_PROGBITS
indicates that the section contains information defined by the program. The format and meaning of the information are determined by the program.

SHT_SYMTAB and SHT_DYNSYM
sections contain a symbol table. An object file may have only one section of each type. SHT_SYMTAB contains symbols used in link editing, but may also be used for dynamic linking. It may contain many symbols unnecessary for dynamic linking. Consequently, an object may also contain a SHT_DYNSYM section that contains a minimal set of dynamic linking symbols.

SHT_STRTAB
indicates that the section holds a string table. An object file may have multiple string table sections.

SHT_RELA
indicates that the section contains relocation entries with explicit addends, such as type Elf32_Rela for the 32–bit class of object files. An object file may have multiple relocation sections.

SHT_HASH
marks a section that holds a symbol hash table. An object file may have only one hash table.

SHT_DYNAMIC
indicates that the section contains information used in dynamic linking. An object file may have only one dynamic section.

SHT_NOTE
indicates that the section holds information that marks the file in some way.

SHT_NOBITS
indicates that the section occupies no space but otherwise resembles a section of type SHT_PROGBITS. Although this section occupies no space in a file, its *sh_offset* field contains the conceptual file offset.

SHT_REL
indicates that the section contains relocation entries without explicit addends. An object file may have multiple relocation sections.

SHT_SHLIB
is a reserved type that has no semantics. A program that contains a section of this type does not conform to the ABI.

SHT_LOPROC through SHT_HIPROC
are reserved for processor–specific semantics.

SHT_LOUSER
indicates the lower bound of the range of indexes reserved for application programs.

SHT_HIUSER
indicates the upper bound of the range of indexes reserved for application programs. Sections types between SHT_LOUSER and SHT_HIUSER may be used by applications without conflicting with current or future section types reserved for system use.

SHT_MIPS_LIBLIST
: indicates that the section contains information about the set of dynamic shared object libraries, such as library name and version, used when statically linking a program. See the Quickstart section in Chapter 12 of this manual for details.

SHT_MIPS_CONFLICT
: marks a section that contains a list of symbols in an executable object whose definitions conflict with symbols defined in shared objects.

SHT_MIPS_GPTAB
: indicates that the section contains the *global pointer table*. The global pointer table contains a list of possible global data area sizes which allows the linker to provide the user with information on the optimal size criteria to use for gp register relative addressing. See the Global Data Area section of this chapter.

SHT_MIPS_UCODE
: indicates that the section contains MIPS ucode instructions.

Other section type values are reserved. The section header for index 0 (SHN_UNDEF) marks undefined section references. This entry has the following values:

Name	Value	Note
sh_name	0	No name
sh_type	SHT_NULL	Inactive
sh_flags	0	No flags
sh_addr	0	No address
sh_offset	0	No file offset
sh_size	0	No size
sh_link	SHN_UNDEF	No link information
sh_info	0	No auxiliary information
sh_addralign	0	No alignment
sh_entsize	0	No entries

sh_flag
contains bit flags describing attributes of the file. The following flags are defined:

SHF_WRITE	0x1
SHF_ALLOC	0x2
SHF_EXECINSTR	0x4
SHF_MASKPROC	0xf0000000
SHF_MIPS_GPREL	0x10000000

SHF_WRITE
If this bit is set, the section contains data that must be writable during process execution.

SHF_ALLOC
This bit indicates that the section occupies memory during process execution.

SHF_EXECINSTR
If this bit is set, the section contains executable machine instructions.

SHF_MASK_PROC
All the bits included in this mask are reserved for processor-specific semantics.

SHF_MIPS_GPREL
: This bit indicates that the section contains data that must be made part of the global data area during program execution. Data in this section is addressable with a gp relative address. The *sh_link* field of a section with this attribute must be a Section Header Index of a section of type SHT_MIPS_GPTAB.

sh_addr
: If the section appears in the memory image of a process, this member contains the address of the first byte of the section. Otherwise, its value is zero.

sh_offset
: Contains the byte offset from the beginning of the file of this section.

sh_size
: Contains the size of the section in bytes.

sh_link
: Contains a Section Header Table index link. The interpretation of this value depends on the section type (see Table 11.1).

sh_info
: Contains miscellaneous information. The interpretation of the value depends on the section type (see table 11.1).

Table 11.1: sh_link and sh_info values

sh_type	sh_link	sh_info
SHT_DYNAMIC	The section header index of the string table used by entries in the section.	0
SHT_HASH	The section header index of the symbol table to which the hash table applies.	0
SHT_REL SHT_DYNSYM	The section header index of the associated symbol table.	The section header index of the section to which the relocation applies.
SHT_SYMTAB SHT_DYNSYM	The section header index of the associated string table.	One greater than the symbol table index of the last local symbol (bind STB_LOCAL).
SHT_MIPS_LIBLIST	The section header index of the string table used by entries in this section.	The number of entries in this section.
SHT_MIPS_GPTAB	not used	The section header index of the SHF_ALLOC + SHF_WRITE section.
other	SHN_UNDEF	0

sh_addralign
Indicates address alignment constraints for the section. For example, if a section contains a doubleword value, the entire section must be aligned on a doubleword boundary. The value of this member may be 0 or a positive integral power of 2; 0 or 1 indicates that the section has no alignment constraints.

sh_entsize
If the section holds a table of fixed-size entries, such as a symbol table, this member gives the size in bytes of each entry. A value of zero indicates that the section does not contain a table of fixed-size entries.

Special Sections

An object file has the following special sections:

Table 11.2: Special Sections

Name	Type	Attributes
.bss	SHT_NOBITS	SHF_ALLOC+SHF_WRITE
.comment	SHT_PROGBITS	none
.data	SHT_PROGBITS	SHF_ALLOC+SHF_WRITE
.data1	SHT_PROGBITS	SHF_ALLOC+SHF_WRITE
.debug	SHT_PROGBITS	none
.dynamic	SHT_DYNAMIC	SHF_ALLOC
.dynstr	SHT_STRTAB	SHF_ALLOC
.dynsym	SHT_DYNSYM	SHF_ALLOC
.fini	SHT_PROGBITS	SHF_ALLOC+ SHF_EXECINSTR
.got	SHT_PROGBITS	see below
.hash	SHT_HASH	SHF_ALLOC
.init	SHT_PROGBITS	SHF_ALLOC+ SHF_EXECINSTR
.interp	SHT_PROGBITS	see below
.line	SHT_PROGBITS	none
.note	SHT_PROGBITS	none
.plt	SHT_PROGBITS	see below
.rel*name*	SHT_REL	see below
.rela*name*	SHT_RELA	see below
.rodata	SHT_PROGBITS	SHF_ALLOC
.rodata1	SHT_PROGBITS	SHF_ALLOC
.shstrtab	SHT_STRTAB	none
.strtab	SHT_STRTAB	see below
.symtab	SHT_SYMTAB	see below
.text	SHT_PROGBITS	SHF_ALLOC+ SHF_EXECINSTR
.sdata	SHT_PROGBITS	SHF_ALLOC+SHF_WRITE+ SHF_MIPS_GPREL
.sbss	SHT_NOBITS	SHF_ALLOC+SHF_WRITE+ SHF_MIPS_GPREL
.lit4	SHT_PROGBITS	SHF_ALLOC+SHF_MIPS_GPREL
.lit8	SHT_PROGBITS	SHF_ALLOC+ SHF_MIPS_GPREL
.reginfo	SHT_MIPS_REGINFO	SHF_ALLOC
.liblist	SHT_MIPS_LIBLIST	SHF_ALLOC
.conflict	SHT_CONFLICT	SHF_ALLOC
.gptab	SHT_MIPS_GPTAB	none
.got	SHT_PROGBITS	SHF_ALLOC+ SHF_WRITE+ SHF_MIPS_GPREL
.ucode	SHT_MIPS_UCODE	none
.mdebug	SHT_MIPS_DEBUG	none

Note: A MIPS ABI compliant system must support the *.sdata*, *.sbss*, *.lit4*, *.lit8*, *.reginfo*, and *.gptab* sections. A MIPS ABI compliant system must recognize, but may choose to ignore, the *.liblist*, *.msym*, and *.conflict* sections. However, if any one of these sections is supported, all three must be supported. A MIPS ABI compliant system is not required to support the *.ucode* section, but if this section is present, it must conform to the description in this manual.

.bss
This section holds uninitialized data. The system initializes the data to zeros when the program is started. This section occupies no file space.

.comment
This section holds version information.

.data
This section contains initialized data.

.debug
This section hold information used for symbolic debugging.

.dynamic
This section contains information used for dynamic linking. See Chapter 12 for more details on dynamic linking.

.dynstr
This section contains strings needed for dynamic linking, usually strings representing the names associated with symbol table entries.

.fini
This section uholds executable instructions that are executed when the program terminates normally.

.got
This section hold the Global Offset Table.

.hash
This section contains a hash table for symbols. See the Symbol Table section of this chapter for a description of the symbol table.

.init
This section holds executable instructions that are executed before the system calls the main entry point for the program.

.interp
This section holds the path name of a program interpreter. If the file has a loadable segment that includes the section, the section attributes include SHF_ALLOC.

.line
This section contains line number information describing the correspondence between source code lines and machine code. This information is used for symbolic debugging.

.note
This section contains information that marks the file in some way. See the Note Section section in Chapter 12.

.plt
This section holds the Procedure Linkage Table.

.relname
This section contains relocation information. *name* is supplied by the section to which the relocation applies. If the file has a loadable segment that includes the section, the section attributes include SHF_ALLOC.

.relaname
This section contains relocation information. *name* is supplied by the section to which the relocation applies. If the file has a loadable segment that includes the section, the section attributes include SHF_ALLOC.

