DEC OSF/1

Assembly Language Programmer's Guide

Order Number: AA-PS31A-TE

March 1993

Product Version: DEC OSF/1 Version 1.2 or higher

This manual describes the assembly language supported by the DEC OSF/1 Alpha AXP compiler system, its syntax rules, and how to write some assembly programs.

digital equipment corporation
Maynard, Massachusetts

Contents

About This Manual

Audience xiii
Organization xiii
Other Sources of Information xiv
Conventions xiv

1 Architecture-Based Considerations

1.1 Registers 1–1
  1.1.1 Integer Registers 1–1
  1.1.2 Floating-Point Registers 1–2
1.2 Bit and Byte Ordering 1–2
1.3 Addressing 1–4
  1.3.1 Aligned Data Operations 1–5
  1.3.2 Unaligned Data Operations 1–5
1.4 Exceptions 1–6
  1.4.1 Main Processor Exceptions 1–6
  1.4.2 Floating-Point Processor Exceptions 1–6

2 Lexical Conventions

2.1 Blank and Tab Characters 2–1

2.2 Comments 2–1

2.3 Identifiers 2–1

2.4 Constants 2–2

2.4.1 Scalar Constants 2–2
2.4.2 Floating-Point Constants 2–2
2.4.3 String Constants 2–3

2.5 Multiple Lines Per Physical Line 2–5

2.6 Statements 2–5

2.6.1 Label Definitions 2–5
2.6.2 Null Statements 2–5
2.6.3 Keyword Statements 2–5

2.7 Expressions 2–6

2.7.1 Expression Operators 2–6
2.7.2 Expression Operator Precedence Rules 2–7
2.7.3 Data Types 2–7
2.7.4 Type Propagation in Expressions 2–8

2.8 Address Formats 2–9

3 Main Instruction Set

3.1 Load and Store Instructions 3–2

3.1.1 Load Instruction Descriptions 3–3
3.1.2 Store Instruction Descriptions 3–8

3.2 Arithmetic Instructions 3–9

3.3 Logical and Shift Instructions 3–18

3.4 Relational Instructions 3–21

3.5 Move Instructions 3–22

3.6 Control Instructions 3–24
3.7 Byte-Manipulation Instructions 3–27
3.8 Special-Purpose Instructions 3–32

4 Floating-Point Instruction Set

4.1 Background Information on Floating-Point Operations 4–2
4.1.1 Floating-Point Data Types 4–3
4.1.2 Floating-Point Control Register 4–4
4.1.3 Floating-Point Exceptions 4–5
4.1.4 Floating-Point Rounding Modes 4–6
4.1.5 Floating-Point Instruction Qualifiers 4–7
4.2 Floating-Point Load and Store Instructions 4–9
4.3 Floating-Point Arithmetic Instructions 4–11
4.4 Floating-Point Relational Instructions 4–14
4.5 Floating-Point Move Instructions 4–15
4.6 Floating-Point Control Instructions 4–17
4.7 Floating-Point Special-Purpose Instructions 4–17

5 Assembler Directives

6 Programming Considerations

6.1 Calling Conventions 6–1
6.2 Program Model 6–2
6.3 General Coding Concerns 6–2
6.3.1 Register Use 6–3
6.3.2 Using Directives to Control Sections and Location Counters . 6–4
6.3.3 The Stack Frame 6–6
6.3.4 Examples 6–10
6.4 Developing Code for Procedure Calls 6–13

6.4.1 Calling a High-Level Language Procedure 6–14
6.4.2 Calling an Assembly-Language Procedure 6–15

6.5 Memory Allocation 6–17

7 Object Files

7.1 Object File Overview 7–1

7.2 Object File Sections 7–4

7.2.1 File Header 7–4
7.2.2 Optional Header 7–6
7.2.3 Section Headers 7–7
7.2.4 Section Data 7–10
7.2.5 Section Relocation Information 7–12

7.2.5.1 Relocation Table Entry 7–12
7.2.5.2 Assembler and Linker Processing of Relocation Entries 7–15

7.3 Object-File Formats (OMAGIC, NMAGIC, ZMAGIC) 7–21

7.3.1 Impure Format (OMAGIC) Files 7–21
7.3.2 Shared Text (NMAGIC) Files 7–22
7.3.3 Demand Paged (ZMAGIC) Files 7–24
7.3.4 Ucode Objects 7–25

7.4 Loading Object Files 7–26

7.5 Archive Files 7–26

7.6 Linker Defined Symbols 7–27

8 Symbol Table

8.1 Symbol Table Overview 8–1

8.2 Format of Symbol Table Entries 8–8

8.2.1 Symbolic Header 8–8
8.2.2 Line Number Table 8–9
8.2.3 Procedure Descriptor Table 8–13

8.2.4 Local Symbol Table 8–14

8.2.4.1 Symbol Type (st) Constants 8–16
8.2.4.2 Storage Class (sc) Constants 8–17

8.2.5 Optimization Symbol Table 8–17
8.2.6 Auxiliary Symbol Table 8–18
8.2.7 File Descriptor Table 8–21
8.2.8 External Symbol Table 8–21

9 Program Loading and Dynamic Linking

9.1 Object File Considerations 9–1

9.1.1 Structures 9–1
9.1.2 Base Addresses 9–2
9.1.3 Segment Access Permissions 9–2
9.1.4 Segment Contents 9–2

9.2 Program Loading 9–3

9.3 Dynamic Linking 9–4

9.3.1 Dynamic Loader 9–4
9.3.2 Dynamic Section (.dynamic) 9–5
9.3.3 Shared Object Dependencies 9–11
9.3.4 Global Offset Table (.got) 9–12
9.3.5 Calling Position-Independent Functions 9–13
9.3.6 Dynamic Symbol Section (.dynsym) 9–15
9.3.7 Dynamic Relocation Section (.rel.dyn) 9–17
9.3.8 Hash Table Section (.hash) 9–18
9.3.9 Dynamic String Section (.dynstr) 9–19
9.3.10 Initialization and Termination Functions 9–20
9.3.11 Quickstart 9–20

9.3.11.1 Shared Object List (.liblist) 9–20
9.3.11.2 Conflict Section (.conflict) 9–22
9.3.11.3 Ordering of Sections 9–22

A Instruction Summaries

B 32-Bit Considerations

B.1 Canonical Form B–1
B.1.1 Canonical Form Operands B–1
B.2 Longword Instructions B–1
B.3 Quadword Instructions for Longword Operations B–2
B.4 Logical Shift Instructions B–3
B.5 Conversions to Quadword B–3
B.6 Conversions to Longword B–3

C Basic Machine Definition

C.1 Implicit Register Use C–1
C.2 Addresses C–2
C.3 Immediate Values C–2
C.4 Load and Store Instructions C–3
C.5 Integer Arithmetic Instructions C–3
C.6 Floating-Point Load Immediate Instructions C–4
C.7 One-to-One Instruction Mappings C–4

D PALcode Instruction Summaries

D.1 Unprivileged PALcode Instructions D–1
D.2 Privileged PALcode Instructions D–2

Index

Examples

6-1: Nonleaf Procedure 6–11

6-2: Leaf Procedure Without Stack Space for Local Variables 6–12

6-3: Leaf Procedure With Stack Space for Local Variables 6–12

Figures

1-1: Byte Ordering 1–4

4-1: Floating-Point Data Formats 4–3

4-2: Floating-Point Control Register 4–4

6-1: Section and Location Counters 6–5

6-2: Stack Organization 6–8

6-3: Layout of Memory (User Program View) 6–18

7-1: Object File Format 7–3

7-2: Organization of Section Data 7–11

7-3: Relocation Table Entry for Undefined External Symbols 7–16

7-4: Relocation Table Entry for a Local Relocation Entry 7–17

7-5: Layout of OMAGIC Files in Virtual Memory 7–22

7-6: Layout of NMAGIC Files in Virtual Memory 7–23

7-7: Layout of ZMAGIC Files 7–25

8-1: The Symbol Table Overview 8–2

8-2: Functional Overview of the Symbolic Header 8–3

8-3: Logical Relationship Between the File Descriptor Table and Local Symbols 8–5

8-4: Physical Relationship of a File Descriptor Entry to Other Tables 8–6

8-5: Logical Relationship Between the File Descriptor Table and Other Tables 8–7

9-1: Text and Data Segments of Object Files 9–3
9-2: Relationship Between .dynsym and .got 9–17
9-3: Hash Table Section 9–19

Tables

2-1: Backslash Conventions 2–4
2-2: Expression Operators 2–6
2-3: Data Types 2–7
2-4: Address Formats 2–9
3-1: Load and Store Formats 3–2
3-2: Load Instruction Descriptions 3–4
3-3: Store Instruction Descriptions 3–8
3-4: Arithmetic Instruction Formats 3–10
3-5: Arithmetic Instruction Descriptions 3–11
3-6: Logical and Shift Instruction Formats 3–19
3-7: Logical and Shift Instruction Descriptions 3–19
3-8: Relational Instruction Formats 3–21
3-9: Relational Instruction Descriptions 3–22
3-10: Move Instruction Formats 3–23
3-11: Move Instruction Descriptions 3–23
3-12: Control Instruction Formats 3–24
3-13: Control Instruction Descriptions 3–25
3-14: Byte-Manipulation Instruction Formats 3–28
3-15: Byte-Manipulation Instruction Descriptions 3–29
3-16: Special-Purpose Instruction Formats 3–33
3-17: Special-Purpose Instruction Descriptions 3–33
4-1: Qualifier Combinations for Floating-Point Instructions 4–9

4-2: Load and Store Instruction Formats 4–10
4-3: Load and Store Instruction Descriptions 4–10
4-4: Arithmetic Instruction Formats 4–11
4-5: Arithmetic Instruction Descriptions 4–12
4-6: Relational Instruction Formats 4–14
4-7: Relational Instruction Descriptions 4–14
4-8: Move Instruction Formats 4–15
4-9: Move Instruction Descriptions 4–16
4-10: Control Instruction Formats 4–17
4-11: Control Instruction Descriptions 4–17
4-12: Special-Purpose Instruction Formats 4–18
4-13: Control Register Instruction Descriptions 4–18
5-1: Summary of Assembler Directives 5–1
6-1: Integer Registers 6–3
6-2: Floating-Point Registers 6–4
6-3: Argument Locations 6–10
7-1: File Header Format 7–4
7-2: File Header Magic Numbers 7–5
7-3: File Header Flags 7–5
7-4: Optional Header Definitions 7–6
7-5: Optional Header Magic Numbers 7–7
7-6: Section Header Format 7–7
7-7: Section Header Constants for Section Names 7–8
7-8: Format of s_flags Section Header Entry 7–8
7-9: Format of a Relocation Table Entry 7–13
7-10: Section Numbers for Local Relocation Entries 7–13
7-11: Relocation Types 7–14
7-12: Literal Usage Types 7–15

7-13: Linker Defined Symbols 7–27
8-1: Format of the Symbolic Header 8–8
8-2: Format of a Line Number Entry 8–9
8-3: Format of a Procedure Descriptor Table Entry 8–13
8-4: Format of a Local Symbol Table Entry 8–14
8-5: Index and Value as a Function of Symbol Type and Storage Class 8–15
8-6: Symbol Type (st) Constants 8–16
8-7: Storage Class Constants 8–17
8-8: Auxiliary Symbol Table Entries 8–18
8-9: Format of an Type Information Record Entry 8–18
8-10: Basic Type (bt) Constants 8–19
8-11: Type Qualifier (tq) Constants 8–20
8-12: Format of File Descriptor Entry 8–21
8-13: External Symbol Table Entries 8–22
9-1: Segment Access Permissions 9–2
9-2: Dynamic Array Tags (d_tag) 9–6
9-3: Processor-Specific Dynamic Array Tags (d_tag) 9–9
A-1: Main Instruction Set Summary A–2
A-2: Floating-Point Instruction Set Summary A–6
A-3: Rounding and Trapping Modes A–9
D-1: Unprivileged PALcode Instructions D–1
D-2: Privileged PALcode Instructions D–2

About This Manual

This book describes the assembly language supported by the DEC OSF/1 compiler 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 `as`(1) and `ld`(1) reference pages.

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 that might have different machine instructions.

The assembler exists primarily to produce object modules from the assembly instructions generated by some high-level language compilers. It therefore lacks many functions normally present in assemblers.

Audience

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

It is recommended that you use the assembler only when you need to perform programming tasks such as the following:

- 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.

Organization

This manual is organized as follows:

Chapter 1 Describes the format for the general registers, the special registers, and the floating point registers. It also describes how addressing works and the exceptions you might encounter with assembly programs.

Chapter 2	Describes the lexical conventions that the assembler follows.
Chapter 3	Describes the main processor's instruction set, including notation, load and store instructions, computational instructions, and jump and branch instructions.
Chapter 4	Describes the floating point instruction set.
Chapter 5	Describes the assembler directives.
Chapter 6	Describes calling conventions for all supported high-level languages. It also discusses memory allocation and register use.
Chapter 7	Provides an overview of the components comprising the object file and describes the headers and sections of the object file.
Chapter 8	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 9	Describes the object file structures that relate to program execution and dynamic linking, and also describes how the process image is created from these files.
Appendix A	Summarizes all assembler instructions.
Appendix B	Describes issues relating to processing 32-bit data.
Appendix C	Describes instructions that generate more than one machine instruction.
Appendix D	Describes the PALcode (privileged architecture library code) instructions required to support an Alpha AXP system.

Other Sources of Information

DEC OSF/1 Programmer's Guide

Alpha Architecture Reference Manual

Conventions

`file`	Italic (slanted) type indicates variable values and instruction operands.

[|] **{ | }**	In syntax definitions, brackets indicate items that are optional and braces indicate items that are required. Vertical bars separating items inside brackets or braces indicate that you choose one item from among those listed.
. . .	In syntax definitions, a horizontal ellipsis indicates that the preceding item can be repeated one or more times.
`cat`(1)	A cross-reference to a reference page includes the appropriate section number in parentheses. For example, `cat`(1) indicates that you can find information on the `cat` command in Section 1 of the reference pages.

Architecture-Based Considerations 1

This chapter describes programming considerations that are determined by the Alpha AXP system architecture. It addresses the following topics:

- Registers (Section 1.1)
- Bit and byte ordering (Section 1.2)
- Addressing (Section 1.3)
- Exceptions (Section 1.4)

1.1 Registers

This section discusses the registers that are available on Alpha AXP systems and describes how memory organization affects them. Refer to Section 6.3 for information regarding register use and linkage.

Alpha AXP systems have the following types of registers:

- Integer registers
- Floating-point registers

You must use integer registers where the assembly instructions expect integer registers and floating-point registers where the assembly instructions expect floating-point registers. If you confuse the two, the assembler issues an error message.

The assembler reserves all register names (see Section 6.3.1). All register names start with a dollar sign ($) and all alphabetic characters in register names are lowercase.

1.1.1 Integer Registers

Alpha AXP systems have 32 integer registers, each of which is 64 bits wide. Integer registers are sometimes referred to as *general* registers in other system architectures.

The integer registers have the names `$0` to `$31`.

By including the file `regdef.h` (use `#include <alpha/regdef.h>`) in your assembly language program, you can use the software names of all of the integer registers, except for `$28`, `$29`, and `$30`. The operating system and the assembler use the integer registers `$28`, `$29`, and `$30` for specific purposes.

Note

> If you need to use the registers reserved for the operating system and the assembler, you must specify their alias names in your program, not their regular names. The alias names for `$28`, `$29`, and `$30` are `$at`, `$gp`, and `$sp`, respectively. To prevent you from using these registers unknowingly and thereby producing potentially unexpected results, the assembler issues warning messages if you specify their regular names in your program.
>
> Integer register `$29` is available as a general register on some compiler systems when the `-G 0` compilation option is specified. It is not available as a general register on Alpha AXP systems under any circumstances.

Integer register `$31` always contains the value 0. All other integer registers can be used interchangeably, except for integer register `$30`, which is assumed to be the stack pointer (certain PALcode assumes that `$30` is the stack pointer). See Table 6-1 for a description of integer register assignments. See Appendix D and the *Alpha Architecture Handbook* for information on PALcode (Privileged Architecture Library code).