.rodata
This section holds read–only data that is generally found in the non–writable segmeunt of the process image. See Chapter 12 for more information on segments.

.rodata1
This section holds read–only data that is generally found in the non–writable segment of the process image. See Chapter 12 for more information on segments.

.shstrtab
This section contains strings representing section names.

.strtab
This section contains strings, including the strings representing the names associated with symbol table entries. If the file has a loadable segment that includes the symbol string table, the section attributes include SHF_ALLOC.

.symtab
This section holds a symbol table. See the Symbol Table section of this chapter for a description. If the file has a loadable segment that includes the symbol table, the section attributes include SHF_ALLOC.

.text
This section contains executable instructions.

.sdata
This section holds initialized short data.

.sbss
This section holds uninitialized short data. The system sets the data to zeros when thue program is started. Unlike the *.bss* section, this section occupies file space.

.lit4
This section holds 4 byte read–only literals. The section is part of a non–writable segment in the process image.

.lit8
This section holds 8 byte read–only literals. The section is part of a non–writable segment in the process image.

.reginfo
This section contains information on the program's register usage.

.liblist
This section contains information on the libraries used at static link time.

.conflict
This section provides additional dynamic linking information for symbols in an executable file that conflict with symbols defined in the dynamic shared libraries.

.gptab
This section contains a Global Pointer Table. The section is named *.gptab.sbss*, *.gptab.sdata*, *.gptab.bss*, or *.gptab.data* depending on the data section to which the section refers.

.ucode
This section holds U–code instructions generated by the compiler.

.mdebug
This section contains MIPS specific symbol table information. The contents of this section are described in Chapter 10. The information in this section is dependent on the location of other sections in the file. If an object is relocated, this section must be updated. This section must be discarded if an object file is relocated and the ABI compliant system chooses not to update the section.

.got
This section contains the Global Offset Table. The *sh_info* field holds the Global Pointer value used for this Global Offset Table.

.dynamic

This section is a MIPS–specific dynamic section. It is the same as the previously mentiouned *.dynamic* section except that its attributes do not include SHF_WRITE.

String Tables

String table sections contain null–terminated character sequences (strings) that represent symbol and section names. A string is referenced by an index into the String Table Section.

The first byte of a string section, accessed by index zero, contains a null character. The last byte also contains a null character, ensuring that all strings are null terminated. A string whose index is zero specifies either no name or a null name, depending on the context.

A String Table Section may be empty. In this case, the *sh_size* field for the section would contain zero. Non–zero indexes are invalid for an empty string table.

The following figure shows an example of a string table:

Index	0	1	2	3	4	5	6	7	8	9
	\0	a	b	c	d	\0	v	a	r	n
10	a	m	e	\0	f	o	o	\0	b	a
20	r	\0								

A string table index may refer to any byte in the section; references to substrings are permitted. A single string may be referenced multiple times and unreferenced strings may exist.

ELF Symbol Table

The ELF symbol table is found in the *.symtab* section of an object file. It is an array of structures containing information needed to locate and relocate the symbol definitions and references of a program.

A symbol table index is a subscript into this array. Index zero is the first entry in the table and is also the undefined symbol index. A symbol table entry has the following format:

Declaration	Name
Elf32_Word	st_name;
Elf32_Addr	st_value;
Elf32_Word	st_size;
unsigned char	st_info;
unsigned char	st_other;
Elf32_Half	st_shndx;

st_name
Holds an index into the symbol string table. If its value is non-zero, it indicates a string that is the symbol name. Otherwise, the symbol table entry has no associated name.

st_value
Contains the value of the associated symbol.

st_size
Contains the size (the number of bytes comprising the data object) of the associated symbol. This value is zero if the symbol has no size or the size is unknown.

st_info
Specifies the type of the symbol and its binding attributes. The following code fragment shows how to manipulate the binding and type:

```
#define     ELF32_ST_BIND(i)          ((i)>>4)
#define     ELF32_ST_TYPE(i)          ((i)&0xf)
#define     ELF32_ST_INFO(b,t)    ((b)<<4+((t)&0xf))
```

A symbol's binding determines the linkage visibility. The value of *st_info* may be one of the following:

STB_LOCAL	0
STB_GLOBAL	1
STB_WEAK	2
STB_LOPROC	13
STB_HIPROC	15

STB_LOCAL indicates local symbols. These symbols are not visible outside of the object file containing the definition. Local symbols with the same name may exist in multiple object files without causing conflicts.

STB_GLOBAL indicates global symbols. Global symbols are visible to all the object files being combined. A global symbol defined in one file satisfies a reference to an undefined global symbol in another file.

STB_WEAK indicates weak symbols. Weak symbols are similar to global symbols, but have lower precedence.

STB_LOPROC through STB_HIPROC values are reserved for processor–specific semantics.

In each symbol table, all local symbols precede the global and weak symbols. As indicated in the Section Header section of this chapter, the *sh_info* field of the section header contains the symbol table index for the first non–local symbol. Global and weak symbols differ in two ways:

- When the link editor combines several relocatable object files, it does not allow multiple definitions of STB_GLOBAL symbols with the same name. If a defined global symbol exists, the appearance of a weak symbol with the same name does not cause an error. The link editor ignores the weak symbol and uses the global definition.
- When the link editor searches archive libraries, it extracts members of the archive that contain definitions of undefined global symbols. The definition in the extracted member may be either a global or a weak symbol. The link editor does not extract archive members to resolve undefined weak symbols; unresolved weak symbols have a value of zero.

st_other
Contains zero and is currently unused.

st_shndx
Contains the Section Header Table index for the symbol table entry.

Symbol Type

The following symbol types are defined:

Table 11.3: Symbol Types

Name	Value
STT_NOTYPE	0
STT_OBJECT	1
STT_FUNC	2
STT_SECTION	3
STT_FILE	4
STT_LOPROC	13
STT_HIPROC	15

STT_NOTYPE indicates that the symbol has no type.

STT_OBJECT indicates that the symbol is associated with a data object, such as a variable.

STT_FUNC indicates that the symbol is associated with a function or other executable code.

STT_SECTION indicates that the symbol is associated with a section. Entries of this type are primarily for relocation and normally have STB_LOCAL binding.

STT_FILE indicates that the symbol name is the name of the source file associated with the object file. A file symbol has STB_LOCAL binding, its section index is SHN_ABS, and, if present, it precedes the other STB_LOCAL symbols.

STT_LOPROC through STT_HIPROC are reserved for processor-specific semantics.

Function symbols (type STT_FUNC) have special significance. When another object file references a function that is part of a shared object, the link editor creates a Procedure Linkage Table entry for the referenced symbol. Symbols in shared objects that have types other than STT_FUNC are not referenced automatically through the Procedure Linkage Table.

If the value of a symbol refers to a location within a section, the *st_shndx* field for the symbol contains an index into the Section Header table. When the section is relocated, the symbol's value is changed and references to the symbol continue to point to the same location in the program. Some special section index values have other semantics:

SHN_ABS indicates that the symbol has an absolute value that does not change because of relocation.

SHN_COMMON indicates that the symbol is a label for a common block that has not yet been allocated. The symbol's value gives alignment constraints, similar to a section's *sh_addralign* field. The link editor allocates the storage for the symbol at an address that is a multiple of *st_value*. The symbol's size tells how many bytes are required.

SHN_UNDEF indicates that the symbol is undefined. When the link editor combines the object file with another that contains the definition for the symbol, this file's references to the symbol are linked to the actual definition.

The symbol table entry for index zero (STN_UNDEF) is reserved and holds the following information:

Name	Value	Note
st_name	0	No name
st_value	0	Zero Value
st_size	0	No size
st_info	0	No type, local binding
st_other	0	
st_shndx	SHN_UNDEF	No section

Symbol Values

Symbol table entries for different object file types have slightly different interpretation for the *st_value* field:

- In relocatable files, *st_value* contains the alignment constraints for a symbol whose section index is SHN_COMMON.
- In relocatable files, *st_value* contains a section offset for a defined symbol; *st_value* is an offset from the beginning of the section that *st_shndx* indicates.
- In executable and shared object files, *st_value* contains a virtual address. The section offset gives way to a virtual address for which the section number is irrelevant.

If an executable file contains a reference to a function defined in a shared object, the symbol table section for the file contains an entry for that symbol. The *st_shndx* field of the symbol table entry for the function contains SHN_UNDEF. If there is a stub for the function in the executable file, and the *st_value* field for the symbol table entry is non-zero, the field contains the virtual address of the first instruction of the function's stub.

Otherwise, the *st_value* field contains zero. This stub is used to call the dynamic linker at runtime for lazy text evaluation.

Global Data Area

The global data area is part of the data segment of an executable program. It contains *short* data items which can be addressed by the *gp* register relative addressing mode. The global data area comprises all the sections with the SHF_MIPS_GPREL attribute.

The compilers generate *short-form* (one machine instruction) *gp* relative addressing for all data items in any of these sections with the SHF_MIPS_GPREL attribute. The compilers must generate two machine instructions to load or store data items outside the global data area. A program executes faster if more data items are placed in the global data area.

The size of the global data area is limited by the addressing constraints on *gp* relative addressing, namely plus or minus 32 kilobytes relative to *gp*. This limits the size of the global data area to 64 kilobytes.

The compilers decide whether or not a data item is placed in the global data area based on its size. All data items less than or equal to a given size are placed in the global data area. Initialized data items are placed in a *.sdata* section, uninitialized data items are placed in a *.sbss* section, and floating-point literals are placed in *.lit4* and *.lit8* sections. The *.got* section is also combined into the global data area.

In order to provide the user with information on the optimal size criteria for placement of data items in the *.sdata* and *.sbss* sections, the linker maintains tables of possible global data area sizes for each of these sections. These tables are maintained in *.gptab* sections. Each *.gptab* section contains both the actual value used as the size criteria for an object file and a sorted list of possible short data and bss area sizes based on different data item size selections. The size criteria value is also known as the –G *num*.

The *.gptab* section is an array of structures that have the following format:

```
typedef union {
        struct {
                Elf32_Word gt_current_g_value;
                Elf32_Word gt_unused;
        } gt_header;
        struct {
                Elf32_Word gt_g_value;
                Elf32_Word gt_bytes;
        } gt_entry;
} Elf32_gptab;
```

gt_header.gt_current_g_value
Is the –G *num* used for this object file. Data items of this size or smaller are referenced with gp relative addressing and reside in a SHF_MIPS_GPREL section.

gt_header.gt_unused
Is not used in the first entry of the array.

gt_entry.gt_g_value
Is a hypothetical –G *num* value.

gt_entry.gt_bytes
Is the length of the global data area if the corresponding *gt_entry.gt_g_value* were used.

Each of the *gt_entry. gt_g_value* fields is the size of a data item encountered during compilation or assembly, including zero. Each separate size criteria results in an overall size for the global data area. The various entries are sorted and duplicates are removed. The resulting set of entries, including the –G *num* used, yields the *.gptab* section.

There are always at least two *.gptab* sections, one that corresponds to initialized data and one which corresponds to uninitialized data. The *sh_info* field of the section specifies the section index of the data section to which this *.gptab* section applies. Normally the two *.gptab* sections would apply to the *.sdata* and *.sbss* sections, but if one or both of these sections do not exist, the *.gptab* applies to the *.data* and *.bss* sections.

The section to which the *.gptab* section applies is derived from its name. The four possible name for this type of section are *.gptab.sbss*, *.gptab.sdata*, *.gptab.bss*, and *.gptab.data*.

Register Information

The compilers and assembler collect information on the registers used by the code in the object file. This information is communicated to the operating system kernel in the *.reginfo* section. The operating system kernel could use this information to decide what registers it might not need to save or which coprocessors the program uses. The section also contains a field which specifies the initial value for the *gp* register, based on the final location of the global data area in memory. The register information structure has the following format:

```
typedef struct {
        Elf32_Word ri_gprmask;;
        Elf32_Word ri_cprmask[4];
        Elf32_Word ri_gp_value;
} ELF_RegInfo;
```

ri_gprmask
contains a bit–mask of general registers used by the program. Each set bit indicates a general integer register used by the program. Each clear bit indicates a general integer register not used by the program. For instance, bit 31 set indicates register *$31* is used by the program; bit 27 clear indicates register *$27* is not used by the program.

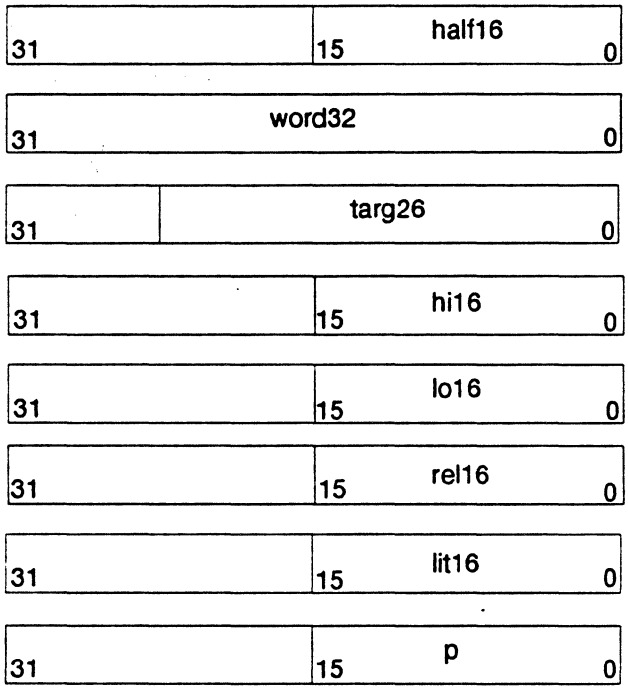
ri_cprmask
contains the bit–mask of co–processor registers used by the program. The MIPS RISC architecture can support up to four co–processors, each with 32 registers. Each array element corresponds to one set of co–processor registers. Each of the bits within the element corresponds to individual registers in the co–processor register set. The 32 bits of the words correspond to the 32 registers, with bit number 31 corresponding to register 31, bit number 30 to register 30, etc. Set bits indicate the corresponding register is used by the program; clear bits indicate the program does not use the corresponding register.

ri_gp_value
contains the gp register value. In relocatable object files it is used for relocation of the R_MIPS_GPREL and R_MIPS_LITERAL relocation types.

Note: Only co–processor 1 may be used by ABI compliant programs. This means that only the *ri_cprmask*[1] array element may have a non–zero value. *ri_cprmask*[0], *ri_cprmask*[2], and *ri_cprmask*[3] must all be zero in an ABI compliant program.

Relocation

Relocation entries describe how to alter instruction and data fields for relocation; bit numbers appear in the lower box corners.



Calculations below assume the actions are transforming a relocatable file into either an executable or a shared object file. Conceptually, the linker merges one or more relocatable files to form the output. It first decides how to combine and locate the input files, then updates the symbol values, and finally performs the relocation.

Relocations applied to executable or shared object files are similar and accomplish the same result. The descriptions in Table 11.4 use the following notation:

A The addend used to compute the value of the relocatable field.

AHL Another type of addend used to compute the value of the relocatable field. See the note below for more detail.

P The location (section offset or address) of the storage unit being relocated (computed using *r_offset*).

S The value of the symbol whose index resides in the relocation entry, unless the symbol is STB_LOCAL and is of type STT_SECTION, in which case it means the original *sh_addr* minus the final *sh_addr*.

G The offset into the global offset table at which the address of the relocation entry's symbol resides during execution.

GP The final *gp* value that is used for the relocatable, executable, or shared object file being produced.

GPO The *gp* value used to create the relocatable object.

EA The effective address of the symbol prior to relocation.

L The *.lit4* or *.lit8* literal table offset. Prior to relocation, the addend field of a literal reference contains the offset into the global data area. During relocation, each literal section from each contributing file is merged into one and sorted, after which duplicate entries are removed and the section compressed, leaving only unique entries.
The relocation factor L is the mapping from the old offset from the original *gp* to the value of *gp* used in the final file.

A relocation entry's *r_offset* value designates the offset or virtual address of the first byte of the affected storage unit. The relocation type specifies which bits to change and how to calculate their values. Because MIPS uses only *Elf32_Rel* relocation entries, the field to be relocated holds the addend.

The AHL addend is a composite computed from the addends of two consecutive relocation entries. Each relocation type of R_MIPS_HI16 must have an associated R_MIPS_LO16 entry immediately following it in the list of relocations. These relocation entries are always processed as a pair and both addend fields contribute to the AHL addend. If AHI and ALO are the addends from the paired R_MIPS_HI16 and R_MIPS_LO16 entries, then the addend AHL is computed as ((AHI << 16) + (short)ALO). R_MIPS_LO16 entries without an immediately preceding R_MIPS_HI16 entry are orphaned and the previously defined R_MIPS_HI16 is used for computing the addend.

Note: Field names in the following table tell whether the relocation type checks for overflow. A calculated relocation value may be larger than the intended field, and a relocation type may verify (*V*) that the value fits or truncate(*T*) the result. As an example, *V–half16* means that the computed value may not have significant non–zero bits outside the *half16* field.

Table 11.4: Relocation Calculations

Name	Value	Field	Symbol	Calculation
R_MIPS_NONE	0	none	local	none
R_MIPS_16	1	V–half16	external	S + sign–extended(A)
	1	V–half16	local	S + sign–extended(A)
R_MIPS_32	2	T–word32	external	S + A
	2	T–word32	local	S + A
R_MIPS_REL32	3	T–word32	external	A – EA + S
	3	T–word32	local	A – EA + S
R_MIPS_26	4	T–arg26	local	(((A<<2) | (P & 0xf0000000) + S)>>2)
	4	T–arg26	external	(sign–extended(A<<2)+ S) >> 2
R_MIPS_HI16	5	T–hi16	external	((AHL + S) – (short)(AHL+ S)) >> 16
	5	T–hi16	local	((AHL + S) – (short) (AHL+ S)) >> 16
R_MIPS_LO16	6	T–lo16	external	AHL + S
	6	T–lo16	local	AHL + S
R_MIPS_GPREL	7	V–rel16	external	sign–extended (A) + S + GP
	7	V–rel16	local	sign–extended(A) + S + GPO – GP
R_MIPS_LITERAL	8	V–lit16	local	signed–extended(A) + L
R_MIPS_GOT16	9	V–rel16	external	G
	9	V–rel16	local	see below
R_MIPS_PC16	10	V–pc16	external	sign–extended(A) + S – P

In the Symbol column in table 11.4, if the symbol referenced by the symbol table index in the relocation entry is STB_LOCAL/STT_SECTION, then it is a local relocation. If it is not, the relocation is considered an external relocation.

The R_MIPS_32 and R_MIPS_REL32 relocation types are the only relocations performed by the dynamic linker.

If an R_MIPS_GOT16 refers to a locally defined symbol, the relocation is done differently than if it refers to an external symbol. In the local case it must be followed immediately by an R_MIPS_LO16 relocation. The AHL addend is extracted and the section in which the referenced data item resides is determined (this requires all sections in an object module to have unique addresses and no overlap). From this address the final address of the data item is calculated. If necessary, a global offset table entry is created to hold the high 16 bits of this address (an existing entry is used when possible). The *rel16* field is replaced by the offset of this entry in the global offset table. The *lo16* field in the following R_MIPS_LO16 relocation is replaced by the low 16 bits of the actual destination address. This is meant for local data references in position–independent code so that only one global offset table entry is necessary for every 64 kilobytes of local data.