1.1.2 Floating-Point Registers

Alpha AXP systems have 32 floating-point registers, each of which is 64 bits wide. Each register can hold one single-precision (32-bit) value or one double-precision (64-bit) value.

The floating-point registers have the names `$f0` to `$f31`.

Floating-point register `$f31` always contains the value 0.0. All other floating-point registers can be used interchangeably. See Table 6-2 for a description of floating-point register assignments.

1.2 Bit and Byte Ordering

A system's byte-ordering scheme (or endian scheme) affects memory organization and defines the relationship between address and byte position of data in memory:

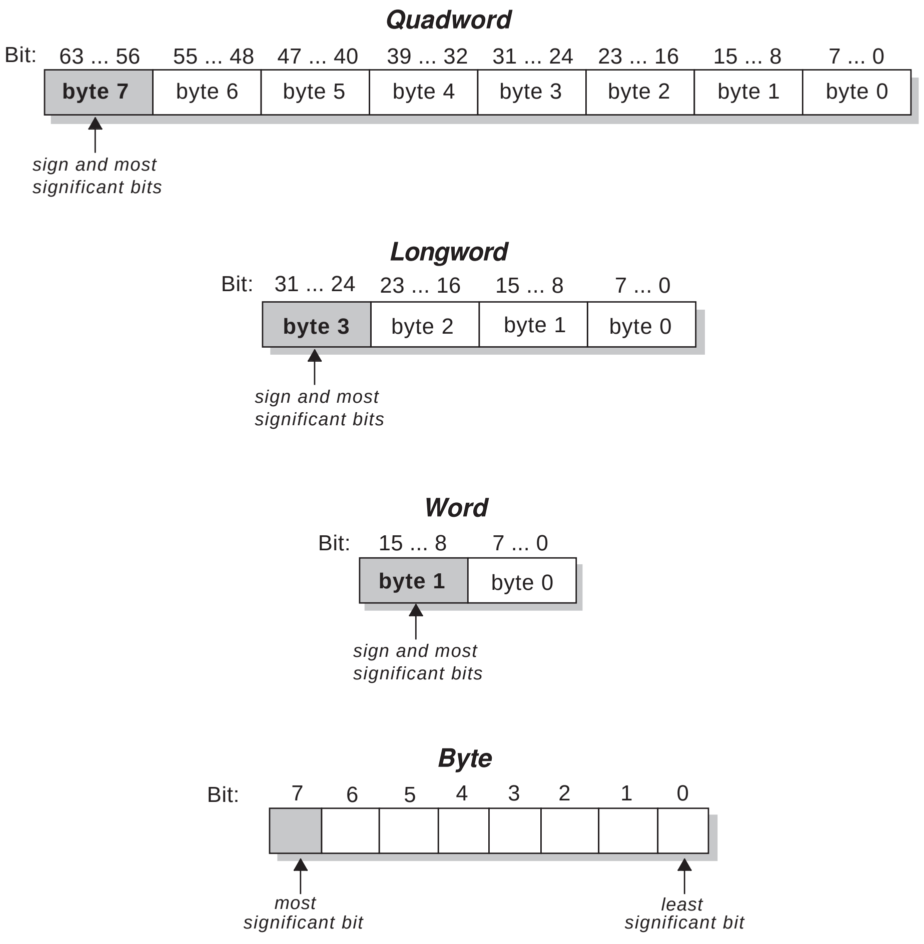
- Big-endian systems store the sign bit in the lowest address byte.
- Little-endian systems store the sign bit in the highest address byte.

Alpha AXP systems use the little-endian scheme. Byte-ordering is as follows:

- The bytes of a quadword are numbered from 7 to 0. Byte 7 holds the sign and most significant bits.
- The bytes of a longword are numbered from 3 to 0. Byte 3 holds the sign and most significant bits.
- The bytes of a word are numbered from 1 to 0. Byte 1 holds the sign and most significant bits.

The bits of each byte are numbered from 7 to 0, using the format shown in Figure 1-1. (No assembler instructions depend on bit numbering; it is a software convention.)

Figure 1-1: Byte Ordering



1.3 Addressing

This section describes the byte-addressing schemes for load and store instructions. (Section 2.8 describes the formats in which you can specify addresses.)

1.3.1 Aligned Data Operations

All Alpha AXP systems use the following byte-addressing scheme for aligned data:

- Access to words requires alignment on byte boundaries that are evenly divisible by two.
- Access to longwords requires alignment on byte boundaries that are evenly divisible by four.
- Access to quadwords requires alignment on byte boundaries that are evenly divisible by eight.

Any attempt to address a data item that does not have the proper alignment causes an alignment exception.

The following instructions load or store aligned data:

- Load quadword (`ldq`)
- Store quadword (`stq`)
- Load longword (`ldl`)
- Store longword (`stl`)
- Load word (`ldw`)
- Store word (`stw`)
- Load word unsigned (`ldwu`)

1.3.2 Unaligned Data Operations

The assembler's unaligned load and store instructions operate on arbitrary byte boundaries. They all generate multiple machine-code instructions. They do not raise alignment exceptions.

The following instructions load and store unaligned data:

- Unaligned load quadword (`uldq`)
- Unaligned store quadword (`ustq`)
- Unaligned load longword (`uldl`)
- Unaligned store longword (`ustl`)
- Unaligned load word (`uldw`)
- Unaligned store word (`ustw`)
- Unaligned load word unsigned (`uldwu`)
- Load byte (`ldb`)
- Store byte (`stb`)

- Load byte unsigned (`ldbu`)

1.4 Exceptions

This section describes the exceptions that you can encounter while running assembly programs. The Alpha AXP system detects some exceptions directly, and other exceptions are signaled as a result of specific tests that are inserted by the assembler. This section describes only those exceptions that occur frequently.

1.4.1 Main Processor Exceptions

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

- Address error exceptions – these occur when an address is invalid for the executing process or, in most instances, when a reference is made to a data item that is not properly aligned.
- Overflow exceptions – these occur when arithmetic operations compute signed values and the destination lacks the precision to store the result.
- Bus exceptions – these occur when an address is invalid for the executing process.
- Divide-by-zero exceptions – these occur when a divisor is zero.

1.4.2 Floating-Point Processor Exceptions

The following exceptions are the most common floating-point exceptions:

- Invalid operation exceptions – these include the following:
  - Magnitude subtraction of infinities, for example: $(+\infty) - (+\infty)$
  - Multiplication of 0 by ∞, with any signs
  - Division of 0 by 0 or ∞ by ∞, 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 integer format. (The `cvttq` instruction converts floating-point numbers to integer formats.)
  - Comparison of predicates that have unordered operands and involve Less Than or Less Than or Equal.
  - Any operation on a signaling NaN. (See the introduction of Chapter 4 for a description of NaN symbols.)
- Divide-by-zero exceptions – these occur when a divisor is zero.

- 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 $\pm 2^{Emin}$ (plus or minus 2 to the minimum expressible exponent).
- Inexact exceptions – these occur if the infinitely precise result differs from the rounded result.

For additional information on floating-point exceptions, see Section 4.1.3.

Lexical Conventions 2

This chapter describes lexical conventions associated with the following items:

- Blank and tab characters (Section 2.1)
- Comments (Section 2.2)
- Identifiers (Section 2.3)
- Constants (Section 2.4)
- Physical lines (Section 2.5)
- Statements (Section 2.6)
- Expressions (Section 2.7)
- Address formats (Section 2.8)

2.1 Blank and Tab Characters

You can use blank and tab characters anywhere between operators, identifiers, and constants. Adjacent identifiers or constants that are not otherwise separated must be separated by a blank or tab.

These characters can also be used within character constants; however, they are not allowed within operators and identifiers.

2.2 Comments

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

Do not start a comment with a number sign in column one; the assembler uses `cpp` (the C language preprocessor) to preprocess assembler code and `cpp` interprets number signs in the first column as preprocessor directives.

2.3 Identifiers

An identifier consists of a case-sensitive sequence of alphanumeric characters (A-Z, a-z, 0-9) and the following special characters:

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

Identifiers can be up to 31 characters long, and the first character cannot be numeric (0-9).

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 name specified by a `.globl` directive (see Chapter 5).

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.

2.4 Constants

The assembler supports the following constants:

- Scalar constants
- Floating-point constants
- String constants

2.4.1 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 the following constants:

- Decimal constants – these consist of a sequence of decimal digits (0-9) without a leading zero.
- Hexadecimal constants – these consist of the characters 0x (or 0X) followed by a sequence of hexadecimal digits (0-9abcdefABCDEF).
- Octal constants – these consist of a leading zero followed by a sequence of octal digits (0-7).

2.4.2 Floating-Point Constants

Floating-point constants can appear only in floating-point directives (see Chapter 5) and in the floating-point load immediate instructions (see Section

4.2). Floating-point constants have the following format:

```
±d1[.d2][e|E±d3]
```

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, the number .02173 can be represented as follows:

```
21.73E-3
```

The floating-point directives (such as `.float` and `.double`) may optionally use hexadecimal floating-point constants instead of decimal constants. A hexadecimal floating-point constant consists of the following elements:

```
[+|-]0x[1|0].<hex-digits>h0x<hex-digits>
```

The assembler places the first set of hexadecimal 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 hexadecimal digits in the exponent field without biasing them. If the mantissa appears to be denormalized, it checks to determine whether the exponent is appropriate. Hexadecimal floating-point constants are useful for generating IEEE special symbols and for writing hardware diagnostics.

For example, either of the following directives generates the single-precision number 1.0:

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

The assembler uses normal (nearest) rounding mode to convert floating-point constants.

2.4.3 String Constants

All characters except the newline character are allowed in string constants. String constants begin and end with double quotation marks (").

The assembler observes most of the backslash conventions used by the C language. Table 2-1 shows the assembler’s backslash conventions.

Table 2-1: 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)
nnn	Character whose octal value is *nnn* (where *n* is 0-7)
Xnn	Character whose hexadecimal value is *nn* (where *n* is 0-9, a-f, or A-F)

Deviations from C conventions are as follows:

- The assembler does not recognize ‘‘\?’’
- The assembler does not recognize the prefix ‘‘L’’ (wide character constant).
- The assembler limits hexadecimal constants to two characters.
- The assembler allows the leading ‘‘x’’ character in a hexadecimal constants to be either uppercase or lowercase; that is, both \x*nn* and \X*nn* are allowed.

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 (that is, use a 0 for the first character of a single-character hexadecimal number).

2.5 Multiple Lines Per Physical Line

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

2.6 Statements

The assembler supports the following types of statements:

- Null statements
- Keyword statements

Each statement consists of an optional label, an operation code, and one or more operands.

2.6.1 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.

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 specified number in the lexically forward or backward direction.

2.6.2 Null Statements

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

```
label: ; ;
```

2.6.3 Keyword Statements

A keyword statement begins with a predefined keyword. The syntax for the rest of the statement depends on the keyword. Keywords are either assembler instructions (mnemonics) or directives. (Assembler instructions in the main instruction set and the floating-point instruction set are described in Chapter 3 and Chapter 4, respectively. Assembler directives are described in Chapter 5.)

2.7 Expressions

An expression is a sequence of symbols that represents a value. Each expression and its result have data types. The assembler does arithmetic in twos complement integers with 64 bits of precision. Expressions follow precedence rules and consist of the following elements:

- Operators
- Identifiers
- Constants

You can also use a single character string in place of an integer within an expression. For example, the following two pairs of statements are equivalent:

```
.byte "a" ; .word "a"+0x19
.byte 0x61 ; .word 0x7a
```

2.7.1 Expression Operators

The assembler supports the operators shown in Table 2-2.

Table 2-2: Expression Operators

Operator	Meaning
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Remainder
<<	Shift left
>>	Shift right (sign is not extended)
^	Bitwise EXCLUSIVE OR
&	Bitwise AND
|	Bitwise OR
-	Minus (unary)
+	Identity (unary)
~	Complement

2.7.2 Expression Operator Precedence Rules

For the order of operator evaluation within expressions, you can rely on the precedence rules or you can group expressions with parentheses. 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. The following table lists the precedence rules from lowest to highest:

Precedence	Operators
Least binding, lowest precedence	Binary +, -
.	
.	Binary *, /, %, <<, >>, ^, &, |
.	
Most binding, highest precedence	Unary -, +, ~

Note

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

2.7.3 Data Types

Each symbol you reference or define in an assembly program belongs to one of the type categories shown in Table 2-3.

Table 2-3: Data Types

Type	Description
undefined	Any symbol that is referenced but not defined becomes *global undefined*. (Declaring such a symbol in a `.globl` directive merely makes its status clearer.)
absolute	A constant defined in an ‘‘=’’ expression.
text	Any symbol defined while the `.text` directive is in effect belongs to the text section. The text section contains the program's instructions, which are not modifiable during execution.

Table 2-3: (continued)

Type	Description
data	Any symbol defined while the `.data` directive is in effect belongs to the data section. The data section contains memory that the linker can initialize to nonzero values before your program begins to execute.
sdata	This category is similar to `data`, except that defining a symbol while the `.sdata` (‘‘small data’’) directive is in effect causes the linker to place it within the small data section. This increases the chance that the linker will be able to optimize memory references to the item by using gp-relative addressing.
rdata	Any symbol defined while the `.rdata` directive is in effect belongs to this category, which is similar to `data`, but may not be modified during execution.
bss and sbss	Any symbol defined in a `.comm` or `.lcomm` directive belongs to these sections (except that a `.data`, `.sdata`, or `.rdata` directive can override a `.comm` directive). The bss and sbss sections consist of memory that the kernel loader initializes to zero before your program begins to execute. If a symbol’s size is less than the number of bytes specified by the `-G` compilation option (which defaults to eight), it belongs to sbss (small bss), and the linker places it within the small data section. This increases the chance that the linker will be able to optimize memory references to the item by using gp-relative addressing. Local symbols in bss or sbss defined by `.lcomm` directives are allocated memory by the assembler, global symbols are allocated memory by the linker, and symbols defined by `.comm` directives are overlaid upon like-named symbols (in the fashion of Fortran COMMON blocks) by the linker.

Symbols in the undefined category are always global; that is, they are visible to the linker and can be shared with other modules of your program. Symbols in the absolute, text, data, sdata, rdata, bss, and sbss type categories are local unless declared in a `.globl` directive.

2.7.4 Type Propagation in Expressions

For any expression, the result’s type depends on the types of the operands and on the operator. The following type propagation rules are used in expressions:

- If an operand is undefined, the result is undefined.

- If both operands are absolute, the result is absolute.
- If the operator is a plus sign (+) and the first operand refers to an undefined external symbol or a relocatable text-section, data-section, or bss-section symbol, the result has the first operand's type and the other operand must be absolute.
- If the operator is a minus sign (-) and the first operand refers to a relocatable text-section, data-section, or bss-section symbol, type propagation rules can vary:
  - The second operand can be absolute (if it was previously defined) and the result has the first operand's type.
  - 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.

2.8 Address Formats

The assembler accepts addresses expressed in the following formats:

Table 2-4: Address Formats

Format	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 the value at the specified address.
expression(*base-register*)	Specifies a based address. To get the address, the value of the expression is added to the contents of the base register. The assembler generates the most locally efficient code for referencing the value at the specified address.
relocatable-symbol	Specifies a relocatable address. The assembler generates the necessary instructions to address the item and generates relocation information for the linker.

Table 2-4: (continued)

Format	Address Description
relocatable-symbol ± *expression*	Specifies a relocatable address. To get the address, the value of the expression, which has an absolute value, is added or subtracted from the relocatable symbol. The assembler generates the necessary instructions to address the item and generates relocation information for the linker. 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 index register is added to the relocatable symbol's address. The assembler generates the necessary instructions to address the item and generates relocation information for the linker. If the symbol name does not appear as a label anywhere in the assembly, the assembler assumes that the symbol is external.
relocatable-symbol ± *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 index register. The assembler generates the necessary instructions to address the item and generates relocation 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.