The first instance of R_MIPS_GOT16 causes the link editor to build a global offset table if one has not already been built.

Program Loading and Dynamic Linking

12

Executable files and object files are used to create a process image when a program is started by the system. This chapter describes the object file structures that relate to program execution and also describes how the process image is created from these files. Topics in this chapter include:

- Program Header
- Object File Segments
- Program Loading
- Dynamic Linking
- Quickstart

Program Header

The Program Header table is an array of structures, each of which describes a segment or other data used to create a process image. A Program Header is meaningful only for a shared object or executable file.

A description of the Program Header for MIPS COFF format is in Chapter 9. The structure of a Program Header for ELF entry is as follows:

Declaration	Field
Elf32_Word	p_type;
Elf32_Off	p_offset;
Elf32_Addr	p_vaddr;
Elf32_Addr	p_paddr;
Elf32_Word	p_filesz;
Elf32_Word	p_memsz;
Elf32_Word	p_flags;
Elf32_Word	p_align;

The size of the Program Header is specified by the ELF Header *e_phentsize* and *e_phnum* fields (see Chapter 11).

p_type indicates what kind of segment this entry describes or how to interpret the array element's information. *p_type* may have the following values:

Name	Value
PT_NULL	0
PT_LOAD	1
PT_DYNAMIC	2
PT_INTERP	3
PT_NOTE	4
PT_SHLIB	5
PT_PHDR	6
PT_LOPROC	0x70000000
PT_HIPROC	0x7fffffff
PT_MIPS_REGINFO	0x70000000

PT_NULL indicates that the Program Header entry is unused; the values of the other fields of the entry are undefined.

PT_LOAD indicates a loadable segment, described by *p_filesz* and *p_memsz*. The file bytes are mapped into the beginning of the memory segment. If the memory size is larger than the file size, the extra bytes contain zeros and follows the segment's initialized area. The file size may not be larger than the memory size. Loadable segments appear in the Program Header table in ascending order based on the *p_vaddr* field.

PT_DYNAMIC indicates that the entry contains dynamic linking information. See the Dynamic Section section of this chapter for more details.

PT_INTERP indicates that the entry specifies the location and size of a null-terminated path name to invoke as an interpreter. This type is meaningful only for executable files (though it may occur for shared objects) and may not occur more than once in a file. If a segment of this type is present, it must precede any loadable segment entry.

PT_NOTE indicates that the entry gives the location and size of auxiliary information.

PT_SHLIB is reserved and has unspecified semantics. A program which contains a Program Header entry of this type does not conform to the ABI.

PT_PHDR indicates that the entry specifies the location and size of the Program Header table, both in the file and in the memory image of the program. This type may not occur more than once in a file and it may only occur if the Program Header table is part of the memory image of the program. If present, it must precede any loadable segment entries.

PT_LOPROC through PT_HIPROC values are reserved for processor-specific semantics.

PT_MIPS_REGINFO indicates that this entry specifies register usage information. This type may not occur more than once in a file. Its presence is mandatory and it must precede any loadable segment entry. See the Register Information section in Chapter 11.

p_offset gives the offset from the beginning of the file to the first byte of the segment.

p_vaddr gives the virtual address in memory of the first byte of the segment.

p_paddr is reserved for the segment's physical address (on systems for which physical addressing is relevant).

p_filesz contains the number of bytes in the file image of the segment; the value may be zero.

p_memsz holds the number of bytes in the memory images of the segment; the value may be zero.

p_flags contains the flags associated with the segment. The following flags are defined:

Name	Value	Meaning
PF_X	0x1	Execute
PF_W	0x2	Write
PF_R	0x4	Read
PF_MASKPROC	0xf0000000	Unspecified

All bits in PF_MASKPROC are reserved for processor–specific semantics.

p_align indicates the alignment of segments in memory and in the file. Values 0 and 1 mean no alignment is required. Otherwise, *p_align* should be a positive, integral power of 2, and p_vaddr should equal *p_offset* modulo *p_align*.

Base Address

Executable file and shared object files have a base address, which is the lowest virtual address associated with the process image of the program. The base address is used to relocate the process image of the program during dynamic linking.

During execution, the base address is calculated from the memory load address, the maximum page size, and the lowest virtual address of the program's loadable segment. The virtual addresses in the Program Header might not represent actual virtual addresses (see the Program Loading section of this chapter). The base address is computed by determining the memory address associated with the lowest *p_vaddr* for a PT_LOAD segment, and then truncating this memory address to the nearest multiple of the maximum page size. The memory address may or may not match the *p_addr* values.

Segment Permissions

A program that is to be loaded by the system must have at least one loadable segment, even though this is not required by the file format. When the process image is created, it has access permissions as specified in the *p_flags* field.

If a permission bit is zero, that type of access is denied. All flag combinations are valid but the system may allow more access than requested. However, a segment does not have write permission unless it is specified explicitly. Table 12.1 shows the exact and allowable interpretations for *p_flags*.

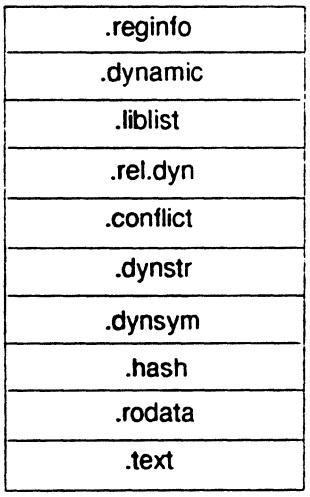
Table 12.1: p_flags Values and Interpretation:

Flags	Value	Exact	Allowable
none	0	All access denied	All access denied
PF_X	1	Execute only	Read, execute
PF_W	2	Write only	Read, write, execute
PF_W+PF_X	3	Write, execute	Read, write, execute
PF_R	4	Read only	Read, execute
PF_R+PF_X	5	Read, execute	Read, execute
PF_R+PF_W	6	Read, write	Read, write, execute
PF_R+PF_W+PF_X	7	Read, write, execute	Read, write, execute

Segment Contents

An object file segment may contain one or more sections. The number of sections in a segment is not important for program loading, but specific information must be present for linking and execution. The figures below illustrate typical segment contents for a MIPS executable or shared object. The order of sections within a segment may vary.

Text segments contain read-only instructions and data, typically including the following sections:



Data segments contain writable data and instructions, typically including the following sections:

.got
.lit4
.lit8
.sdata
.data
.sbss
.bss

Program Loading

As the system creates or augments a process image, it logically copies a file's segment to a virtual memory segment. When, and if, the system physically reads the file depends on the program's execution behavior, system load, etc. A process does not require a physical page unless it references the logical page during execution, and processes commonly leave many pages unreferenced. Therefore, delaying physical reads frequently obviates them, improving system performance. To obtain this efficiency in practice, executable and shared object files must have segment images whose virtual addresses are zero, modulo the file system block size.

Virtual addresses for MIPS text segments must be aligned on 4 K (0x1000) or larger powers of 2 boundaries. MIPS text segments include ELF headers and program headers. MIPS data segments must be aligned on 64 K (0x10000) or larger powers of 2 boundaries. File offsets for MIPS segments must be aligned on 4 K (0x1000) or larger powers of 2 boundaries. Regardless of the 4 K alignment, segments may not overlap in any given 256 K chunk of virtual memory; this helps prevent alias problems in systems with virtual caches. Page size on MIPS systems may vary, but does not exceed 64 K (0x10000).

Figure 12.1 shows an example of an executable file and Table 12.2 shows the Program Header entries for the example text and data segments.

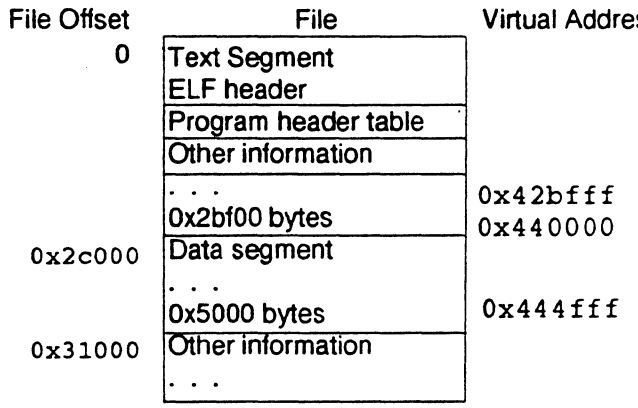


Figure 12.1: Example Executable File

Table 12.1: Text and Data Segments

Member	Text	Data
p_type	PT_LOAD	PT_LOAD
p_offset	0	0x2c000
p_vaddr	0x400000	0x440000
p_paddr	unspecified	unspecified
p_filesz	0x2c000	0x5000
p_memsz	0x2c000	0x7000
p_flags	PF_R+PF_X	PF_R+PF_W+PF_X
p_align	0x10000	0x10000

Because the page size can be larger than the alignment restriction of a segment's file offset, up to four file pages hold impure text or data (depending on page size and file system block size).

- The first text page contains the ELF header, the Program Header table, and other information.
- The last text page may hold a copy of the beginning of data.
- The first data page may have a copy of the end of text.
- The last data page should be zero or else it will conflict with *sbrk* call.

Logically, the system enforces the memory permissions as if each segment were complete and separate; segments' addresses are adjusted to ensure that each logical page in the address space has a single set of permissions. In the example above, with 16KB pages, the region of the file holding the end of text and the beginning of data is mapped twice, at one virtual address for text and at another virtual address for data.

The end of the data segment requires special handling for uninitialized data, which must be set to zeros. If a file's last data page includes information not in the logical memory page, the extraneous data must be set to zero, not the unknown contents of the executable file.

One aspect of segment loading differs between executable files and shared objects. Executable file segments typically contain absolute code. To let the process execute correctly, these segments must reside at the virtual addresses used to build the executable file. Thus the system uses the unchanged *p_vaddr* values as virtual addresses.

On the other hand, shared object segments typically contain position-independent code. This lets a segment's virtual address change from one process to another, without invalidating execution behavior. Though the system chooses virtual addresses for individual processes, it maintains the segments' relative positions. Because position-independent code uses relative addressing between segments, the difference between virtual addresses in memory must match the difference between virtual addresses in the file. The following table shows possible shared object virtual address assignments for several processes, illustrating constant relative positioning. The table also illustrates base address computations.

Table 12.2: Example Shared Object Segment Addresses

Source	Text	Data	Base Address
File	0x200	0x2b000	0x0
Process1	0x50000200	x5002b000	0x50000000
Process2	0x50010200	0x5003b000	0x50010000
Process3	0x60020200	0x6004b000	0x60020000
Process4	0x60030200	0x6005b000	0x60030000

Dynamic Linking

Program Interpreter

An executable file can have only one PT_INTERP Program Header entry. When the system calls *exec*(2) to start the process, the path name of the interpreter is retrieved from the PT_INTERP segment and the initial process image is created from the interpreter file's segments. It is then the interpreter's responsibility to receive control from the system and create the application program's environment.

The interpreter receives control in one of two ways. First, it may receive the file descriptor of the executable file, positioned at the beginning of the file. The file descriptor can then be used to read or map the executable file's segments into memory. Second, depending on the executable file format, the system may load the executable file into memory before giving control to the interpreter. With the possible exception of the file descriptor, the interpreter's initial process state is the same as what the executable file would have received. The interpreter cannot require a second interpreter and may be either a shared object or an executable file.

A shared object is loaded as position-independent with addresses that may vary from one process to another; the system creates the segments in the dynamic segment area used by *mmap*(2) and related services. As a result, a shared object interpreter typically does not conflict with the executable file's original segment addresses.

An executable file is loaded at fixed addresses; the system creates its segments using the virtual addresses from the Program Header table. Consequently, an executable file interpreter's virtual addresses may conflict with those of the executable file. The interpreter is responsible for resolving any conflicts.

Dynamic Linker

When building an executable file that uses dynamic linking, the link editor adds a Program Header entry of type PT_INTERP to the executable file. This entry tells the system to invoke the dynamic linker as the program interpreter. Typically, the dynamic linker requested is *libsys*, the system library. *exec*(2) and the dynamic linker cooperate to create the process image, which involves the following:

- Adding the file segments to the process image.
- Adding shared object segments to the process image.
- Performing relocations for the executable file and its shared objects.

- Closing the file descriptor for the executable file, if a file descriptor was passed to the dynamic linker.
- Transferring control to the program, making it appear that the program received control directly from *exec*(2).

The link editor also constructs various data for shared objects and executable file that assist the dynamic linker. These data are located in loadable segments, are available during execution, and consist of the following:

- A dynamic section of type SHT_DYNAMIC holds various data, including a structure that resides at the beginning of the section and hold the addresses of other dynamic linking information.
- The *.hash* section of type SHT_HASH contains a symbol hash table.
- The *.got* and *.plt* sections, of type SHT_PROGBITS, contain the Global Offset Table and the Procedure Linkage Table, respectively.

Shared objects may be located at virtual addresses that are different from the addresses in the Program Header table. The dynamic linker relocates the memory image and updates absolute addresses before control is given to the program.

If the environment variable LD_BIND_NOW has a non-null value, the dynamic linker processes all relocations before transferring control to the program. The dynamic linker may evaluate procedure linkage table entries lazily, avoiding symbol resolution and relocation for functions that are not called.

The dynamic linker performs linking of objects at run-time and is invoked either through the operating system kernel or by start-up code in the executable. In either case, the initial entry point for the dynamic linker is in entry zero of the Global Offset Table. Each entry should be considered a subroutine:

void entry+0( );
Normal entry point for the dynamic linker when invoked by the operating system kernel. This entry takes no arguments and returns no values.

*void entry+8(Elf32_Addr base, char **envp);*
This entry point is 8 bytes beyond the entry point given by location zero in the GOT. This entry is called when the dynamic linker is invoked by start-up code in the executable. The argument *base* should be the value of the extern _BASE_ADDRESS. It is a pointer to the first location in the text

segment. The *envp* argument is a pointer to the environment. This entry point returns a pointer to the dynamic linker's object list.

Dynamic Section

An object file that is used in dynamic linking has an entry in its Program Header Table of type PT_DYNAMIC. This segment contains the *.dynamic* section, which is labeled _DYNAMIC and is an array with entries of the following type:

```
typedef struct {
        Elf32_Sword       d_tag;
        union {
              Elf32_Word  d_val;
              Elf32_Addr  d_ptr;
        } d_un;
} Elf32_Dyn;
```

dtag indicates how the *d_un* field is to be interpreted.

d_val represents integer values.

d_ptr represents program virtual address. A file's virtual addresses may not match the memory virtual addresses during execution. The dynamic linker computes actual addresses based on the virtual address from the file and the memory base address. Object files do not contain relocation entries to correct addresses in the dynamic structure.

The tag (*d_tag*) requirements for executable and shared object files are summarized in the following table. If the executable entry indicates

mandatory, the dynamic linking array must contain an entry of that type. Optional indicates that an entry for the tag may exist but is not required.

Name	Value	d_un	Executable	Shared Object
DT_NULL	0	ignored	mandatory	mandatory
DT_NEEDED	1	d_val	optional	optional
DT_PLTRELSZ	2	d_val	optional	optional
DT_PLTGOT	3	d_ptr	optional	optional
DT_HASH	4	d_ptr	mandatory	mandatory
ST_STRTAB	5	d_ptr	mandatory	mandatory
DT_RELA	7	d_ptr	mandatory	optional
DT_RELASZ	8	d_val	mandatory	optional
DT_RELAENT	9	d_val	mandatory	optional
DT_STRSZ	10	d_val	mandatory	mandatory
DT_SYMENT	11	d_val	mandatory	mandatory
DT_INIT	12	d_ptr	optional	optional
DT_FINI	13	d_ptr	optional	optional
DT_SONAME	14	d_val	ignored	optional
DT_RPATH	15	d_val	optional	ignored
DT_SYMBOLIC	16	ignored	igmored	optional
DT_REL	17	d_ptr	mandatory	optional
DT_RELSZ	18	d_val	mandatory	optional
DT_RELENT	19	d_val	mandatory	optional
DT_PLTREL	20	d_val	optional	optional
DT_DEBUG	21	d_ptr	optional	ignored
DT_TEXTREL	22	ignored	optional	optional
DT_JMPREL	23	d_ptr	optional	optional
DT_LOPROC	0x70000000	unspecified	unspecified	unspecified
DT_HIPROC	0x7fffffff	unspecified	unspecified	unsepcified

DT_NULL
: An entry of this type marks the end of the _DYNAMIC array.

DT_NEEDED
: This element contains the string table offset of a null terminated string that is the name of a library. The offset is an index into the table indicated in the DT_STRTAB entry. The dynamic array may contain multiple entries of this type. The order of this entries is significant.

DT_PLTRELSZ
This element contains the total size in bytes of the relocation entries associated with the Procedure Linkage Table. If an entry of type DT_JMPREL is present, it must have an associated DT_PLTRELSZ entry.

DT_PLTGOT
Procedure Linkage Table and/or the Global Offset Table.

DT_HASH
This element contains the address of the symbol hash table.

DT_STRTAB
This entry contains the address of the string table.

DT_SYMTAB
This entry contains the address of the symbol table with Elf32_Sym entries for the 32–bit class of files.

DT_RELA
This element contains the address of a relocation table. Entries in the table have explicit addends, such as Elf32_Rela. An object file may have multiple relocation sections. When the link editor builds the relocation table for an executable or shared object, these sections are concatenated to form a single table. While the sections are independent in the object file, the dynamic linker sees a single table. When the dynamic linker creates a process image or adds a shared object to a process image, it reads the relocation table and performs the associated actions. If this element is present, the dynamic structure must also contains DT_RELASZ and DT_RELAENT entries. When relocation is mandatory for a file, either DT_RELA or DT_REL may be present.

DT_RELASZ
This entry contains the size in bytes of the DT_RELA relocation table.

DT_RELAENT
This entry contains the size in bytes of the DT_RELA relocation entry.

DT_STRSZ
This element contains the size in bytes of the string table.

DT_SYMENT
This entry contains the size in bytes of a symbol table entry.

DT_INIT
This element contains the address of the initialization function.

DT_FINI
This element contains the address of the termination function.

DT_SONAME
This entry contains the string table offset of a null-terminated string that gives the name of the shared object. The offset is an index into the table indicated in the DT_STRTAB entry.

DT_RPATH
This element contains the string table offset of a null-terminated search library search path string. The offset is an index into the table indicated in the DT_STRTAB entry.

DT_SYMBOLIC
If this element is present, the dynamic linker uses a different symbol resolution algorithm for references within a library. The symbol search starts from the shared object instead of the executable file. If the shared object does not supply the referenced symbol, the executable file and other shared objects are searched.

DT_REL
This entry is similar to DT_RELA, except that its table has implicit addends. If this element is present, the dynamic structure must also contain DT_RELASZ and DT_RELENT elements.

DT_RELSZ
This entry contains the size in bytes of the DT_REL relocation table.