Main Instruction Set 3

The assembler's instruction set consists of a main instruction set and a floating-point instruction set. This chapter describes the main instruction set; Chapter 4 describes the floating-point instruction set. For details on the instruction set beyond the scope of this manual, refer to the *Alpha Architecture Reference Manual*.

The assembler's main instruction set contains the following classes of instructions:

- Load and store instructions (Section 3.1) – these instructions load immediate values and move data between memory and general registers.
- Arithmetic instructions (Section 3.2) – these instructions perform arithmetic operations on values in registers.
- Logical and shift instructions (Section 3.3) – these instructions perform logical operations and shifts on values in registers.
- Relational instructions (Section 3.4) – these instructions compare values in registers.
- Move instructions (Section 3.5) – these instructions move data between registers.
- Control instructions (Section 3.6) – these instructions affect the sequence in which instructions are executed by transferring control from one location in a program to another.
- Byte-manipulation instructions (Section 3.7) – these instructions perform byte operations on values in registers.
- Special-purpose instructions (Section 3.8) – these instructions perform miscellaneous tasks.

Tables in this chapter show the format of each instruction in the main instruction set. The tables list the instruction names and the forms of operands that can be used with each instruction. The specifiers used in the tables to identify operands have the following meanings:

Operand Specifier	Description
address	A symbolic expression whose effective value is used as an address.

Operand Specifier	Description
b_reg	Base register. An integer register containing a base address to which is added an offset (or displacement) value to produce an effective address.
d_reg	Destination register. An integer register that receives a value as a result of an operation.
d_reg/s_reg	One integer register that is used as both a destination register and a source register.
label	A label that identifies a location in a program.
no_operands	No operands are specified.
offset	An immediate value that is added to the contents of a base register to calculate an effective address.
palcode	A value that determines the operation performed by a PAL instruction.
s_reg, s_reg1, s_reg2	Source registers. Registers whose contents are to be used in an operation.
val_expr	An expression whose value is used as an absolute value.
val_immed	An immediate value that is to be used in an operation.
jhint	An address operand that provides a hint of where a `jmp` or `jsr` instruction will transfer control.
rhint	An immediate operand that provides software with a hint about how a `ret` or `jsr_coroutine` instruction is used.

3.1 Load and Store Instructions

This section describes the general-purpose load and store instructions supported by the assembler.

Table 3-1 lists the mnemonics and operands for instructions that perform load and store operations. The table is segmented into groups of functionally related instructions. The operands specified within a particular segment of the table apply to all of the instructions contained in that segment.

Table 3-1: Load and Store Formats

Instruction	Mnemonic	Operands
Load Address	`lda`	*d_reg, address*
Load Byte	`ldb`	
Load Byte Unsigned	`ldbu`	

Table 3-1: (continued)

Instruction	Mnemonic	Operands
Load Word	`ldw`	
Load Word Unsigned	`ldwu`	
Load Sign Extended Longword	`ldl`	
Load Sign Extended Longword Locked	`ldl_l`	
Load Quadword	`ldq`	
Load Quadword Locked	`ldq_l`	
Load Quadword Unaligned	`ldq_u`	
Unaligned Load Word	`uldw`	
Unaligned Load Word Unsigned	`uldwu`	
Unaligned Load Longword	`uldl`	
Unaligned Load Quadword	`uldq`	
Load Address High	`ldah`	*d_reg, offset(b_reg)*
Load Global Pointer	`ldgp`	
Load Immediate Longword	`ldil`	*d_reg, val_expr*
Load Immediate Quadword	`ldiq`	
Store Byte	`stb`	*s_reg, address*
Store Word	`stw`	
Store Longword	`stl`	
Store Longword Conditional	`stl_c`	
Store Quadword	`stq`	
Store Quadword Conditional	`stq_c`	
Store Quadword Unaligned	`stq_u`	
Unaligned Store Word	`ustw`	
Unaligned Store Longword	`ustl`	
Unaligned Store Quadword	`ustq`	

Section 3.1.1 describes the operations performed by load instructions and Section 3.1.2 describes the operations performed by store instructions.

3.1.1 Load Instruction Descriptions

Load instructions move values (addresses, values of expressions, or contents of memory locations) into registers. For all load instructions, the effective address is the 64-bit twos-complement sum of the contents of the index register and the sign-extended offset.

Instructions whose address operands contain symbolic labels imply an index register, which the assembler determines. Some assembler load instructions can produce multiple machine-code instructions (see Section C.4).

Note

Load instructions can generate many code sequences for which the linker must fix the address by resolving external data items.

Table 3-2 describes the operations performed by load instructions.

Table 3-2: Load Instruction Descriptions

Instruction	Description
Load Address (`lda`)	Loads the destination register with the effective address of the specified data item.
Load Byte (`ldb`)	Loads the least significant byte of the destination register with the contents of the byte specified by the effective address. Because the loaded byte is a signed value, its sign bit is replicated to fill the other bytes in the destination register. (The assembler uses temporary registers `AT` and `t9` for this instruction.)
Load Byte Unsigned (`ldbu`)	Loads the least significant byte of the destination register with the contents of the byte specified by the effective address. Because the loaded byte is an unsigned value, the other bytes of the destination register are cleared to zeros. (The assembler uses temporary registers `AT` and `t9` for this instruction.)
Load Word (`ldw`)	Loads the two least significant bytes of the destination register with the contents of the word specified by the effective address. Because the loaded word is a signed value, its sign bit is replicated to fill the other bytes in the destination register. If the effective address is not evenly divisible by two, a data alignment exception may be signaled. (The assembler uses temporary registers `AT` and `t9` for this instruction.)
Load Word Unsigned (`ldwu`)	Loads the two least significant bytes of the destination register with the contents of the word specified by the effective address. Because the loaded word is an unsigned value, the other bytes of the destination register are cleared to zeros.

Table 3-2: (continued)

Instruction	Description
	If the effective address is not evenly divisible by two, a data alignment exception may be signaled. (The assembler uses temporary registers `AT` and `t9` for this instruction.)
Load Sign Extended Longword (`ldl`)	Loads the four least significant bytes of the destination register with the contents of the longword specified by the effective address. Because the loaded longword is a signed value, its sign bit is replicated to fill the other bytes in the destination register. If the effective address is not evenly divisible by four, a data alignment exception is signaled.
Load Sign Extended Longword Locked (`ldl_l`)	Loads the four least significant bytes of the destination register with the contents of the longword specified by the effective address. Because the loaded longword is a signed value, its sign bit is replicated to fill the other bytes in the destination register. If the effective address is not evenly divisible by four, a data alignment exception is signaled. If a `ldl_l` instruction executes without faulting, the processor records the target physical address in a per-processor locked-physical-address register and sets the per-processor lock flag. If the per-processor lock flag is still set when a `stl_c` instruction is executed, the store occurs; otherwise, it does not occur.
Load Quadword (`ldq`)	Loads the destination register with the contents of the quadword specified by the effective address. All bytes of the register are replaced with the contents of the loaded quadword. If the effective address is not evenly divisible by eight, a data alignment exception is signaled.

Table 3-2: (continued)

Instruction	Description
Load Quadword Locked (ldq_l)	Loads the destination register with the contents of the quadword specified by the effective address. All bytes of the register are replaced with the contents of the loaded quadword. If the effective address is not evenly divisible by eight, a data alignment exception is signaled. If an ldq_l instruction executes without faulting, the processor records the target physical address in a per-processor locked-physical-address register and sets the per-processor lock flag. If the per-processor lock flag is still set when a stq_c instruction is executed, the store occurs; otherwise, it does not occur.
Load Quadword Unaligned (ldq_u)	Loads the destination register with the contents of the quadword specified by the effective address (with the three low-order bits cleared). The address does not have to be aligned on an 8-byte boundary; it can be any byte address.
Unaligned Load Word (uldw)	Loads the two least significant bytes of the destination register with the word at the specified address. The address does not have to be aligned on a 2-byte boundary; it can be any byte address. Because the loaded word is a signed value, its sign bit is replicated to fill the other bytes in the destination register. (The assembler uses temporary registers AT, t9, and t10 for this instruction.)
Unaligned Load Word Unsigned (uldwu)	Loads the two least significant bytes of the destination register with the word at the specified address. The address does not have to be aligned on a 2-byte boundary; it can be any byte address. Because the loaded word is an unsigned value, the other bytes of the destination register are cleared to zeros. (The assembler uses temporary registers AT, t9, and t10 for this instruction.)

Table 3-2: (continued)

Instruction	Description
Unaligned Load Longword (`uldl`)	Loads the four least significant bytes of the destination register with the longword at the specified address. The address does not have to be aligned on a 4-byte boundary; it can be any byte address in memory. (The assembler uses temporary registers `AT`, `t9`, and `t10` for this instruction.)
Unaligned Load Quadword (`uldq`)	Loads the destination register with the quadword at the specified address. The address does not have to be aligned on an 8-byte boundary; it can be any byte address in memory. (The assembler uses temporary registers `AT`, `t9`, and `t10` for this instruction.)
Load Address High (`ldah`)	Loads the destination register with the effective address of the specified data item. In computing the effective address, the signed constant offset is multiplied by 65536 before adding to the base register. The signed constant must be in the range -32768 to 32767.
Load Global Pointer (`ldgp`)	Loads the destination register with the global pointer value for the procedure. The sum of the base register and the sign-extended offset specifies the address of the `ldgp` instruction.
Load Immediate Longword (`ldil`)	Loads the destination register with the value of an expression that can be computed at assembly time. The value is converted to canonical longword form before being stored in the destination register; bit 31 is replicated in bits 32 though 63 of the destination register. (See Appendix B for additional information on canonical forms.)
Load Immediate Quadword (`ldiq`)	Loads the destination register with the value of an expression that can be computed at assembly time.

3.1.2 Store Instruction Descriptions

For all store instructions, the effective address is the 64-bit twos-complement sum of the contents of the index register and the sign-extended 16-bit offset.

Instructions whose address operands contain symbolic labels imply an index register, which the assembler determines. Some assembler store instructions can produce multiple machine-code instructions (see Section C.4).

Table 3-3 describes the operations performed by store instructions.

Table 3-3: Store Instruction Descriptions

Instruction	Description
Store Byte (`stb`)	Stores the least significant byte of the source register in the memory location specified by the effective address. (The assembler uses temporary registers `AT`, `t9`, and `t10` for this instruction.)
Store Word (`stw`)	Stores the two least significant bytes of the source register in the memory location specified by the effective address. If the effective address is not evenly divisible by two, a data alignment exception may be signaled. (The assembler uses temporary registers `AT`, `t9`, and `t10` for this instruction.)
Store Longword (`stl`)	Stores the four least significant bytes of the source register in the memory location specified by the effective address. If the effective address is not evenly divisible by four, a data alignment exception is signaled.
Store Longword Conditional (`stl_c`)	Stores the four least significant bytes of the source register in the memory location specified by the effective address, if the lock flag is set. The lock flag is returned in the source register and is then set to zero. If the effective address is not evenly divisible by four, a data alignment exception is signaled.
Store Quadword (`stq`)	Stores the contents of the source register in the memory location specified by the effective address. If the effective address is not evenly divisible by eight, a data alignment exception is signaled.

Table 3-3: (continued)

Instruction	Description
Store Quadword Conditional (`stq_c`)	Stores the contents of the source register in the memory location specified by the effective address, if the lock flag is set. The lock flag is returned in the source register and is then set to zero. If the effective address is not evenly divisible by eight, a data alignment exception is signaled.
Store Quadword Unaligned (`stq_u`)	Stores the contents of the source register in the memory location specified by the effective address (with the three low-order bits cleared).
Unaligned Store Word (`ustw`)	Stores the two least significant bytes of the source register in the memory location specified by the effective address. The address does not have to be aligned on a 2-byte boundary; it can be any byte address. (The assembler uses temporary registers `AT`, `t9`, `t10`, `t11`, and `t12` for this instruction.)
Unaligned Store Longword (`ustl`)	Stores the four least significant bytes of the source register in the memory location specified by the effective address. The address does not have to be aligned on a 4-byte boundary; it can be any byte address. (The assembler uses temporary registers `AT`, `t9`, `t10`, `t11`, and `t12` for this instruction.)
Unaligned Store Quadword (`ustq`)	Stores the contents of the source register in a memory location specified by the effective address. The address does not have to be aligned on an 8-byte boundary; it can be any byte address. (The assembler uses temporary registers `AT`, `t9`, `t10`, `t11`, and `t12` for this instruction.)

3.2 Arithmetic Instructions

This section describes the general-purpose arithmetic instructions supported by the assembler. (Floating-point arithmetic instructions are described in Section 4.3.)

Table 3-4 lists the mnemonics and operands for instructions that perform arithmetic operations. The table is segmented into groups of functionally related instructions. The operands specified within a particular segment of the table apply to all of the instructions contained in that segment.

Table 3-4: Arithmetic Instruction Formats

Instruction	Mnemonic	Operands
Clear	`clr`	*d_reg*
Absolute Value Longword	`absl`	{ *s_reg, d_reg* / *d_reg /s_reg* / *val_immed, d_reg* }
Absolute Value Quadword	`absq`	
Negate Longword (without overflow)	`negl`	
Negate Longword (with overflow)	`neglv`	
Negate Quadword (without overflow)	`negq`	
Negate Quadword (with overflow)	`negqv`	
Sign-Extension Longword	`sextl`	
Add Longword (without overflow)	`addl`	{ *s_reg1, s_reg2, d_reg* / *d_reg/s_reg1, s_reg2* / *s_reg1, val_immed, d_reg* / *d_reg/s_reg1, val_immed* }
Add Longword (with overflow)	`addlv`	
Add Quadword (without overflow)	`addq`	
Add Quadword (with overflow)	`addqv`	
Scaled Longword Add by 4	`s4addl`	
Scaled Quadword Add by 4	`s4addq`	
Scaled Longword Add by 8	`s8addl`	
Scaled Quadword Add by 8	`s8addq`	
Multiply Longword (without overflow)	`mull`	
Multiply Longword (with overflow)	`mullv`	
Multiply Quadword (without overflow)	`mulq`	
Multiply Quadword (with overflow)	`mulqv`	
Subtract Longword (without overflow)	`subl`	
Subtract Longword (with overflow)	`sublv`	
Subtract Quadword (without overflow)	`subq`	
Subtract Quadword (with overflow)	`subqv`	
Scaled Longword Subtract by 4	`s4subl`	
Scaled Quadword Subtract by 4	`s4subq`	
Scaled Longword Subtract by 8	`s8subl`	
Scaled Quadword Subtract by 8	`s8subq`	
Unsigned Quadword Multiply High	`umulh`	
Divide Longword	`divl`	
Divide Longword Unsigned	`divlu`	
Divide Quadword	`divq`	
Divide Quadword Unsigned	`divqu`	
Longword Remainder	`reml`	
Longword Remainder Unsigned	`remlu`	
Quadword Remainder	`remq`	
Quadword Remainder Unsigned	`remqu`	

Table 3-5 describes the operations performed by arithmetic instructions.

Table 3-5: Arithmetic Instruction Descriptions

Instruction	Description
Clear (`clr`)	Sets the contents of the destination register to zero.
Absolute Value Longword (`absl`)	Computes the absolute value of the contents of the source register and puts the result in the destination register. If the value in the source register is -2147483648, an overflow exception is signaled.
Absolute Value Quadword (`absq`)	Computes the absolute value of the contents of the source register and puts the result in the destination register. If the value in the source register is -9223372036854775808, an overflow exception is signaled.
Negate Longword (without overflow) (`negl`)	Negates the integer contents of the four least significant bytes in the source register and puts the result in the destination register. An overflow occurs if the value in the source register is -2147483648, but the overflow exception is not signaled.
Negate Longword (with overflow) (`neglv`)	Negates the integer contents of the four least significant bytes in the source register and puts the result in the destination register. If the value in the source register is -2147483648, an overflow exception is signaled.
Negate Quadword (without overflow) (`negq`)	Negates the integer contents of the source register and puts the result in the destination register. An overflow occurs if the value in the source register is -2147483648, but the overflow exception is not signaled.
Negate Quadword (with overflow) (`negqv`)	Negates the integer contents of the source register and puts the result in the destination register. If the value in the source register is -9223372036854775808, an overflow exception is signaled.