DT_RELENT
This entry contains the size in bytes of the DT_REL relocation entry.

DT_PLTREL
This element specifies the type of relocation entry referred to by the Procedure Linkage Table. The d_val member holds DT_REL or DT_RELA, as appropriate. All relocations in a Procedure Linkage Table must use the same relocation.

DT_DEBUG
This entry is used for debugging. Its contents are not specified. Programs that access this entry do not conform to the ABI.

DT_TEXTREL
If this element is not present, then no relocation entry should cause a modification to a non-writable segment. If this element is present, one or more relocation entries might request modifications to a non-writable segment.

DT_JMPREL
: If this element is present, its *d_ptr* field contains the address of relocation entries associated only with the Procedure Linkage Table. The dynamic linker may ignore these entries during process initialization if lazy binding is enabled.

DT_LOPROC through DT_HIPROC
: The values in this range are reserved for processor–specific semantics.

Table 12.3 lists MIPS–specific tags:

Table 12.3: Dynamic Arrays Tags d_tag:

Name	Value	d_un	Executable	Shared Object
DT_MIPS_RLD_VERSION	0x70000001	d_val	mandatory	mandatory
DT_MIPS_TIME_STAMP	0x70000002	d_val	optional	optional
DT_MIPS_ICHECKSUM	0x70000003	d_val	optional	optional
DT_MIPS_IVERSION	0x70000004	d_val	optional	optional
DT_MIPS_FLAGS	0x70000005	d_val	mandatory	mandatory
DT_MIPS_BASE_ADDRESS	0x70000006	d_ptr	mandatory	mandatory
DT_MIPS_CONFLICT	0x70000008	d_ptr	optional	optional
DT_MIPS_LIBLIST	0x70000009	d_ptr	optional	optional
DT_MIPS_LOCAL_GOTNO	0x7000000A	d_val	mandatory	mandatory
DT_MIPS_CONFLICTNO	0x7000000B	d_val	optional	optional
DT_MIPS_LIBLISTNO	0x70000010	d_val	optional	optional
DT_MIPS_SYMTABNO	0x70000011	d_val	optional	optional
DT_MIPS_UNREFEXTNO	0x70000012	d_val	optional	optional
DT_MIPS_GOTSYM	0x70000013	d_val	mandatory	mandatory
DT_MIPS_HIPAGENO	0x70000014	d_val	mandatory	mandatory
DT_MIPS_RLD_MAP	0x70000016	d_val	optional	optional

DT_MIPS_RLD_VERSION
: This element holds an index into the object file's string table, which holds the version of the Runtime Linker Interface. The version is currently 1.

DT_MIPS_TIME_STAMP
: This entry contains a 32–bit time stamp.

DT_MIPS_CHECKSUM
: This elements's value is the sum of all external strings and common sizes.

DT_MIPS_IVERSION
This element holds an index into the object file's string table. The version string is a series of colon (:) separated version strings. An index value of zero means no version string was specified.

DT_MIPS_FLAGS
This entry contains a set of 1-bit flags. Flag definitions appear below.

DT_MIPS_BASE_ADDRESS This element contains the base address as defined in the generic ABI.

DT_MIPS_CONFLICT
This entry contains the address of the .conflict section.

DT_MIPS_LIBLIST
This element contains the address of the .liblist section.

DT_MIPS_LOCAL_GOTNO
This element contains the number of local GOT entries.

DT_MIPS_CONFLICTNO
This entry contains the number of entries in the .conflict section and is mandatory if there is a .conflict section.

DT_MIPS_RLD_MAP
This entry contains the address of the word that contains a pointer to the dynamic linker's object list.

DT_MIPS_SYMTABNO
This entry indicates the number of entries in the .dynsym section.

DT_MIPS_LIBLISTNO
This element indicates the number of entries in the *.liblist* section.

DT_MIPS_UNREFEXTNO
This field holds the indes into the dynamic symbol table which is the entry of the first external symbol that is not referenced within the same object.

DT_MIPS_GOTSYM
This field holds the index of the first dynamic symbol table entry that corresponds to an entry in the global offset table.

DT_MIPS_HIPAGENO
This field holds the number of page table entries in the global offset table. A page table entry here refers to a 64K byte chunk of data space. This field is used by profiling tools and is optional.

Entries may appear in any order, except for the relative order of the DT_NEEDED elements and the DT_NULL element at the end of the array. All other tag values are reserved.

The following flags are defined for DT_MIPS_FLAGS:

RHF_NONE	0x00000000	none
RHF_HARDWAY	0x00000001	ignore shortcut pointers
RHF_NOTPOT	0x00000002	hash size not a power of two

Shared Object Dependencies

When the link editor processes an archive library, library members are extracted and copied into the output object file. These statically linked services are available during execution and do not involve the dynamic linker. Shared objects also provide services which require the dynamic linker to include the appropriate shared object files in the process image. To accomplish this, executable and shared object files must describe their dependencies.

The dependencies, indicated in the DT_NEEDED entries of the dynamic structure, tell what shared objects are required for the program. The dynamic linker builds a process image by connecting the referenced shared objects and their dependencies. When resolving symbolic references, the dynamic linker looks first at the symbol table of the executable program, then at the symbol tables of the DT_NEEDED entries (in order), then at the second level DT_NEEDED entries, and so on. Shared object files must be readable by the process.

Note: Even if a shared object is referenced more than once in the dependency list, the dynamic linker only includes one instance of the object in the process image.

Names in the dependency list are copies of either the DT_SONAME strings or the path names of the shared objects used to build the object file. If the link editor builds an executable file from a shared object with a DT_SONAME entry of *liba* and another shared object with path name */usr/lib/libz*, the executable file contains *liba* and */usr/lib/libz* in its dependency list.

If a shabred object name has one or more slash characters in its name, such as */usr/lib/libz*, the dynamic linker uses the string as the path name. If the name has no slashes, such as *liba*, the object is searched for as follows:

- First, the dynamic array tag DT_RPATH may give a string that holds a list of directories separated by colons, such as */usr/new/lib:/usr/local/lib*. The dynamic linker searches these directories in order and if a library is not located, also looks in the current directory.

- Second, the environment variable LD_LIBRARY_PATH may hold a list of colon separated directories, optionally followed by a semicolon and another directory list. These directories are searched after those specified by DT_RPATH.
- Finally, if the library was not located in any of the directories specified by DT_PATH or LD_LIBRARY_PATH, the dynamic linker searches */lib*, then */usr/lib*, and then */usr/lib/cmplrs/cc*.

MIPS defines the following environment variables:

_RLD_PATH	path to dynamic linker (rld)
_RLD_ARGS	argument list to dynamic linker
_RLD_ROOT	prefix dynamic linker prepends to all paths

Note: For security, the dynamic linker ignores environmental search specifications, such as LD_LIBRARY_PATH, for set-user-ID and set-group-ID programs.

Global Offset Table (GOT)

Position-independent code cannot, in general, contain absolute virtual addresses. Global Offset Tables (GOT) hold absolute addresses in private data, thus making the addresses available without compromising the position-independence and sharability of a program's text. A program references its Global Offset Table using position-independent addressing and extracts absolute values, thus redirecting position-independent references to absolute locations.

The Global Offset Table is split into two logically separate subtables: locals and externals. Local entries reside in the first part of the table; these are entries for which there are standard local relocation entries. These entries only require relocation if they occur in a shared object and the shared object's memory load address differs from the virtual address of the shared object's loadable segments. As with the defined external entries in the Global Offset Table, these local entries contain actual addresses.

External entries reside in the second part of the section. Each entry in the external part of the GOT corresponds to an entry in the *.dynsym* section. The first symbol in the *.dynsym* section corresponds to the first word of the table, the second symbol corresponds to the second word, and so on. Each word in the external entry part of the GOT contains the *actual address* for its corresponding symbol. The external entries for defined symbols must contain actual addresses. If an entry corresponds to an undefined symbol and the table entry contains a zero, the entry must be resolved by the dynamic linker, even if the dynamic linker is performing a *quickstart*. See the Quickstart section of this chapter for more information.

After the system creates memory segments for a loadable object file, the dynamic linker may process the relocation entries. The only relocation entries remaining are type R_MIPS_REL32, referring to local entries in the GOT and data containing addresses. The dynamic linker determines the associated symbol (or section) values, calculates their absolute addresses, and sets the proper values. Although the absolute addresses may be unknown when the link editor builds an object file, the dynamic linker knows the addresses of all memory segments and can find the correct symbols and calculate the absolute addresses.

If a program requires direct access to the absolute address of a symbol, it uses the appropriate GOT entry. Because the executable file and shared objects have separate Global Offset Tables, a symbol's address may appear in several tables. The dynamic linker processes all necessary relocations before giving control to the process image, thus ensuring the absolute addresses are available during execution.

The zero entry of the *.dynsym* section is reserved and holds a null symbol table entry. The corresponding zero entry in the GOT is reserved to hold the address of the entry point in the dynamic linker to call when lazy resolving text symbols; see the Procedure Linkage Table section in this chapter. When a program begins execution, it must check this entry and if it is zero, the program must invoke the dynamic linker; otherwise, the system has done so for the program as part of program loading.

If the system has not invoked the dynamic linker and the program fails to map in a dynamic linker, or the program fails to find a dynamic linker, then the program must execute a BREAK instruction with a code of 10. This allows the parent program to determine the reason for failure.

The system may choose different memory segment addresses for the same shared object in different programs; it may even choose different library addresses for different executions of the same program. Nonetheless, memory segments do not change addresses once the process image is established. As long as a process exists, its memory segments reside at fixed virtual addresses.

Calling Position Independent Functions

The GOT is used to hold addresses of position independent functions as well as data addresses. It is not possible to resolve function calls from one executable or shared object to another at static link time, so all of the function address entries in the GOT would normally be resolved at execution time. The dynamic linker would then resolve all of these undefined relocation entries at run-time. Through the use of specially constructed pieces of code known as stubs this run-time resolution can be be deferred through a technique known as *lazy binding*.

Using this technique, the link editor (or a combination of the compiler, assembler, and link editor) builds a stub for each called function, and allocates a GOT entry that initially points to the stub. Because of the normal calling sequence for position independent code, the call ends up invoking the stub the first time the call is made.

```
stub_xyz: .
        lw   t9, 0(gp)
        move t7, ra
        jal  t9
        li   t8, .dynsym_index
```

The stub code loads register *t9* with an entry from the GOT which contains a well-known entry point in the dynamic linker; it also loads register *t8* with the index into the *.dynsym* section of the referenced external. The code saves register *ra* and transfers control to the dynamic linker. The dynamic linker determines the correct address for the called function and replaces the address of the stub in the GOT with the address of the function.

Most undefined text references can be handled by lazy text evaluation except when the address of a function is used other than in a jump and link instruction. In this case, rather than the actual address of the function you would get the address of the stub.

The dynamic linker detects this usage in the following manner:

The link editor generates symbol table entries for all function references with the *st_shndx* field containing SHN_UNDEF and the *st_type* field containing STT_FUNC. The dynamic linker examines each symbol table entry when it starts execution. If the *st_value* field for one of these symbols is non-zero then there were only jump and link references to the function and nothing need be done to the GOT entry; if the field is zero, then there was some other kind of reference to the function and the GOT entry must be replaced with the actual address of the referenced function.

The LD_BIND_NOW environment variable can also change dynamic linking behavior. If its value is non-null, the dynamic linker evaluates all

symbol table entries of type STT_FUNC, replacing their stub addresses in the GOT with the actual address of the referenced function.

Note: Lazy binding generally improves overall application performance, because unused symbols do not incur the dynamic linking overhead. Nevertheless, two situations make lazy binding undesirable for some applications. First, the initial reference to a shared object function takes longer than subsequent calls, because the dynamic linker intercepts the call to resolve the symbol. Some applications cannot tolerate this unpredictability. Second, if an error occurs and the dynamic linker cannot resolve the symbol, the dynamic linker terminates the program. Under lazy binding, this might occur at arbitrary times. Once again, some applications cannot tolerate this unpredictability. By turning off lazy binding, the dynamic linker forces the failure to occur during process initialization, before the application receives control.

Symbols

All externally visible symbols, both defined and undefined, must be hashed into the hash table.

Undefined symbols of type STT_FUNC which have been referenced only by jump and link instructions may contain non-zero values in the their *st_value* field denoting the stub address used for lazy evaluation for this symbol. The run-time linker uses this to reset the GOT entry for this external to its stub address when unlinking a shared object. All other undefined symbols must contain zero in their *st_value* fields.

Defined symbols in an executable may not be preempted. The symbol table in the executable is always searched first to resolve any symbol references. The executable may or may not contain position independent code.

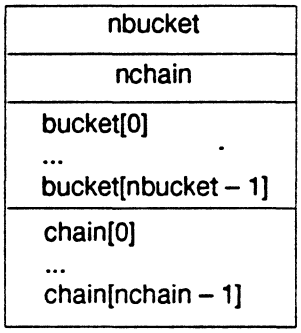
Relocations

There may be only one dynamic relocation section to resolve addresses in data and local entries in the GOT. It must be called *.rel.dyn*. Executables may contain normal relocation sections in addition to a dynamic relocation section. The normal relocation sections may contain resolutions for any absolute values in the main program. The dynamic linker does not resolve these or relocate the main program.

As noted previously, only R_MIPS_REL32 relocation entries are supported in the dynamic relocation section.

Hash table

A hash table of Elf32_Word entries supplies symbol table access. The hash table can be viewed as follows:



nbucket indicates the number of entries in the *bucket* array and *nchain* indicates the number of entries in the *chain* array. Both *bucket* and *chain* hold symbol table indexes; the entries in *chain* parallel the symbol table. The number of symbol table entries should be equal to *nchain*; symbol tables indexes also select *chain* entries.

A hashing function accepts a symbol name and returns a value that may be used to compute a *bucket* index. If the hashing function returns the value *X* for a name, *bucket*[*X* % *nbucket*] gives an index, *Y*, into the symbol table and *chain* array. If the symbol table entry indicated is not the correct one, *chain*[*Y*] indicates the next symbol table entry with the same hash value. The *chain* links can be followed until either the desired symbol table entry is located, or the *chain* entry contains the value STN_UNDEF.

Initialization and Termination Functions

After the dynamic linker has created the process image and performed relocations, each shared object gets the opportunity to execute initialization code. The initialization functions are called in no particular order, but all shared object initialization occurs before the executable file gains control.

Similarly, shared object may have termination functions that are executed by the *atexit*(3) mechanism when the process is being terminated. The order in which the dynamic linker calls the termination functions is unspecified.

Shared objects designate initialization and termination functions through DT_INIT and DT_FINI entries in the dynamic structure. Typically, the code for these functions resides in the *.init* and *.fini* sections.

Note: Although the *atexit*(3) termination processing normally is done, it is not guaranteed to have executed upon process death. In particular, the process does not execute the termination processing if it calls *_exit*(2) or if the process dies because it received a signal that it neither caught nor ignored.

Quickstart

MIPS supports several sections which are useful for faster startup of programs that have been linked with shared objects. Some ordering constraints are imposed on these sections. The group of structures defined in these sections and the ordering constraints allow the dynamic linker to operate more efficiently. These additional sections are also used for more complete dynamic shared object version control.

Note: An ABI compliant system may ignore any of the three sections defined here, but if it supports one of these sections, it must support all three. If an object file is relinked or relocated on secondary stoarage and these sections cannot be processed, they must be deleted.

Shared Object List

A shared object list section is an array of *Elf32_Lib* structures which contains information about the various dynamic shared objects used to statically link the object file. Each shared object used has an entry in the array; each entry has the following format:

```
typedef struct {
        Elf32_Word l_name;
        Elf32_Word l_time_stamp;
        Elf32_Word l_checksum;
        Elf32_Word l_version;
        Elf32_Word l_flags;
} Elf32_Lib;
```

l_name
: This member specifies the name of a shared object. Its value is a string table index. This name may be a trailing component of the path to be used with RPATH + LD_LIBPATH, a name containing '/'s which is relative to '.', or it may be a full path name.

l_time_stamp
: This member's value is a 32-bit time stamp. The value can be combined with the *l_checksum* value and the *l_version* string to form a unique id for this shared object.

l_checksum
This member's value is the sum of all externally visible symbols' string names and common sizes.

l_version
This member specifies the interface version. Its value is a string table index. The interface version is a single string containing no colons. It is compared to a colon separated string of versions pointed to by a dynamic section entry of the shared object. Shared objects with matching names may be considered incompatible if the interface version strings are deemed incompatible. An index value of zero means no version string is specified.

l_flags
This member is a set of 1-bit flags. The following flags are defined:

LL_EXACT_MATCH	0x00000001	require exact match
LL_IGNORE_INT_VER	0x00000002	ignore interface version

LL_EXACT_MATCH

At run-time use a unique id composed of the *l_time_stamp*, *l_checksum*, and *l_version* fields to demand that the run-time dynamic shared object match exactly the shared object used at static link time.

LL_IGNORE_INT_VER

At run-time, ignore any version incompatibility between the dynamic shared object and the object used at static link time.

Normally, if neither LL_EXACT_MATCH nor LL_IGNORE_INT_VER bits are set, the dynamic linker requires that the version of the dynamic shared library match at least one of the colon separated version strings indexed by the l_version string table index.

Conflict Section

Each *.conflict* section is an array of indexes into the *.dynsym* section. Each index identifies a symbol whose attributes conflict with a shared object on which it depends, either in type or size, such that this definition preempts the shared object's definition. The dependent shared object is identified at static link time. The *.conflict* section is an array of Elf32_Conflict elements.

```
typedef Elf32_Addr Elf32_Conflict;
```

Ordering

In order to take advantage of Quickstart functionality, ordering constraints are imposed on the *.dynsym* and *.rel.dyn* sections. The *.dynsym* section must be ordered on increasing values of the *st_value* field. Note that this requires the *.got* section to be ordered in the same way, since it must correspond to the *.dynsym* section.

The *.rel.dyn* section must have all local entries first, followed by the external entries. Within these sub–sections, the entries must be ordered by symbol index. This groups each symbol's relocations together.

Instruction Summary

A

The tables in this chapter summarize the assembly language instruction set. Most of the assembly language instructions have direct machine equivalents. Refer to Appendix A and Appendix B of the *MIPS RISC Architecture* book published by Prentice–Hall for detailed instruction descriptions. In the tables in this appendix, the operand terms have the following meanings:

Operand	Description
destination	Destination register.
address	Expression.
source	Source register.
expression	Aboslute value.
immediate	Immediate value.
label	Symbol label.
breakcode	Value that determines the break.