Table 3-5: (continued)

Instruction	Description
Sign-Extension Longword (`sextl`)	Moves the four least significant bytes of the source register into the four least significant bytes of the destination register. Because the moved longword is a signed value, its sign bit is replicated to fill the other bytes in the destination register.
Add Longword (without overflow) (`addl`)	Computes the sum of two signed 32-bit values. This instruction adds the contents of s_reg1 to the contents of s_reg2 or the immediate value and then puts the result in the destination register. Overflow exceptions never occur.
Add Longword (with overflow) (`addlv`)	Computes the sum of two signed 32-bit values. This instruction adds the contents of *s_reg1* to the contents of *s_reg2* or the immediate value and then puts the result in the destination register. If the result cannot be represented as a signed 32-bit number, an overflow exception is signaled.
Add Quadword (without overflow) (`addq`)	Computes the sum of two signed 64-bit values. This instruction adds the contents of *s_reg1* to the contents of *s_reg2* or the immediate value and then puts the result in the destination register. Overflow exceptions never occur.
Add Quadword (with overflow) (`addqv`)	Computes the sum of two signed 64-bit values. This instruction adds the contents of *s_reg1* to the contents of *s_reg2* or the immediate value and then puts the result in the destination register. If the result cannot be represented as a signed 64-bit number, an overflow exception is signaled.
Scaled Longword Add by 4 (`s4addl`)	Computes the sum of two signed 32-bit values. This instruction scales (multiplies) the contents of *s_reg1* by four and then adds the contents of *s_reg2* or the immediate value. The result is stored in the destination register. Overflow exceptions never occur.

Table 3-5: (continued)

Instruction	Description
Scaled Quadword Add by 4 (`s4addq`)	Computes the sum of two signed 64-bit values. This instruction scales (multiplies) the contents of *s_reg1* by four and then adds the contents of *s_reg2* or the immediate value. The result is stored in the destination register. Overflow exceptions never occur.
Scaled Longword Add by 8 (`s8addl`)	Computes the sum of two signed 32-bit values. This instruction scales (multiplies) the contents of *s_reg1* by eight and then adds the contents of *s_reg2* or the immediate value. The result is stored in the destination register. Overflow exceptions never occur.
Scaled Quadword Add by 8 (`s8addq`)	Computes the sum of two signed 64-bit values. This instruction scales (multiplies) the contents of *s_reg1* by eight and then adds the contents of *s_reg2* or the immediate value. The result is stored in the destination register. Overflow exceptions never occur.
Multiply Longword (without overflow) (`mull`)	Computes the product of two signed 32-bit values. This instruction puts the 32-bit product of *s_reg1* and *s_reg2* or the immediate value in the destination register. Overflows are not reported.
Multiply Longword (with overflow) (`mullv`)	Computes the product of two signed 32-bit values. This instruction puts the 32-bit product of *s_reg1* and *s_reg2* or the immediate value in the destination register. If an overflow occurs, an overflow exception is signaled.
Multiply Quadword (without overflow) (`mulq`)	Computes the product of two signed 64-bit values. This instruction puts the 64-bit product of *s_reg1* and *s_reg2* or the immediate value in the destination register. Overflow is not reported.

Table 3-5: (continued)

Instruction	Description
Multiply Quadword (with overflow) (`mulqv`)	Computes the product of two signed 64-bit values. This instruction puts the 64-bit product of *`s_reg1`* and *`s_reg2`* or the immediate value in the destination register. If an overflow occurs, an overflow exception is signaled.
Subtract Longword (without overflow) (`subl`)	Computes the difference of two signed 32-bit values. This instruction subtracts either the contents of *`s_reg2`* or an immediate value from the contents of *`s_reg1`* and then puts the result in the destination register. Overflow exceptions never happen.
Subtract Longword (with overflow) (`sublv`)	Computes the difference of two signed 32-bit values. This instruction subtracts either the contents of *`s_reg2`* or an immediate value from the contents of *`s_reg1`* and then puts the result in the destination register. If the true result's sign differs from the destination register's sign, an overflow exception is signaled.
Subtract Quadword (without overflow) (`subq`)	Computes the difference of two signed 64-bit values. This instruction subtracts the contents of *`s_reg2`* or an immediate value from the contents of *`s_reg1`* and then puts the result in the destination register. Overflow exceptions never occur.
Subtract Quadword (with overflow) (`subqv`)	Computes the difference of two signed 64-bit values. This instruction subtracts the contents of *`s_reg2`* or an immediate value from the contents of *`s_reg1`* and then puts the result in the destination register. If the true result's sign differs from the destination register's sign, an overflow exception is signaled.

Table 3-5: (continued)

Instruction	Description
Scaled Longword Subtract by 4 (`s4subl`)	Computes the difference of two signed 32-bit values. This instruction subtracts the contents of *s_reg2* or the immediate value from the scaled (by 4) contents of *s_reg1*. The result is stored in the destination register. Overflow exceptions never occur.
Scaled Quadword Subtract by 4 (`s4subq`)	Computes the difference of two signed 64-bit values. This instruction subtracts the contents of *s_reg2* or the immediate value from the scaled (by 4) contents of *s_reg1*. The result is stored in the destination register. Overflow exceptions never occur.
Scaled Longword Subtract by 8 (`s8subl`)	Computes the difference of two signed 32-bit values. This instruction subtracts the contents of *s_reg2* or the immediate value from the scaled (by 8) contents of *s_reg1*. The result is stored in the destination register. Overflow exceptions never occur.
Scaled Quadword Subtract by 8 (`s8subq`)	Computes the difference of two signed 64-bit values. This instruction subtracts the contents of *s_reg2* or the immediate value from the scaled (by 8) contents of *s_reg1*. The result is stored in the destination register. Overflow exceptions never occur.
Unsigned Quadword Multiply High (`umulh`)	Computes the product of two unsigned 64-bit values. This instruction multiplies the contents of *s_reg1* by the contents of *s_reg2* or the immediate value and then puts the high-order 64 bits of the 128-bit product in the destination register.

Table 3-5: (continued)

Instruction	Description
Divide Longword (`divl`)	Computes the quotient of two signed 32-bit values. This instruction divides the contents of *s_reg1* by the contents of *s_reg2* or the immediate value and then puts the quotient in the destination register. The `divl` instruction rounds toward zero. If the divisor is zero, an error is signaled and a `call_pal PAL_gentrap` instruction may be issued. Overflow is signaled when dividing -2147483648 by -1. A `call_pal PAL_gentrap` instruction may be issued for either divide-by-zero or overflow.
Divide Longword Unsigned (`divlu`)	Computes the quotient of two unsigned 32-bit values. This instruction divides the contents of *s_reg1* by the contents of *s_reg2* or the immediate value and then puts the quotient in the destination register. If the divisor is zero, an exception is signaled and a `call_pal PAL_gentrap` instruction may be issued. Overflow exceptions never occur. (The assembler uses temporary registers `AT`, `t9`, `t10`, `t11`, and `t12` for this instruction.)
Divide Quadword (`divq`)	Computes the quotient of two signed 64-bit values. This instruction divides the contents of *s_reg1* by the contents of *s_reg2* or the immediate value and then puts the quotient in the destination register. The `divq` instruction rounds toward zero. If the divisor is zero, an error is signaled and a `call_pal PAL_gentrap` instruction may be issued. Overflow is signaled when dividing -9223372036854775808 by -1. A `call_pal PAL_gentrap` instruction may be issued for either divide-by-zero or overflow. (The assembler uses temporary registers `AT`, `t9`, `t10`, `t11`, and `t12` for this instruction.)

Table 3-5: (continued)

Instruction	Description
Divide Quadword Unsigned (`divqu`)	Computes the quotient of two unsigned 64-bit values. This instruction divides the contents of *`s_reg1`* by the contents of *`s_reg2`* or the immediate value and then puts the quotient in the destination register. If the divisor is zero, an exception is signaled and a `call_pal PAL_gentrap` instruction may be issued. Overflow exceptions never occur. (The assembler uses temporary registers `AT`, `t9`, `t10`, `t11`, and `t12` for this instruction.)
Longword Remainder (`reml`)	Computes the remainder of the division of two signed 32-bit values. The remainder `reml(i,j)` is defined as `i-(j*divl(i,j))` where j != 0. This instruction divides the contents of *`s_reg1`* by the contents of *`s_reg2`* or by the immediate value and then puts the remainder in the destination register. The `reml` instruction rounds toward zero. For example, `divl(5,-3)=-1`, and `reml(5,-3)=2`. For divide-by-zero, an error is signaled and a `call_pal PAL_gentrap` instruction may be issued. (The assembler uses temporary registers `AT`, `t9`, `t10`, `t11`, and `t12` for this instruction.)
Longword Remainder Unsigned (`remlu`)	Computes the remainder of the division of two unsigned 32-bit values. The remainder `remlu(i,j)` is defined as `i-(j*divlu(i,j))` where j != 0. This instruction divides the contents of *`s_reg1`* by the contents of *`s_reg2`* or the immediate value and then puts the remainder in the destination register. For divide-by-zero, an error is signaled and a `call_pal PAL_gentrap` instruction may be issued. (The assembler uses temporary registers `AT`, `t9`, `t10`, `t11`, and `t12` for this instruction.)

Table 3-5: (continued)

Instruction	Description
Quadword Remainder (`remq`)	Computes the remainder of the division of two signed 64-bit values. The remainder `remq(i,j)` is defined as `i-(j*divq(i,j))` where j != 0. This instruction divides the contents of *`s_reg1`* by the contents of *`s_reg2`* or the immediate value and then puts the remainder in the destination register. The `remq` instruction rounds toward zero. For example, `divq(5,-3)=-1`, and `remq(5,-3)=2`. For divide-by-zero, an error is signaled and a `call_pal PAL_gentrap` instruction may be issued. (The assembler uses temporary registers `AT`, `t9`, `t10`, `t11`, and `t12` for this instruction.)
Quadword Remainder Unsigned (`remqu`)	Computes the remainder of the division of two unsigned 64-bit values. The remainder `remqu(i,j)` is defined as `i-(j*divqu(i,j))` where j != 0. This instruction divides the contents of *`s_reg1`* by the contents of *`s_reg2`* or the immediate value and then puts the remainder in the destination register. For divide-by-zero, an error is signaled and a `call_pal PAL_gentrap` instruction may be issued. (The assembler uses temporary registers `AT`, `t9`, `t10`, `t11`, and `t12` for this instruction.)

3.3 Logical and Shift Instructions

This section describes the logical and shift instructions supported by the assembler.

Table 3-6 lists the mnemonics and operands for instructions that perform logical and shift operations. Each of the instructions listed in the table can take an operand in any of the forms shown in the table.

Table 3-6: Logical and Shift Instruction Formats

Instruction	Mnemonic	Operands
Logical Complement (NOT)	`not`	*s_reg, d_reg* *d_reg/s_reg* *val_immed, d_reg* *s_reg1, s_reg2, d_reg* *d_reg/s_reg1, s_reg2* *s_reg1, val_immed, d_reg* *d_reg/s_reg1, val_immed*
Logical Product (AND)	`and`	
Logical Sum (OR)	`bis`	
Logical Sum (OR)	`or`	
Logical Difference (XOR)	`xor`	
Logical Product with Complement (ANDNOT)	`bic`	
Logical Product with Complement (ANDNOT)	`andnot`	
Logical Sum with Complement (ORNOT)	`ornot`	
Logical Equivalence (XORNOT)	`eqv`	
Logical Equivalence (XORNOT)	`xornot`	
Shift Left Logical	`sll`	
Shift Right Logical	`srl`	
Shift Right Arithmetic	`sra`	

Table 3-7 describes the operations performed by logical and shift instructions.

Table 3-7: Logical and Shift Instruction Descriptions

Instruction	Description
Logical Complement (NOT) (`not`)	Computes the Logical NOT of a value. This instruction complements (bit-wise) the contents of *`s_reg1`* and puts the result in the destination register.
Logical Product (AND) (`and`)	Computes the Logical AND of two values. This instruction ANDs (bit-wise) the contents of *`s_reg1`* with the contents of *`s_reg2`* or the immediate value and then puts the result in the destination register.
Logical Sum (OR) (`bis`)	Computes the Logical OR of two values. This instruction ORs (bit-wise) the contents of *`s_reg1`* with the contents of *`s_reg2`* or the immediate value and then puts the result in the destination register.

Table 3-7: (continued)

Instruction	Description
Logical Sum (OR) (`or`)	Synonym for `bis`.
Logical Difference (XOR) (`xor`)	Computes the XOR of two values. This instruction XORs (bit-wise) the contents of *s_reg1* with the contents of *s_reg2* or the immediate value and then puts the result in the destination register.
Logical Product with Complement (ANDNOT) (`bic`)	Computes the Logical AND of two values. This instruction ANDs (bit-wise) the contents of *s_reg1* with the ones complement of *s_reg2* or the immediate value and then puts the result in the destination register.
Logical Product with Complement (ANDNOT) (`andnot`)	Synonym for `bic`.
Logical Sum with Complement (ORNOT) (`ornot`)	Computes the logical OR of two values. This instruction ORs (bit-wise) the contents of *s_reg1* with the ones complement of *s_reg2* or an immediate value and then puts the result in the destination register.
Logical Equivalence (XORNOT) (`eqv`)	Computes the logical XOR of two values. This instruction XORs (bit-wise) the contents of *s_reg1* with the ones complement of *s_reg2* or the immediate value and then puts the result in the destination register.
Logical Equivalence (XORNOT) (`xornot`)	Synonym for `eqv`.
Shift Left Logical (sll)	Shifts the contents of a register left (toward the sign bit) and inserts zeros in the vacated bit positions. The contents of *s_reg1* specifies the value to shift, and the contents of *s_reg2* or the immediate value specifies the amount to shift. If *s_reg2* or the immediate value is greater than 63 or less than zero, *s_reg1* shifts by the result of the following AND operation: *s_reg2* AND 63.

Table 3-7: (continued)

Instruction	Description
Shift Right Logical (`srl`)	Shifts the contents of a register right (toward the least significant bit) and inserts zeros in the vacated bit positions. The contents of *s_reg1* specifies the value to shift, and the contents of *s_reg2* or the immediate value specifies the amount to shift. If *s_reg2* or the immediate value is greater than 63 or less than zero, *s_reg1* shifts by the result of the result of the following AND operation: *s_reg2* AND 63.
Shift Right Arithmetic (`sra`)	Shifts the contents of a register right (toward the least significant bit) and inserts the sign bit in the vacated bit position. The contents of *s_reg1* specifies the value to shift, and the contents of *s_reg2* or the immediate value specifies the amount to shift. If *s_reg2* or the immediate value is greater than 63 or less than zero, *s_reg1* shifts by the result of the following AND operation: *s_reg2* AND 63.

3.4 Relational Instructions

This section describes the relational instructions supported by the assembler.

Table 3-8 lists the mnemonics and operands for instructions that perform relational operations. Each of the instructions listed in the table can take an operand in any of the forms shown in the table.

Table 3-8: Relational Instruction Formats

Instruction	Mnemonic	Operands
Compare Signed Quadword Equal	`cmpeq`	*s_reg1, s_reg2, d_reg* *d_reg/s_reg1, s_reg2* *s_reg1, val_immed, d_reg* *d_reg/s_reg1, val_immed*
Compare Signed Quadword Less Than	`cmplt`	
Compare Signed Quadword Less Than or Equal	`cmple`	
Compare Unsigned Quadword Less Than	`cmpult`	
Compare Unsigned Quadword Less Than or Equal	`cmpule`	

Table 3-9 describes the operations performed by relational instructions.

Table 3-9: Relational Instruction Descriptions

Instruction	Description
Compare Signed Quadword Equal (`cmpeq`)	Compares two 64-bit values. If the contents of *s_reg1* equals the contents of *s_reg2* or the immediate value, this instruction sets the destination register to one; otherwise, it sets the destination register to zero.
Compare Signed Quadword Less Than (`cmplt`)	Compares two signed 64-bit values. If the contents of *s_reg1* is less than the contents of s_reg2 or the immediate value, this instruction sets the destination register to one; otherwise, it sets the destination register to zero.
Compare Signed Quadword Less Than or Equal (`cmple`)	Compares two signed 64-bit values. If the contents of *s_reg1* is less than or equal to the contents of *s_reg2* or the immediate value, this instruction sets the destination register to one; otherwise, it sets the destination register to zero.
Compare Unsigned Quadword Less Than (`cmpult`)	Compares two unsigned 64-bit values. If the contents of *s_reg1* is less than the contents of *s_reg2* or the immediate value, this instruction sets the destination register to one; otherwise, it sets the destination register to zero.
Compare Unsigned Quadword Less Than or Equal (`cmpule`)	Compares two unsigned 64-bit values. If the contents of *s_reg1* is less than or equal to the contents of *s_reg2* or the immediate value, this instruction sets the destination register to one; otherwise, it sets the destination register to zero.

3.5 Move Instructions

This section describes the move instructions supported by the assembler.

Table 3-10 lists the mnemonics and operands for instructions that perform move operations. The table is segmented into groups of functionally related instructions. The operands specified within a particular segment of the table apply to all of the instructions contained in that segment.

Table 3-10: Move Instruction Formats

Instruction	Mnemonic	Operands
Move	`mov`	*s_reg, d_reg* *val_immed, d_reg*

Move if Equal to Zero
Move if Not Equal to Zero
Move if Less Than Zero
Move if Less Than or Equal to Zero
Move if Greater Than Zero
Move if Greater Than or Equal to Zero
Move if Low Bit Clear
Move if Low Bit Set
	`cmoveq`
	`cmovne`
	`cmovlt`
	`cmovle`
	`cmovgt`
	`cmovge`
	`cmovlbc`
	`cmovlbs`
		s_reg1, s_reg2, d_reg
		d_reg/s_reg1, s_reg2
		s_reg1, val_immed, d_reg
		d_reg/s_reg1, val_immed

Table 3-11 describes the operations performed by move instructions.

Table 3-11: Move Instruction Descriptions

Instruction	Description
Move (`mov`)	Moves the contents of the source register or the immediate value to the destination register.
Move if Equal to Zero (`cmoveq`)	Moves the contents of *`s_reg2`* or the immediate value to the destination register if the contents of *`s_reg1`* is equal to zero.
Move if Not Equal to Zero (`cmovne`)	Moves the contents of *`s_reg2`* or the immediate value to the destination register if the contents of *`s_reg1`* is not equal to zero.
Move if Less Than Zero (`cmovlt`)	Moves the contents of *`s_reg2`* or the immediate value to the destination register if the contents of *`s_reg1`* is less than zero.
Move if Less Than or Equal to Zero (`cmovle`)	Moves the contents of *`s_reg2`* or the immediate value to the destination register if the contents of *`s_reg1`* is less than or equal to zero.

Table 3-11: (continued)

Instruction	Description
Move if Greater Than Zero (`cmovgt`)	Moves the contents of *s_reg2* or the immediate value to the destination register if the contents of *s_reg1* is greater than zero.
Move if Greater Than or Equal to Zero (`cmovge`)	Moves the contents of *s_reg2* or the immediate value to the destination register if the contents of *s_reg1* is greater than or equal to zero.
Move if Low Bit Clear (`cmovlbc`)	Moves the contents of *s_reg2* or the immediate value to the destination register if the low-order bit of *s_reg1* is equal to zero.
Move if Low Bit Set (`cmovlbs`)	Moves the contents of *s_reg2* or the immediate value to the destination register if the low-order bit of *s_reg1* is not equal to zero.

3.6 Control Instructions

This section describes the control instructions supported by the assembler. Control instructions enable you to change the control flow of an assembly program.

Table 3-12 lists the mnemonics and operands for instructions that perform control operations. The table is segmented into groups of functionally related instructions. The operands specified within a particular segment of the table apply to all of the instructions contained in that segment.

Table 3-12: Control Instruction Formats

Instruction	Mnemonic	Operands
Branch if Equal to Zero	`beq`	*s_reg, label*
Branch if Not Equal to Zero	`bne`	
Branch if Less Than Zero	`blt`	
Branch if Less Than or Equal to Zero	`ble`	
Branch if Greater Than Zero	`bgt`	
Branch if Greater Than or Equal to Zero	`bge`	
Branch if Low Bit is Clear	`blbc`	

Table 3-12: (continued)

Instruction	Mnemonic	Operands
Branch if Low Bit is Set	`blbs`	
Branch Branch to Subroutine	`br` `bsr`	*d_reg, label* *label*
Jump Jump to Subroutine	`jmp` `jsr`	*d_reg, (s_reg), jhint* *d_reg, (s_reg)* *(s_reg), jhint* *(s_reg)* *d_reg, address* *address*
Return from Subroutine Jump to Subroutine Return	`ret` `jsr_coroutine`	*d_reg, (s_reg), rhint* *d_reg, (s_reg)* *d_reg, rhint* *d_reg* *(s_reg), rhint* *(s_reg)* *rhint* *no_operands*

Table 3-13 describes the operations performed by control instructions. For all branch instructions described in the table, the branch destinations must be defined in the source being assembled, not in an external source file.

Table 3-13: Control Instruction Descriptions

Instruction	Description
Branch if Equal to Zero (`beq`)	Branches to the specified label if the contents of the source register is equal to zero.
Branch if Not Equal to Zero (`bne`)	Branches to the specified label if the contents of the source register is not equal to zero.

Table 3-13: (continued)

Instruction	Description
Branch if Less Than Zero (`blt`)	Branches to the specified label if the contents of the source register is less than zero. The comparison treats the source register as a signed 64-bit value.
Branch if Less Than or Equal to Zero (`ble`)	Branches to the specified label if the contents of the source register is less than or equal to zero. The comparison treats the source register as a signed 64-bit value.
Branch if Greater Than Zero (`bgt`)	Branches to the specified label if the contents of the source register is greater than zero. The comparison treats the source register as a signed 64-bit value.
Branch if Greater Than or Equal to Zero (`bge`)	Branches to the specified label if the contents of the source register is greater than or equal to zero. The comparison treats the source register as a signed 64-bit value.
Branch if Low Bit is Clear (`blbc`)	Branches to the specified label if the low-order bit of the source register is equal to zero.
Branch if Low Bit is Set (`blbs`)	Branches to the specified label if the low-order bit of the source register is not equal to zero.
Branch (`br`)	Branches unconditionally to the specified label. If a destination register is specified, the address of the instruction following the `br` instruction is stored in that register.
Branch to Subroutine (`bsr`)	Branches unconditionally to the specified label and stores the return address in the destination register. If a destination register is not specified, register `$26` (`ra`) is used.
Jump (`jmp`)	Unconditionally jumps to a specified location. A symbolic address or the source register specifies the target location. If a destination register is specified, the address of the instruction following the `jmp` instruction is stored in the specified register.

Table 3-13: (continued)

Instruction	Description
Jump to Subroutine (`jsr`)	Unconditionally jumps to a specified location and stores the return address in the destination register. If a destination register is not specified, register `$26` (`ra`) is used. A symbolic address or the source register specifies the target location. The instruction `jsr` *`procname`* transfers to *`procname`* and saves the return address in register `$26`.
Return from Subroutine (`ret`)	Unconditionally returns from a subroutine. If a destination register is specified, the address of the instruction following the `ret` instruction is stored in the specified register. The source register contains the return address. If the source register is not specified, register `$26` (`ra`) is used. If a hint is not specified, a hint value of one is used.
Jump to Subroutine Return (`jsr_coroutine`)	Unconditionally returns from a subroutine and stores the return address in the destination register. If a destination register is not specified, register `$26` (`ra`) is used. The source register contains the target address. If the source register is not specified, register `$26` (`ra`) is used.

All jump instructions (`jmp, jsr, ret, jsr_coroutine`) perform identical operations. They differ only in hints to possible branch-prediction logic. See the *Alpha Architecture Reference Manual* for information about branch-prediction logic.

3.7 Byte-Manipulation Instructions

This section describes the byte-manipulation instructions supported by the assembler.

Table 3-14 lists the mnemonics and operands for instructions that perform byte-manipulation operations. Each of the instructions listed in the table can take an operand in any of the forms shown in the table.

Table 3-14: Byte-Manipulation Instruction Formats

Instruction	Mnemonic	Operands
Compare Byte	`cmpbge`	*s_reg1, s_reg2, d_reg* *d_reg/s_reg1, s_reg2* *s_reg1, val_immed, d_reg* *d_reg/s_reg1, val_immed*
Extract Byte Low	`extbl`	
Extract Word Low	`extwl`	
Extract Longword Low	`extll`	
Extract Quadword Low	`extql`	
Extract Word High	`extwh`	
Extract Longword High	`extlh`	
Extract Quadword High	`extqh`	
Insert Byte Low	`insbl`	
Insert Word Low	`inswl`	
Insert Longword Low	`insll`	
Insert Quadword Low	`insql`	
Insert Word High	`inswh`	
Insert Longword High	`inslh`	
Insert Quadword High	`insqh`	
Mask Byte Low	`mskbl`	
Mask Word Low	`mskwl`	
Mask Longword Low	`mskll`	
Mask Quadword Low	`mskql`	
Mask Word High	`mskwh`	
Mask Longword High	`msklh`	
Mask Quadword High	`mskqh`	
Zero Bytes	`zap`	
Zero Bytes NOT	`zapnot`	

Table 3-15 describes the operations performed by byte-manipulation instructions.

Table 3-15: Byte-Manipulation Instruction Descriptions

Instruction	Description
Compare Byte (`cmpbge`)	Performs eight parallel unsigned byte comparisons between corresponding bytes of register *s_reg1* and *s_reg2* or the immediate value. A bit is set in the destination register if a byte in *s_reg1* is greater than or equal to the corresponding byte in *s_reg2* or the immediate value. The results of the comparisons are stored in the eight low-order bits of the destination register; bit 0 of the destination register corresponds to byte 0 and so forth. (The 56 high-order bits of the destination register are cleared.)
Extract Byte Low (`extbl`)	Shifts the register *s_reg1* right by 0-7 bytes, inserts zeros into the vacated bit positions, and then extracts the low-order byte into the destination register. The seven high-order bytes of the destination register are cleared to zeros. Bits 0-2 of register *s_reg2* or the immediate value specify the shift count.
Extract Word Low (`extwl`)	Shifts the register *s_reg1* right by 0-7 bytes, inserts zeros into the vacated bit positions, and then extracts the two low-order bytes and stores them in the destination register. The six high-order bytes of the destination register are cleared to zeros. Bits 0-2 of register *s_reg2* or the immediate value specify the shift count.
Extract Longword Low (`extll`)	Shifts the register *s_reg1* right by 0-7 bytes, inserts zeros into the vacated bit positions, and then extracts the four low-order bytes and stores them in the destination register. The four high-order bytes of the destination register are cleared to zeros. Bits 0-2 of register *s_reg2* or the immediate value specify the shift count.
Extract Quadword Low (`extql`)	Shifts the register *s_reg1* right by 0-7 bytes, inserts zeros into the vacated bit positions, and then extracts all eight bytes and stores them in the destination register. Bits 0-2 of register *s_reg2* or the immediate value specify the shift count.

Table 3-15: (continued)

Instruction	Description
Extract Word High (`extwh`)	Shifts the register *s_reg1* left by 0-7 bytes, inserts zeros into the vacated bit positions, and then extracts the two low-order bytes and stores them in the destination register. The six high-order bytes of the destination register are cleared to zeros. Bits 0-2 of register *s_reg2* or the immediate value specify the shift count.
Extract Longword High (`extlh`)	Shifts the register *s_reg1* left by 0-7 bytes, inserts zeros into the vacated bit positions, and then extracts the four low-order bytes and stores them in the destination register. The four high-order bytes of the destination register are cleared to zeros. Bits 0-2 of register *s_reg2* or the immediate value specify the shift count.
Extract Quadword High (`extqh`)	Shifts the register *s_reg1* left by 0-7 bytes, inserts zeros into the vacated bit positions, and then extracts all eight bytes and stores them in the destination register. Bits 0-2 of register *s_reg2* or the immediate value specify the shift count.
Insert Byte Low (`insbl`)	Shifts the register *s_reg1* left by 0-7 bytes, inserts the byte into a field of zeros, and then puts the result in the destination register. Bits 0-2 of register *s_reg2* or the immediate value specify the shift count.
Insert Word Low (`inswl`)	Shifts the register *s_reg1* left by 0-7 bytes, inserts the word into a field of zeros, and then puts the result in the destination register. Bits 0-2 of register *s_reg2* or the immediate value specify the shift count.
Insert Longword Low (`insll`)	Shifts the register *s_reg1* left by 0-7 bytes, inserts the longword into a field of zeros, and then puts the result in the destination register. Bits 0-2 of register *s_reg2* or the immediate value specify the shift count.

Table 3-15: (continued)

Instruction	Description
Insert Quadword Low (`insql`)	Shifts the register *s_reg1* left by 0-7 bytes, inserts the quadword into a field of zeros, and then puts the result in the destination register. Bits 0-2 of register *s_reg2* or the immediate value specify the shift count.
Insert Quadword Low (`insql`)	Shifts the register *s_reg1* left by 0-7 bytes, inserts the quadword into a field of zeros, and then puts the result in the destination register. Bits 0-2 of register *s_reg2* or the immediate value specify the shift count.
Insert Word High (`inswh`)	Shifts the register *s_reg1* right by 0-7 bytes, inserts the word into a field of zeros, and then puts the result in the destination register. Bits 0-2 of register *s_reg2* or the immediate value specify the shift count.
Insert Longword High (`inslh`)	Shifts the register *s_reg1* right by 0-7 bytes, inserts the longword into a field of zeros, and then puts the result in the destination register. Bits 0-2 of register *s_reg2* or the immediate value specify the shift count.
Insert Quadword High (`insqh`)	Shifts the register *s_reg1* right by 0-7 bytes, inserts the quadword into a field of zeros, and then puts the result in the destination register. Bits 0-2 of register *s_reg2* or the immediate value specify the shift count.
Mask Byte Low (`mskbl`)	Sets a byte in register *s_reg1* to zero and stores the result in the destination register. Bits 0-2 of register *s_reg2* or the immediate value specify the offset of the byte.
Mask Word Low (`mskwl`)	Sets a word in register *s_reg1* to zero and stores the result in the destination register. Bits 0-2 of register *s_reg2* or the immediate value specify the offset of the word.
Mask Longword Low (`mskll`)	Sets a longword in register *s_reg1* to zero and stores the result in the destination register. Bits 0-2 of register *s_reg2* or the immediate value specify the offset of the longword.

Table 3-15: (continued)

Instruction	Description
Mask Quadword Low (`mskql`)	Sets a quadword in register *s_reg1* to zero and stores the result in the destination register. Bits 0-2 of register *s_reg2* or the immediate value specify the offset of the quadword.
Mask Word High (`mskwh`)	Sets a word in register *s_reg1* to zero and stores the result in the destination register. Bits 0-2 of register *s_reg2* or the immediate value specify the offset of the word.
Mask Longword High (`msklh`)	Sets a longword in register *s_reg1* to zero and stores the result in the destination register. Bits 0-2 of register *s_reg2* or the immediate value specify the offset of the longword.
Mask Quadword High (`mskqh`)	Sets a quadword in register *s_reg1* to zero and stores the result in the destination register. Bits 0-2 of register *s_reg2* or the immediate value specify the offset of the quadword.
Zero Bytes (`zap`)	Sets selected bytes of register *s_reg1* to zero and puts the result in the destination register. Bits 0-7 of register *s_reg2* or an immediate value specify the bytes to be cleared to zeros. Each bit corresponds to one byte in register *s_reg1*; for example, bit 0 corresponds to byte 0. A bit with a value of one indicates its corresponding byte should be cleared to zeros.
Zero Bytes NOT (`zapnot`)	Sets selected bytes of register *s_reg1* to zero and puts the result in the destination register. Bits 0-7 of register *s_reg2* or an immediate value specify the bytes to be cleared to zeros. Each bit corresponds to one byte in register *s_reg1*; for example, bit 0 corresponds to byte 0. A bit with a value of zero indicates its corresponding byte should be cleared to zeros.