Table A.1: Main Processor Instruction Summary

Description	Op-code	Operand
Load Address Load Byte Load Byte Unsigned Load Halfword Load Halfword Unsigned Load Word Load Coprocessor z Load Word Left Load Word Right	la lb lbu lh lhu lw lwcz lwl lwr	destination,address
Store Byte Store Halfword Store Word Store Word Coprocessor z Store Word Left Store Word Right Unaligned Load Halfword Unaligned Load Halfword Unsigned Unaligned Load Word Unaligned Store Halfword Unaligned Store Word	sb sh sw swcz swl swr ulh ulhu ulw ush usw	source,address
Conditional Trap Trap if Equal Trap if not Equal Trap if Less Than Trap if Less than, Unsigned Trap if Greater Than or Equal Trap if Greater than or Equal, Unsigned	 teq tne tlt tltu tge tgeu	 src1, src2 src1, immediate

Table A.1: Main Processor Instruction Summary (continued)

Description	Op-code	Operand
Load Immediate Load Upper Immediate Restore From Exception Syscall	li lui rfe syscall	destination,expression
Absolute Value Negate (with overflow) Negate (without overflow) NOT	abs neg negu not	destination,src1 destination/src1
Add (with overflow) Add (without overflow) AND Divide (with overflow) Divide (without overflow) EXCLUSIVE OR Multiply Multiply (with overflow) Multiply (with overflow) Unsigned NOT OR OR Remainder Remainder Unsigned Rotate Left Rotate Right Set Equal SEt Less Than Set Less Than Unsigned Set Less/Equal Set Less/Equal Unsigned Set Greater Than Set Greater Than Unsigned Set Greater/Equal Set Greater/Equal Unsigned Set Not Equal Shift Left Logical Shift Right Arithmetic Shift Right Logical Subtract (with overflow) Subtract (without overflow)	add addu and div divu xor mul mulo mulou nor or rem remu rol ror seq slt sltu sle sleu sgt sgut sge sgeu sne sll sra srl sub subu	destination,src1,src2 destination/src1,src2 destination,src1,immediate destination/src1,immediate
Multiply Multiply Unsigned	mult multu	src1,src2

Table A.1: Main Processor Instruction Summary (continued)

Description	Op-code	Operand
Branch Branch Coprocessor z True Branch Coprocessor z False	b bczt bczf	label
Branch on Equal Branch on Greater Branch on Greater/Equal Branch on Greater/Equal Unsigned Branch on Greater Than Unsigned Branch on Less Branch on Less/Equal Branch on Less/Equal Unsigned Branch on Less Than Unsigned Branch on Not Equal	beq bgt bge bgeu bgtu blt ble bleu bltu bne	src1,src2,label src1,immediate,label
Branch and Link	bal	label
Branch on Equal Zero Branch on Greater/Equal Zero Branch on Greater or Equal to zero and Link Branch on Greater Than Zero Branch on Less/Equal Zero Branch on Less Than Zero Branch on Less Than Zero andLink Branch on Not Equal Zero	beqz bgez bgezal bgtz blez bltz bltzal bnqz	src1,label
Jump Jump and Link	j jal	address src1
Break	break	breakcode
Coprocessor z Operation	cz	expression
Move	move	destination,src1
Move From HI Register Move To HI Register Move From LO Register Move To LO Register	mfhi mthi mflo mtlo	register
Move From Coprocessor z Move To Coprocessor z	mfcz mtcz	dest-gpr, source src-gpr, destination
Control From Coprocessor z Control to Coprocessor z	cfcz ctcz	src-gpr, destination dest-gpr, source

Table A.2: System Coprocessor Instruction Summary

Description	Op-code	Operand
Translation Lookaside Buffer Probe	tlbp	
Translation Lookaside Buffer Read	tlbr	
Translation Lookaside Buffer Write Random	tlbwr	
Translation Lookaside Write Index	tlbwi	

Table A.3: Floating Point Instruction Summary

Description	Op-code	Operand
Load Fp		
Double	l.d	destination,offset(base)
Single	l.s	
Store FP		
Double	s.d	source,offset(base)
Single	s.s	
Absolute Value Fp		
Double	abs.d	destination,src1
Single	abs.s	
Add Fp		
Double	add.d	destination,src1,src2
Single	add.s	
Divide Fp		
Double	div.d	
Single	div.s	
Multiply		
Double	mul.d	
Single	mul.s	
Subtract Fp		
Double	sub.d	
Single	sub.s	

Table A.3: Floating Point Instruction Summary (continued)

Description	Op-code	Operand
Convert Source to Specified Precision Fp		
Double to Single	cvt.s.d	destination,src2
Fixed Point to Single	cvt.s.w	
Fixed Point to Double	cvt.d.w	
Single to Double	cvt.d.s	
Double to Fixed Point	cvt.w.d	
Single to Fixed Point	cvt.w.s	
Negate Floating Point		
Double	neg.d	
Single	neg.s	

Table A.3: Floating Point Instruction Summary (continued)

Description	Op-code	Operand
Compare Fp		
F Single	c.f.s	src1,src2
F Double	c.f.d	
UN Single	c.un.s	
UN Double	c.un.d	
*EQ Single	c.eq.s	
*EQ Double	c.eq.d	
UEQ Single	c.ueq.s	
UEQ Double	c.ueq.d	
OLT Single	c.olt.s	
OLT Double	c.olt.d	
ULT Single	c.ult.s	
ULT Double	c.ult.d	
OLE Single	c.ole.s	
OLE Double	c.ole.d	
ULE Single	c.ule.s	
ULE Double	c.ule.d	
SF Single	c.sf.s	
SF Double	c.sf.d	
NGLE Single	c.ngle.s	
NGLE Double	c.ngle.d	
SEQ Single	c.deq.s	
SEQ Double	c.seq.d	
NGL Single	c.ngl.s	
NGL Double	c.ngl.d	

Note: Starred items (*) are the most common *Compare* instructions. The other *Compare* instructions are for IEEE compatibility.

Table A.3: Table Floating Point Instruction Summary (continued)

Description	Op-code	Operand
Compare Fp		
*LT Single	c.lt.s	src1,src2
*LT Double	c.lt.d	
NGE Single	c.nge.s	
NGE Double	c.nge.d	
*LE Single	c.le.s	
*LE Double	c.le.d	
NGT Single	c.ngt.s	
NGT Double	c.ngt.d	
Move Fp		
Single	mov.s	destination,src1
Double	mov.d	

Note: Starred items (*) are the most common Compare instructions. The other Compare instructions are for IEEE compatibility.

Basic Machine Definition

B

The assembly language instructions described in this book are a superset of the actual machine instructions. Generally, the assembly language instructions match the machine instructions; however, in some cases the assembly language instruction are macros that generate more than one machine instruction (the assembly language multiplication instructions are examples).

You can, in most instances, consider the assembly instructions as machine instructions; however, for routines that require tight coding for performance reasons, you must be aware of the assembly instructions that generate more than one machine language instruction, as described in this appendix.

Load and Store Instructions

If you use an *address* as an operand in an assembler *Load* or *Store* instruction and the address references a data item that is not addressable through register *$gp* or the data item does not have an absolute address in the range –32768...32767, the assembler instruction generates a *lui* (load upper immediate) machine instruction and generates the appropriate offset to *$at*. The assembler then uses *$at* as the index address for the reference. This condition occurs when the address has a relocatable external name offset (or index) from where the offset began.

The assembler's *la* (load address) instruction generates an *addiu* (add unsigned immediate) machine instruction. If the address requires it, the *la* instruction also generates a *lui* (load upper immediate) machine instruction. The machine requires the *la* instruction because *la* couples relocatable information with the instruction for symbolic addresses.

Depending on the expression's value, the assembler's *li* (load immediate) instruction can generate one or two machine instructions. For values in the –32768...65535 range or for values that have zeros as the 16 least significant bits, the *li* instruction generates a single machine instruction; otherwise it generates two machine instructions.

Computational Instructions

If a computational instruction immediate value falls outside the 0...65535 range for Logical ANDs, Logical ORs, or Logical XORs (exclusive or), the immediate field causes the machine to explicitly load a constant to a temporary register. Other instructions generate a single machine instruction when a value falls in the –32768...32767 range.

The assembler's *seq* (set equal) and *sne* (set not equal) instructions generate three machine instructions each.

If one operand is a literal outside the range –32768...32767, the assembler's *sge* (set greater than or equal to) and *sle* (set less/equal) instructions generate two machine instructions each.

The assembler's *mulo* and *mulou* (multiply) instructions generate machine instructions to test for overflow and to move the result to a general register; if the destination register is *$0*, the check and move are not generated.

The assembler's *mul* (multiply unsigned) instruction generates a machine instruction to move the result to a general register; if the destination register is *$0*, the move and divide–by–zero checking is not generated. The assembler's divide instructions, *div* (divide with overflow) and *divu* (divide without overflow), generate machine instructions to check for division by zero and to move the quotient into a general register; if the destination register is *$0*, the move is not generated.

The assembler's *rem* (signed) and *remu* (unsigned) instructions also generate multiple instructions.

The rotate instructions *ror* (rotate right) and *rol* (rotate left) generate three machine instructions each.

The abs (absolute value) instruction generates three machine instructions.

Branch Instructions

If the immediate value is not zero, the branch instructions *beq* (branch on equal) and *bne* (branch on not equal), each generate a load literal machine instruction. The relational instructions generate a *slt* (set less than) machine instruction to determine whether one register is less than or greater than another. Relational instructions can reorder the operands and branch on either zero or not zero as required to do an operation.

Coprocessor Instructions

For symbolic addresses, the coprocessor interface *Load* and *Store* instructions, *lcz* (load coprocessor z) and *scz* (store coprocessor z) can generate a *lui* (load upper immediate) machine instruction.

Special Instructions

The assembler's *break* instruction packs the *breakcode* operand in unused register fields. An operating system convention determines the position.

Index

NEC
NEC Electronics Inc.
CORPORATE HEADQUARTERS
475 Ellis Street
P.O. Box 7241
Mountain View, CA 94039
TEL 415-960-6000

For literature, call toll-free 7 a.m. to 6 p.m. Pacific time: **1-800-366-9782**
or FAX your request to: **1-800-729-9288**

50776