3.8 Special-Purpose Instructions

This section describes the special-purpose instructions supported by the assembler.

Table 3-16 lists the mnemonics and operands for instructions that perform special operations. The table is segmented into groups of functionally related instructions. The operands specified within a particular segment of the table apply to all of the instructions contained in that segment.

Table 3-16: Special-Purpose Instruction Formats

Instruction	Mnemonic	Operands
Call Privileged Architecture Library	`call_pal`	*palcode*
Prefetch Data Prefetch Data, Modify Intent	`fetch` `fetch_m`	*address*
Read Process Cycle Counter	`rpcc`	*d_reg*

No Operation
Universal No Operation
Trap Barrier
Exception Barrier
Memory Barrier
Write Memory Barrier
	`nop`
	`unop`
	`trapb`
	`excb`
	`mb`
	`wmb`
		no_operands

Table 3-17 describes the operations performed by special-purpose instructions.

Table 3-17: Special-Purpose Instruction Descriptions

Instruction	Description
Call Privileged Architecture Library (`call_pal`)	Unconditionally transfers control to the exception handler. The *`palcode`* operand is interpreted by software conventions.
Prefetch Data (`fetch`)	Indicates that the 512-byte block of data specified by the effective address should be moved to a faster-access part of the memory hierarchy.

Table 3-17: (continued)

Instruction	Description
Prefetch Data, Modify Intent (`fetch_m`)	Indicates that the 512-byte block of data specified by the effective address should be moved to a faster-access part of the memory hierarchy. In addition, this instruction is a hint that part or all of the data may be modified.
Read Process Cycle Counter (`rpcc`)	Returns the contents of the process cycle counter in the destination register.
No Operation (`nop`)	Has no effect on the machine state.
Universal No Operation (`unop`)	Has no effect on the machine state.
Trap Barrier (`trapb`)	Guarantees that all previous arithmetic instructions complete, without incurring any arithmetic traps, before any instructions after the `trapb` instruction are issued.
Exception Barrier (`excb`)	Guarantees that all previous instructions complete any exception-related behavior or rounding-mode behavior before any instructions after the `excb` instruction are issued.
Memory Barrier (`mb`)	Used to serialize access to memory. See the *Alpha Architecture Reference Manual* for addition information on memory barriers.
Write Memory Barrier (`wmb`)	Guarantees that all previous store instructions access memory before any store instructions issued after the `wmb` instruction.

Floating-Point Instruction Set 4

This chapter describes the assembler's floating-point instructions. See Chapter 3 for a description of the integer instructions. For details on the instruction set beyond the scope of this manual, refer to the *Alpha Architecture Reference Manual*.

The assembler's floating-point instruction set contains the following classes of instructions:

- Load and store instructions (Section 4.2) – these instructions load values and move data between memory and floating-point registers.
- Arithmetic instructions. (Section 4.3) – these instructions perform arithmetic and logical operations on values in floating-point registers.
- Relational instructions (Section 4.4) – these instructions compare two floating-point values.
- Move instructions (Section 4.5) – these instructions move data between floating-point registers.
- Control instructions (Section 4.6) – these instructions test floating-point registers and conditionally branch.
- Special-purpose instructions (Section 4.7) – these instructions perform miscellaneous tasks.

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

Tables in this chapter show the format for each instruction in the floating-point instruction set. The tables list the instruction names and the forms of operands that can be used with each instruction. The specifiers used in the tables to identify operands have the following meanings:

Operand Specifier	Description
address	A symbolic expression whose effective value is used as an address.
d_reg	Destination register. A floating-point register that receives a value as a result of an operation.
d_reg/s_reg	One floating-point register that is used as both a destination register and a source register.

Operand Specifier	Description
label	A label that identifies a location in a program
s_reg, s_reg1, s_reg2	Source registers. Floating-point registers whose contents are to be used in an operation.
val_expr	An expression whose value is a floating-point constant.

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

Term	Meaning
Infinite	A value of +∞ or -∞.
Infinity	A symbolic entity that represents values with magnitudes greater than the largest magnitude for a particular format.
Ordered	The usual result from a comparison, namely: less than (<), equal (=), or greater than (>).
NaN	Symbolic entities that represent values not otherwise available in floating-point formats. (NaN is an acronym for not-a-number.)
Unordered	The condition that results from a floating-point comparison when one or both operands are NaNs.

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.

4.1 Background Information on Floating-Point Operations

The following sections contain information on these topics:

- Floating-point data types (Section 4.1.1)
- The floating-point control register (Section 4.1.2)
- Floating-point exceptions (Section 4.1.3)
- Floating-point rounding modes (Section 4.1.4)
- Floating-point instruction qualifiers (Section 4.1.5)

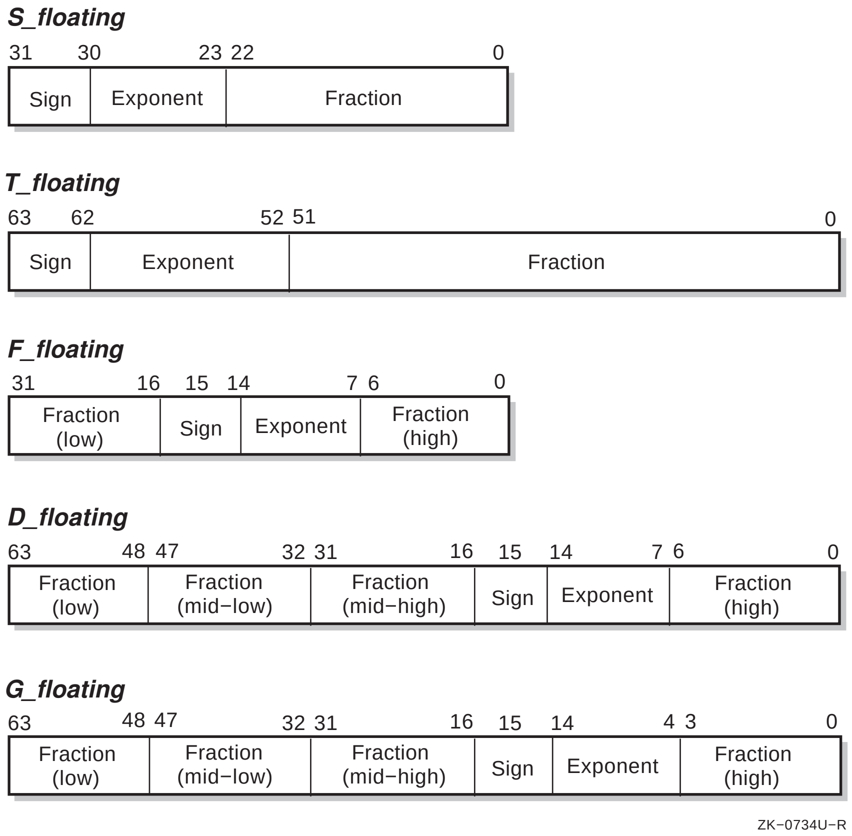
4.1.1 Floating-Point Data Types

Floating-point instructions operate on the following data types:

- D_floating (VAX double precision, limited support)
- F_floating (VAX single precision)
- G_floating (VAX double precision)
- S_floating (IEEE single precision)
- T_floating (IEEE double precision)
- Longword integer and quadword integer

Figure 4-1 shows the memory formats for the single and double precision floating-point data types.

Figure 4-1: Floating-Point Data Formats



4.1.2 Floating-Point Control Register

The floating-point control register (FPCR) contains status and control information. It controls the arithmetic rounding mode of instructions that specify dynamic rounding (`d` qualifier) and gives a summary for each exception type of the exception conditions detected by the floating-point instructions. It also contains an overall summary bit indicating whether an exception occurred.

Figure 4-2 shows the format of the floating-point control register.

Figure 4-2: Floating-Point Control Register

63	62 60	59 58	57	56	55	54	53	52	51 0
sum	raz/ ign	dyn	iov	ine	unf	ovf	dze	inv	raz/ign

ZK−0735U−R

The fields of the floating-point control register have the following meaning:

Bits	Name	Description
63	`sum`	Summary – records the bitwise OR of the FPCR exception bits (bits 57 to 52).
62-60	`raz/ign`	Read-As-Zero – ignored when written.
59-58	`dyn`	Dynamic Rounding Mode – indicates the current rounding mode to be used by an IEEE floating-point instruction that specifies dynamic mode (`d` qualifier). The bit assignments for this field are as follows: 00 – Chopped rounding mode 01 – Minus infinity 10 – Normal rounding 11 – Plus infinity
57	`iov`	Integer overflow.
56	`ine`	Inexact result.
55	`unf`	Underflow.
54	`ovf`	Overflow.
53	`dze`	Division by zero.
52	`inv`	Invalid operation.

Bits	Name	Description
51-0	`raz/ign`	Read-As-Zero – ignored when written.

The floating-point exceptions associated with bits 57 to 52 are described in Section 4.1.3.

4.1.3 Floating-Point Exceptions

Six exception conditions can result from the use of floating-point instructions. All of the exceptions are signaled by an arithmetic exception trap. The exceptions are as follows:

- Invalid Operation – An invalid operation exception is signaled if any operand of a floating-point instruction, other than `cmpt`*`xx`*, is non-finite. (`cmpt`*`xx`* operates normally with plus and minus infinity.) This trap is always enabled. If this trap occurs, an unpredictable value is stored in the destination register.
- Division by Zero – A division by zero exception is taken if the numerator does not cause an invalid operation trap and the denominator is zero. This trap is always enabled. If this trap occurs, an unpredictable value is stored in the destination register.
- Overflow – An overflow exception is signaled if the rounded result exceeds, in magnitude, the largest finite number of the destination format. This trap is always enabled. If this trap occurs, an unpredictable value is stored in the destination register.
- Underflow – An underflow exception occurs if the rounded result is smaller, in magnitude, than the smallest finite number of the destination format. This trap may be disabled. If this trap occurs, a true zero is always stored in the destination register.
- Inexact Result – An inexact result exception occurs if the infinitely precise result differs from the rounded result. This trap may be disabled. If this trap occurs, the normal rounded result is still stored in the destination register.
- Integer Overflow – An integer overflow exception occurs if the conversion from a floating-point or integer format to an integer format results in a value that is outside of the range of values representable by the destination format. This trap may be disabled. If this trap occurs, the true result is truncated to the number of bits in the destination format and stored in the destination register.

4.1.4 Floating-Point Rounding Modes

If a true result can be exactly represented in a floating-point format, all rounding modes map the true result to that value.

The following abbreviations are used in the descriptions of rounding modes provided in this section:

- LSB (least significant bit) – For a positive representable number A whose fraction is not all ones, A + 1 LSB is the next larger representable number, and A + 1/2 LSB is exactly halfway between A and the next larger representable number.
- MAX – The largest non-infinite representable floating-point number.
- MIN – The smallest nonzero representable normalized floating-point number.

For VAX floating-point operations, two rounding modes are provided and are specified in each instruction:

- Normal rounding (biased)
  - Maps the true result to the nearest of two representable results, with true results exactly halfway between mapped to the larger in absolute value (sometimes referred to as biased rounding away from zero)
  - Maps true results ≥ MAX + 1/2 LSB in magnitude to an overflow
  - Maps true results < MIN - 1/2 LSB in magnitude to an underflow
- Chopped rounding
  - Maps the true result to the smaller in magnitude of two surrounding representable results
  - Maps true results ≥ MAX + 1 LSB in magnitude to an overflow
  - Maps true results < MIN in magnitude to an underflow

For IEEE floating-point operations, four rounding modes are provided:

- Normal rounding (unbiased round to nearest)
  - Maps the true result to the nearest of two representable results, with true results exactly halfway between mapped to the one whose fraction ends in 0 (sometimes called unbiased rounding to even)
  - Maps true results ≥ MAX + 1/2 LSB in magnitude to an overflow
  - Maps true results < MIN - 1/2 LSB in magnitude to an underflow
- Rounding toward minus infinity
  - Maps the true results to the smaller of two surrounding representable results

- Maps true results > MAX in magnitude to an overflow
  - Maps positive true results < +MIN to an underflow
  - Maps negative true results ≥ -MIN + 1 LSB to an underflow
- Chopped rounding (round toward zero)
  - Maps the true result to the smaller in magnitude of two surrounding representable results
  - Maps true results ≥ MAX + 1 LSB in magnitude to an overflow
  - Maps nonzero true results < MIN in magnitude to an underflow
- Rounding toward plus infinity
  - Maps the true results to the larger of two surrounding representable results
  - Maps true results > MAX in magnitude to an overflow
  - Maps positive results ≤ +MIN - 1 LSB to an underflow
  - Maps negative true results > -MIN to an underflow

The first three of the IEEE rounding modes can be specified in the instruction. The last mode, rounding toward plus infinity, can be obtained by setting the floating-point control register (FPCR) to select it and then specifying dynamic rounding mode in the instruction.

Dynamic rounding mode uses the IEEE rounding mode selected by the FPCR and is described in Section 4.1.2. Dynamic rounding can be used with any of the IEEE rounding modes.

Alpha AXP IEEE arithmetic does rounding before detecting overflow or underflow.

4.1.5 Floating-Point Instruction Qualifiers

Many of the floating-point instructions accept a qualifier that specifies rounding and trapping modes.

The following table lists the rounding mode qualifiers. See Section 4.1.4 for a detailed description of the rounding modes.

Rounding Mode	Qualifier
VAX Rounding Mode	
Normal rounding	(no modifier)
Chopped	`c`
IEEE Rounding Mode	
Normal rounding	(no modifier)
Plus infinity	`d` (ensure that the `dyn` field of the FPCR is 11)
Minus infinity	`m`
Chopped	`c`

The following table lists the trapping mode qualifiers. See Section 4.1.3 for a detailed description of the exceptions.

Trapping Mode	Qualifier
VAX Trap Mode	
Imprecise, underflow disabled	(no modifier)
Imprecise, underflow enabled	`u`
Software, underflow disabled	`s`
Software, underflow enabled	`su`
VAX Convert-to-Integer Trap Mode	
Imprecise, integer overflow disabled	(no modifier)
Imprecise, integer overflow enabled	`v`
Software, integer overflow disabled	`s`
Software, integer overflow enabled	`sv`
IEEE Trap Mode	
Imprecise, underflow disabled, inexact disabled	(no modifier)
Imprecise, underflow enabled, inexact disabled	`u`
Software, underflow enabled, inexact disabled	`su`
Software, underflow enabled, inexact enabled	`sui`
IEEE Convert-to-integer Trap Mode	
Imprecise, integer overflow disabled, inexact disabled	(no modifier)

Trapping Mode	Qualifier
Imprecise, integer overflow enabled, inexact disabled	v
Software, integer overflow enabled, inexact disabled	sv
Software, integer overflow enabled, inexact enabled	svi

Table 4-1 lists the qualifier combinations that are supported by one or more of the individual instructions. The Number column of the table is referenced in subsequent sections to identify the combination of qualifiers accepted by the various instructions.

Table 4-1: Qualifier Combinations for Floating-Point Instructions

Number	Qualifiers
1	c, u, uc, s, sc, su, suc
2	c, m, d, u, uc, um, ud, su, suc, sum, sud, sui, suic, suim, suid
3	s
4	su
5	sv, v
6	c, v, vc, s, sc, sv, svc
7	c, v, vc, sv, svc, svi, svic, d, vd, svd, svid
8	c
9	c, m, d, sui, suic, suim, suid

4.2 Floating-Point Load and Store Instructions

The load and store instructions move data between memory and floating-point registers.

Table 4-2 lists the mnemonics and operands for instructions that perform floating-point load and store operations. The table is segmented into groups of functionally related instructions. The operands specified within a particular segment of the table apply to all of the instructions contained in that segment.

Table 4-2: Load and Store Instruction Formats

Instruction	Mnemonic	Operands
Load F_floating	`ldf`	*d_reg, address*
Load G_floating (Load D_floating)	`ldg`	
Load S_floating (Load Longword)	`lds`	
Load T_floating (Load Quadword)	`ldt`	
Load Immediate F_floating	`ldif`	*d_reg, val_expr*
Load Immediate D_floating	`ldid`	
Load Immediate G_floating	`ldig`	
Load Immediate S_floating (Load Longword)	`ldis`	
Load Immediate T_floating (Load Quadword)	`ldit`	
Store F_floating	`stf`	*s_reg, address*
Store G_floating (Store D_floating)	`stg`	
Store S_floating (Store Longword)	`sts`	
Store T_floating (Store Quadword)	`stt`	

Table 4-3 describes the operations performed by floating-point load and store instructions.

The load and store instructions are grouped by function in the table. Refer to Table 4-2 for the instruction names.

Table 4-3: Load and Store Instruction Descriptions

Instruction	Description
Load Instructions (`ldf`, `ldg`, `lds`, `ldt`, `ldif`, `ldid`, `ldig`, `ldis`, `ldit`)	Load eight bytes (G_, D_, and T_floating formats) or four bytes (F_ and S_floating formats) from the specified effective address into the destination register. The address must be quadword aligned for 8-byte load instructions and longword aligned for 4-byte load instructions.
Store Instructions (`stf`, `stg`, `sts`, `stt`)	Store eight bytes (G_, D_, and T_floating formats) or four bytes (F_ and S_floating formats) from the source floating-point register into the specified effective address. The address must be quadword aligned for 8-byte store instructions and longword aligned for 4-byte store instructions.

4.3 Floating-Point Arithmetic Instructions

This section describes the floating-point arithmetic instructions supported by the assembler. These instructions perform arithmetic and logical operations on values in floating-point registers.

Table 4-4 lists the mnemonics and operands for instructions that perform floating-point arithmetic and logical operations. The table is segmented into groups of functionally related instructions. The operands specified within a segment of the table apply to all of the instructions contained in that segment.

The Qualifiers column in Table 4-4 refers to one or more trap or rounding modes as specified in Table 4-1.

Table 4-4: Arithmetic Instruction Formats

Instruction	Mnemonic	Qualifiers	Operands
Floating Clear	`fclr`	–	d_reg
Floating Absolute Value	`fabs`	–	*s_reg, d_reg* *d_reg/s_reg*
Negate F_floating	`negf`	3	
Negate G_floating	`negg`	3	
Negate S_floating	`negs`	4	
Negate T_floating	`negt`	4	
Add F_floating	`addf`	1	*s_reg1, s_reg2, d_reg* *d_reg/s_reg1, s_reg2*
Add G_floating	`addg`	1	
Add S_floating	`adds`	2	
Add T_floating	`addt`	2	
Divide F_floating	`divf`	1	
Divide G_floating	`divg`	1	
Divide S_floating	`divs`	2	
Divide T_floating	`divt`	2	
Multiply F_floating	`mulf`	1	
Multiply G_floating	`mulg`	1	
Multiply S_floating	`muls`	2	
Multiply T_floating	`mult`	2	
Subtract F_floating	`subf`	1	
Subtract S_floating	`subg`	1	
Subtract T_floating	`subs`	2	
	`subt`	2	

Table 4-4: (continued)

Instruction	Mnemonic	Qualifiers	Operands
Convert Quadword to Longword	`cvtql`	5	$\left\{\begin{matrix} s\_reg,\ d\_reg \\ d\_reg/s\_reg \end{matrix}\right\}$
Convert Longword to Quadword	`cvtlq`	–	
Convert G_floating to Quadword	`cvtgq`	6	
Convert T_floating to Quadword	`cvttq`	7	
Convert Quadword to F_floating	`cvtqf`	8	
Convert Quadword to G_floating	`cvtqg`	8	
Convert Quadword to S_floating	`cvtqs`	9	
Convert Quadword to T_floating	`cvtqt`	9	
Convert D_floating to G_floating	`cvtdg`	1	
Convert G_floating to D_floating	`cvtgd`	1	
Convert G_floating to F_floating	`cvtgf`	1	
Convert T_floating to S_floating	`cvtts`	2	
Convert S_floating to T_floating	`cvtst`	3	

Table 4-5 describes the operations performed by floating-point load and store instructions. The arithmetic instructions are grouped by function in the table. Refer to Table 4-4 for the instruction names.

Table 4-5: Arithmetic Instruction Descriptions

Instruction	Description
Clear Instruction (`fclr`)	Clear the destination register.
Absolute Value Instruction (`fabs`)	Compute the absolute value of the contents of the source register and put the floating-point result in the destination register.

Table 4-5: (continued)

Instruction	Description
Negate Instructions (`negf`, `negg`, `negs`, `negt`)	Compute the negative value of the contents of *s_reg* or *d_reg* and put the specified precision floating-point result in *d_reg*.
Add Instructions (`addf`, `addg`, `adds`, `addt`)	Add the contents of *s_reg* or *d_reg* to the contents of *s_reg2* and put the result in *d_reg*. When the sum of two operands is exactly zero, the sum has a positive sign for all rounding modes except round toward -∞. For that rounding mode, the sum has a negative sign.
Divide Instructions (`divf`, `divg`, `divs`, `divt`)	Compute the quotient of two values. These instructions divide the contents of *s_reg1* or *d_reg* by the contents of *s_reg2* and put the result in *d_reg*. If the divisor is a zero, an error is signaled if the divide-by-zero exception is enabled.
Multiply Instructions (`mulf`, `mulg`, `muls`, `mult`)	Multiply the contents of *s_reg1* or *d_reg* with the contents of *s_reg2* and put the result in *d_reg*.
Subtract Instructions (`subf`, `subg`, `subs`, `subt`)	Subtract the contents of *s_reg2* from the contents of *s_reg1* or *d_reg* and put the result in *d_reg*. When the difference of two operands is exactly zero, the difference has a positive sign for all rounding modes except round toward -∞. For that rounding mode, the sum has a negative sign.
Conversion Between Integer Formats Instructions	
(`cvtql`, `cvtlq`)	Convert the integer contents of *s_reg* to the specified integer format and put the result in *d_reg*. If an integer overflow occurs, the truncated result is stored in *d_reg* and, if enabled, an arithmetic trap occurs.
Conversion from Floating-Point to Integer Format Instructions	
(`cvtgq`, `cvttq`)	Convert the floating-point contents of *s_reg* to the specified integer format and put the result in *d_reg*. If an integer overflow occurs, the truncated result is stored in *d_reg* and, if enabled, an arithmetic trap occurs.
Conversion from Integer to Floating-Point Format Instructions	
(`cvtqf`, `cvtqg`, `cvtqs`, `cvtqt`)	Convert the integer contents of *s_reg* to the specified floating-point format and put the result in *d_reg*.
Conversion Between Floating-Point Formats Instructions	
(`cvtdg`, `cvtgd`, `cvtgf`, `cvtts`, `cvtst`)	Convert the contents of *s_reg* to the specified precision, round according to the rounding mode, and put the result in *d_reg*. If an overflow occurs, an unpredictable value is stored in *d_reg* and a floating-point trap occurs.

4.4 Floating-Point Relational Instructions

This section describes the floating-point relational instructions supported by the assembler. The relational instructions compare two floating-point values.

Table 4-6 lists the mnemonics and operands for instructions that perform floating-point relational operations. Each of the instructions can take an operand in any of the forms shown in the table.

The Qualifiers column in Table 4-6 refers to one or more trap or rounding modes as specified in Table 4-1.

Table 4-6: Relational Instruction Formats

Instruction	Mnemonic	Qualifiers	Operands
Compare G_floating Equal	`cmpgeq`	3	*s_reg1, s_reg2, d_reg* *d_reg/s_reg1, s_reg2*
Compare G_floating Less Than	`cmpglt`	3	
Compare G_floating Less Than or Equal	`cmpgle`	3	
Compare T_floating Equal	`cmpteq`	4	
Compare T_floating Less Than	`cmptlt`	4	
Compare T_floating Less Than or Equal	`cmptle`	4	
Compare T_floating Unordered	`cmptun`	4	

Table 4-7 describes the relational instructions supported by the assembler. The relational instructions are grouped by function in the table. Refer to Table 4-6 for the instruction names.

Table 4-7: Relational Instruction Descriptions

Instruction	Description
Compare Equal Instructions (`cmpgeq`, `cmpteq`)	Compare the contents of *`s_reg1`* with the contents of *`s_reg2`*. If *`s_reg1`* equals *`s_reg2`*, a nonzero value is written to the destination register; otherwise, a true zero value is written to the destination. Exceptions are not signaled for unordered values.

Table 4-7: (continued)

Instruction	Description
Compare Less Than Instructions (`cmpglt`, `cmptlt`)	Compare the contents of *s_reg1* with the contents of *s_reg2*. If *s_reg1* is less than *s_reg2*, a nonzero value is written to the destination register; otherwise, a true zero value is written to the destination. Exceptions are not signaled for unordered values.
Compare Less Than or Equal Instructions (`cmpgle`, `cmptle`)	Compare the contents of *s_reg1* with the contents of *s_reg2*. If *s_reg1* is less than or equal to *s_reg2*, a nonzero value is written to the destination register; otherwise, a true zero value is written to the destination. Exceptions are not signaled for unordered values.
Compare Unordered Instruction (`cmptun`)	Compare the contents of *s_reg1* with the contents of *s_reg2*. If either *s_reg1* or *s_reg2* is unordered, a nonzero value is written to the destination register; otherwise, a true zero value is written to the destination. Exceptions are not signaled for unordered values.

4.5 Floating-Point Move Instructions

This section describes the floating-point move instructions supported by the assembler. These instructions move data between floating-point registers.

Table 4-8 lists the mnemonics and operands for instructions that perform floating-point move operations. The table is segmented into groups of functionally related instructions. The operands specified within a particular segment of the table apply to all of the instructions contained in that segment.

Table 4-8: Move Instruction Formats

Instruction	Mnemonic	Operands
Floating Move	`fmov`	*s_reg*, *d_reg*

Table 4-8: (continued)

Instruction	Mnemonic	Operands
Copy Sign	cpys	{ *s_reg1, s_reg2, d_reg* / *d_reg/s_reg1, s_reg2* }
Copy Sign Negate	cpysn	
Copy Sign and Exponent	cpyse	
Move if Equal to Zero	fcmoveq	
Move if Not Equal to Zero	fcmovne	
Move if Less Than Zero	fcmovlt	
Move if Less Than or Equal to Zero	fcmovle	
Move if Greater Than Zero	fcmovgt	
Move if Greater Than or Equal to Zero	fcmovge	

Table 4-9 describes the operations performed by move instructions. The move instructions are grouped by function in the table. Refer to Table 4-8 for the instruction names.

Table 4-9: Move Instruction Descriptions

Instruction	Description
Move Instruction (fmov)	Move the double or single precision contents of the *s_reg* to *d_reg*.
Copy Sign Instruction (cpys)	Fetch the sign bit of *s_reg1* or *d_reg*, combine it with the exponent and fraction of *s_reg2*, and copy the result to *d_reg*.
Copy Sign Negate Instruction (cpysn)	Fetch the sign bit of *s_reg1* or *d_reg*, complement it, combine it with the exponent and fraction of *s_reg2*, and copy the result to *d_reg*.
Copy Sign and Exponent Instruction (cpyse)	Fetch the sign and exponent of *s_reg1* or *d_reg*, combine them with the fraction of *s_reg2*, and copy the result to *d_reg*.
Move If Instructions (fcmoveq, fcmovne, fcmovlt, fcmovle, fcmovgt, fcmovge)	Compare the contents of *s_reg1* or *d_reg* against zero. If the specified condition is true, the contents of *s_reg2* is copied to *d_reg*; otherwise, *d_reg* is unchanged.

4.6 Floating-Point Control Instructions

This section describes floating-point control instructions supported by the assembler. These instructions test floating-point registers and conditionally branch.

Table 4-10 lists the mnemonics and operands for instructions that perform floating-point control operations. The operands specified in the table apply to all of the instructions listed in the table.

Table 4-10: Control Instruction Formats

Instruction	Mnemonic	Operands
Branch Equal to Zero	`fbeq`	*s_reg,label*
Branch Not Equal to Zero	`fbne`	
Branch Less Than Zero	`fblt`	
Branch Less Than or Equal to Zero	`fble`	
Branch Greater Than Zero	`fbgt`	
Branch Greater Than or Equal to Zero	`fbge`	

Table 4-11 describes the operations performed by control instructions The control instructions are grouped by function in the table. Refer to Table 4-10 for instruction names.

Table 4-11: Control Instruction Descriptions

Instruction	Description
Branch Instructions (`fbeq`, `fbne`, `fblt`, `fble`, `fbgt`, `fbge`)	The contents of the source register are compared with zero. If the specified relationship is true, a branch is made to the specified label.

4.7 Floating-Point Special-Purpose Instructions

This section describes special-purpose instructions supported by the assembler. These instructions perform miscellaneous tasks.

Table 4-12 lists the mnemonics and operands for instructions that perform floating-point special-purpose operations.

Table 4-12: Special-Purpose Instruction Formats

Instruction	Mnemonic	Operands
Move from FP Control Register	`mf_fpcr`	*d_reg*
Move to FP Control Register	`mt_fpcr`	*s_reg*
No Operation	`fnop`	

Table 4-13 describes the operations performed by floating-point special-purpose instructions.

Table 4-13: Control Register Instruction Descriptions

Instruction	Description
Move to FPCR Instruction (`mf_fpcr`)	Copy the value in the specified source register to the floating-point control register (FPCR).
Move from FPCR Instruction (`mt_fpcr`)	Copy the value in floating-point control register (FPCR) to the specified destination register.
No Operation Instruction (`fnop`)	This instruction has no effect on the machine state.

Assembler Directives 5

This chapter describes assembler directives. Assembler directives are instructions to the assembler to perform various bookkeeping tasks, storage reservation, and other control functions. To distinquish them from other instructions, directive names begin with a period.

Table 5-1 lists the assembler directives by category.

Table 5-1: Summary of Assembler Directives

Category	Directives

Compiler-Use-Only Directives	`.bgnb`
	`.file`
	`.gjsrlive`
	`.gjsrsaved`
	`.lab`
	`.livereg`
	`.loc`
	`.option`
	`.ugen`
	`.vreg`

Data Storage Directives	`.ascii`
	`.asciiz`
	`.byte`
	`.comm`
	`.d_floating`
	`.float`
	`.f_floating`
	`.gprel32`
	`.g_floating`
	`.lcomm`
	`.long`
	`.quad`
	`.s_floating`
	`.t_floating`
	`.word`

Table 5-1: (continued)

Category	Directives

Location Control Directives	`.align`
	`.data`
	`.rdata`
	`.sdata`
	`.space`
	`.text`

Symbol Declaration Directives	`.extern`
	`.globl`
	`.struct`
	(symbolic equate)
	`.weakext`

Routine Entry Point Definition Directives	`.aent` `.ent`
Repeat Block Directives	`.endr` `.repeat`
Assembler Option Directive	`.set`

Procedure Attribute Directives	`.edata`
	`.eflag`
	`.end`
	`.fmask`
	`.frame`
	`.mask`
	`.prologue`
	`.save_ra`

Version Control Directive	`.verstamp`

The assembly directives are described alphabetically in the following list:

.aent *name [, symno]*
: Sets an alternate entry point for the current procedure. Use this information when you want to generate information for the debugger. This directive must appear between a pair of `.ent` and `.end` directives. (The optional *`symno`* is for compiler use only. It refers to a dense number in a `.T` file (symbol table).)

.alias *reg1, reg2*
: Indicates that memory referenced through the two registers will overlap. The compiler uses this form to improve instruction scheduling.

.align *expression*
: Sets low-order bits in the location counter to zero. The value of *expression* establishes the number of bits to be zeroed. The maximum

value for *expression* is four (octaword alignment).

If the `.align` directive advances the location counter, the assembler fills the skipped bytes with zeros (in data sections) or `nop` instructions (in text sections).

Normally, the `.word`, `.long`, `.quad`, `.float`, `.double`, `.d_floating`, `.f_floating`, `.g_floating`, `.s_floating`, and `.t_floating` directives automatically align their data appropriately. For example, `.word` does an implicit `.align 1`, and `.double` does an implicit `.align 3`.

You can disable the automatic alignment feature with `.align 0`. The assembler reinstates automatic alignment at the next `.text`, `.data`, `.rdata`, or `.sdata` directive that it encounters.

Labels immediately preceding an automatic or explicit alignment are also realigned. For example,

```
foo: .align 3
     .word 0
```

is the same as

```
     .align 3
foo: .word 0
```

.ascii *string [, string] ...*
Assembles each *string* from the list into successive locations. The `.ascii` directive does not null pad the string. You must put quotation marks (") around each string. You can optionally 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 optionally use the backslash escape characters. For a list of the backslash characters, see Chapter 4.

.bgnb *symno*
For use only 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 (symbol table). For an explanation of `.T` files, see Chapter 8.

.byte *expression1 [, expression2] ... [, expressionN]*
Truncates the values of the expressions specified in the comma-separated list to 8-bit values, and assembles the values in successive locations. The values of the expressions must be absolute.

The operands in a `.byte` directive can optionally have the form:

expressionVal [: *expressionRep*]

The *expressionVal* is an 8-bit value. The optional *expressionRep* is a non-negative expression that specifies how many times to replicate the value of *expressionVal*. The expression value (*expressionVal*) and repetition count (*expressionRep*) must be absolute.

.comm *name, expression*
Unless defined elsewhere, *name* becomes a global common symbol at the head of a block of at least *expression* bytes of storage. The linker overlays like-named common blocks, using the expression value of the largest block as the byte size of the overlay.

.data
Directs the assembler to add all subsequent data to the `.data` section.

.d_floating *expression1 [, expression2] ... [, expressionN]*
Initializes memory to double precision (64-bit) VAX D_floating numbers. The values of the expressions must be absolute.

The operands can optionally have the form:

expressionVal [: *expressionRep*]

The *expressionVal* is a 64-bit value. The optional *expressionRep* is a non-negative expression that specifies how many times to replicate the value of *expressionVal*. The expression value (*expressionVal*) and repetition count (*expressionRep*) must be absolute.

This directive automatically aligns its data and any preceding labels to a double-word boundary. You can disable this feature with the `.align 0` directive .

.double *expression1 [, expression2] ... [, expressionN]*
Synonym for `.t_floating`.

.edata *flag data lang-handler*
Marks data related to exception handling.

If *flag* is zero, the assembler moves the data following the directive to the `.xdata` section.

If *flag* is 1, the assembler creates a function table entry for the next `.ent` directive. The function table entry contains the language-specific handler and data.

.eflag *flags*
Encodes exception related flags to be stored in the `PDSC_RPD_FLAGS` field of the procedure's run-time procedure descriptor. Refer to the Alpha AXP Calling Standard for DEC OSF/1 for a description of the individual flags.

.end *[proc_name]*

Sets the end of a procedure. The `.ent` directive sets the beginning of a procedure. Use the `.ent` and `.end` directives when you want to generate information for the debugger.

.endb *symno*

Sets the end of a language block. (See the description of the `.bgnb` directive for details. The `.bgnb` directive sets the beginning of a language block.)

.endr

Signals the end of a repeat block. The `.repeat` directive starts a repeat block.

.ent *proc_name [lex-level]*

Sets the beginning of the procedure *proc_name*. Use this directive when you want to generate information for the debugger. The `.end` directive sets the end of a procedure.

The *lex-level* operand indicates the number of procedures that statically surround the current procedure. This operand is only informational; it does not affect the assembly process (that is, the assembler ignores it).

.err

For use only by compilers. 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` directive, it quietly ceases to assemble the source file. This prevents the assembler from continuing to process a program that is incorrect.

.extern *name*

Indicates that the specified symbol is global and external; that is, the symbol is defined in another object module and cannot be defined until link time. The *name* operand is a global undefined symbol.

.f_floating *expression1 [, expression2] ... [, expressionN]*

Initializes memory to single precision (32-bit) VAX F_floating numbers. The values of the expressions must be absolute.

The operands can optionally have the form:

expressionVal [: *expressionRep*]

The *expressionVal* is a 32-bit value. The optional *expressionRep* is a non-negative expression that specifies how many times to replicate the value of *expressionVal*. The expression value (*expressionVal*) and repetition count (*expressionRep*) must be absolute.

This directive automatically aligns its data and preceding labels to a longword boundary. You can disable this feature by using the `.align 0` directive.

.file *file_number file_name_string*

For use only by compilers. Specifies the source file corresponding to the assembly instructions that follow. This directive causes the assembler to stop generating line numbers that are used by the debugger. A subsequent `.loc` directive causes the assembler to resume generating line numbers.

.float *expression1 [, expression2] ... [, expressionN]*

Synonym for `.s_floating`.

.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 to where the floating-point registers are saved.

You must use `.ent` before `.fmask`, and you can use only one `.fmask` for each `.ent`. Space should be allocated for those registers specified in the `.fmask`.

.frame *frame-register frame-size return_pc-register [local_offset]*

Describes a stack frame. The first register is the frame register, *frame-size* is the size of the stack frame, that is, the number of bytes between the frame register and the virtual frame pointer. The second register specifies the register that contains the return address. The *local_offset* parameter, which is for use only by compilers, specifies the number of bytes between the virtual frame pointer and the local variables.

You must use `.ent` before `.frame`, and you can use only one `.frame` for each `.ent`. No stack traces can be done in the debugger without the `.frame` directive.

.g_floating *expression1 [, expression2] ... [, expressionN]*

Initializes memory to double precision (64-bit) VAX G_floating numbers. The values of the expressions must be absolute.

The operands can optionally have the form:

expressionVal [: *expressionRep*]

The *expressionVal* is a 64-bit value. The optional *expressionRep* is a non-negative expression that specifies how many times to replicate the value of *expressionVal*. The expression value (*expressionVal*) and repetition count (*expressionRep*) must be absolute.

This directive automatically aligns its data and any preceding labels to a quadword boundary. You can disable this feature with the `.align 0` directive.

.gjsrlive

For use only by compilers. Sets the default masks for live registers before a procedure call (a `bsr` or `jsr` instruction).

.gjsrsaved

For use only by compilers. Sets the masks defining the registers whose value is preserved during a procedure call. See Table 6-1 and Table 6-2 for the default for integer and floating-point saved registers.

.globl *name*

Identifies *name* as an external symbol. If the name is otherwise defined (for example, by its appearance as a label), the assembler exports the symbol; otherwise, it imports 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.

.gprel32 *address1 [, address2] ... [, addressN]*

Truncates the signed displacement between the global pointer value and the addresses specified in the comma-separated list to 32-bit values, and assembles the values in successive locations.

The operands can optionally have the form:

addressVal [: *addressRep*]

The *addressVal* is the address value. The optional *addressRep* is a non-negative expression that specifies how many times to replicate the value of *addressVal*. The expression value (*expressionVal*) and repetition count (*expressionRep*) must be absolute.

This directive automatically aligns its data and preceding labels to a longword boundary. You can disable this feature with the `.align 0` directive.

.gretlive

For use by compilers. Sets the default masks for live registers before a procedure's return (a `ret` instruction).

.lab *label_name*

For use only by compilers. Associates a named label with the current location in the program text.

.lcomm *name, expression*

Gives the named symbol (*name*) a data type of `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 less 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.

.livereg *int_bitmask fp_bitmask*

For use only by compilers. Affects the next jump instruction even if it is not the successive instruction. By default, external br instructions and jmp instructions are treated as external calls; that is; all registers are assumed to be live. The .livereg directive cannot appear before an external br instruction because it will affect the next ret, jsr, bsr, jmp, or call_pal callsys instruction instead of the br instruction. The directive cannot be used before a call_pal bpt instruction. For call_pal bpt instructions, the assembler also assumes that all registers are live.

To avoid unsafe optimizations by the reorganizer, .livereg notes to the assembler those registers that are live before a jump. 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, for example, .mask and .fmask). The first bitmap indicates live integer registers and the second indicates live floating-point registers.

When present, this directive causes the assembler to be more conservative and to not destroy the indicated register contents. If omitted, the assembler assumes the default masks. The .livereg directive can be coded before any of the following instructions: bsr, jsr, ret, jmp, and call_pal callsys.

.loc *file_number line_number*

For use only by compilers. Specifies the source file and the line within it 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 Chapter 8.

.long *expression1 [, expression2] ... [, expressionN]*

Truncates the values of the expressions specified in the comma-separated list to 32-bit values, and assembles the values in successive locations. The values of the expression can be relocatable.

The operands can optionally have the form:

expressionVal [: *expressionRep*]

The *expressionVal* is a 32-bit value. The optional *expressionRep* is a non-negative expression that specifies how many times to replicate the value of *expressionVal*. The expression value (*expressionVal*) and repetition count (*expressionRep*) must be absolute.

This directive automatically aligns its data and preceding labels to a longword boundary. You can disable this feature with the `.align 0` directive.

.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 to where the registers are saved.

You must use `.ent` before `.mask`, and you can use only one `.mask` for each `.ent`. Space should be allocated for those registers specified in the `.mask`. If bit zero is set, it is assumed that space is allocated for all 31 integer registers, regardless of whether they appear in the mask.

.noalias *reg1, reg2*

Informs the assembler that *reg1* and *reg2* will never point to the same memory location when they are used as indexed registers. The assembler uses this as a hint to make more liberal assumptions about resource dependency in the program.

.option *options*

For use only by compilers. Informs the assembler that certain options were in effect during compilation. (These options can, for example, limit the assembler's freedom to perform branch optimizations.)

.prologue *flag*

Marks the end of the prologue section of a procedure.

A *flag* of zero indicates that the procedure does not use `$gp`; the caller does not need to set up `$pv` prior to calling the procedure or restore `$gp` on return from the procedure.

A *flag* of one indicates that the procedure does use `$gp`; the caller must set up `$pv` prior to calling the procedure and restore `$gp` on return from the procedure.

If *flag* is not specified, the behavior is as if a value of one was specified.

.quad *expression1 [, expression2] ... [, expressionN]*

Truncates the values of the expressions specified in the comma-separated list to 64-bit values, and assembles the values in successive locations. The values of the expressions can be relocatable.

The operands can optionally have the form:

expressionVal [: *expressionRep*]

The *expressionVal* is a 64-bit value. The optional *expressionRep* is a non-negative expression that specifies how many times to replicate the value of *expressionVal*. The expression value (*expressionVal*) and repetition count (*expressionRep*) must be absolute.

This directive automatically aligns its data and preceding labels to a quadword boundary. You can disable this feature with the `.align 0` directive.

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

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

.save_ra *saved_ra_register*
Specifies that *saved_ra_register* is the register in which the return address is saved during the execution of the procedure. If `.save_ra` is not used, the saved return address register is assumed to be the same as the *return_pc_register* argument of the `frame` directive. The `.save_ra` directive is valid only for register frame procedures.

.sdata
Instructs the assembler to add subsequent data to the `.sdata` section.

.set *option*
Instructs the assembler to enable or disable certain options. The assembler has the following default options: `reorder`, `macro`, `move`, `novolatile`, and `at`. You can specify only one option for each `.set` directive:

- The `reorder` option permits the assembler to reorder machine language instructions to improve performance.
- The `noreorder` option prevents the assembler from reordering machine language instructions. If a machine language instruction violates the hardware pipeline constraints, the assembler issues a warning message.
- The `macro` option permits the assembler to 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 permits the assembler to 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 permit source programs to use `$at` without warnings being issued.

- The `nomove` options instructs the assembler to mark each subsequent instruction so that it cannot be moved during reorganization. The assembler can still move instructions from below the `nomove` region to above the region or vice versa. The `nomove` option has part of the effect of the ‘‘volatile’’ C declaration; it prevents otherwise independent loads or stores from occurring in a different order than intended.
- The `move` option cancels the effect of `nomove`.
- The `volatile` option instructs the assembler that subsequent load and store instructions may not be moved in relation to each other or removed by redundant load removal or other optimization. The `volatile` option is less restrictive than `noreorder`; it allows the assembler to move other instructions (that is, instructions other than load and store instructions) without restrictions.
- The `novolatile` option cancels the effect of the `volatile` option.

.s_floating *expression1 [, expression2] ... [, expressionN]*

Initializes memory to single precision (32-bit) IEEE floating-point numbers. The values of the expressions must be absolute.

The operands for the `.s_floating` directive can optionally have the form:

expressionVal [: *expressionRep*]

The *expressionVal* is a 32-bit value. The optional *expressionRep* is a non-negative expression that specifies how many times to replicate the value of *expressionVal*. The expression value (*expressionVal*) and repetition count (*expressionRep*) must be absolute.

This directive automatically aligns its data and preceding labels to a longword boundary. You can disable this feature with the `.align 0` directive.

.space *expression*

Advances the location counter by the number of bytes specified by the value of *expression*. The assembler fills the space with zeros.

.struct *expression*

Permits you to lay out a structure using labels plus directives such as `.word` or `.byte`. It ends at the next segment directive (`.data`, `.text`, and so forth). It does not emit any code or data, but defines the labels within it to have values that are the sum of *expression* plus their offsets from the `.struct` itself.

(symbolic equate)

Takes one of the following 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 only operators, constants, or equated symbols. You can use the name as a constant in any later statement.

.text

Instructs the assembler to add subsequent code to the `.text` section. (This is the default.)

.t_floating *expression1 [, expression2] ... [, expressionN]*

Initializes memory to double precision (64-bit) IEEE floating-point numbers. The values of the expressions must be absolute.

The operands can optionally have the form:

expressionVal [: *expressionRep*]

The *expressionVal* is a 64-bit value. The optional *expressionRep* is a non-negative expression that specifies how many times to replicate the value of *expressionVal*. The expression value (*expressionVal*) and repetition count (*expressionRep*) must be absolute.

This directive automatically aligns its data and any preceding labels to a quadword boundary. You can disable this feature with the `.align 0` directive.

.ugen

For use only by compilers. Informs the assembler that the source was generated by the code generator.

.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 only 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.

.weakext *name*

Sets *name* to be a weak symbol during linking. Weak symbols can be silently redefined at link time.

.word *expression1 [, expression2] ... [, expressionN]*

Truncates the values of the expressions specified in the comma-separated list to 16-bit values, and assembles the values in successive locations. The values of the expressions must be absolute.

The operands in the `.word` directive can optionally have the form:

expressionVal [: *expressionRep*]

The *expressionVal* is a 16-bit value. The optional *expressionRep* is a non-negative expression that specifies how many times to replicate the value of *expressionVal*. The expression value (*expressionVal*) and repetition count (*expressionRep*) must be absolute.

This directive automatically aligns its data and preceding labels on a word boundary. You can disable this feature with the `.align 0` directive.

Programming Considerations 6

This chapter gives rules and examples to follow when creating an assembly-language program.

The chapter contains the following major sections:

- Calling Conventions (Section 6.1) – explains why the calling conventions observed by the C compiler should be used in your assembly programs.
- Program Model (Section 6.2) – provides an overview of the composition of executable programs.
- General Coding Concerns (Section 6.3) – provides information on the use of registers, section and location counters, and stack frames.
- Developing Code for Procedure Calls (Section 6.4) – describes how a particular calling sequence should work. This technique involves writing a skeleton version of your prospective assembly procedure using a high-level language, and then compiling it with the `-S` option to generate an output file (`.s` file) that contains an assembly language version of the compiled source program. The assembly language in the output file can then be used as the starting point for coding your procedure.
- Memory Allocation (Section 6.5) – describes the default memory allocation scheme used by the Alpha AXP system.

This chapter does not adress coding issues related to performance or optimization. See the *Alpha Architecture Reference Manual* (Appendix A) for information on how to optimize assembly code.

6.1 Calling Conventions

When you write assembly-language procedures, you should use the same calling conventions that the C compiler observes. The reasons for using the same calling conventions are as follows:

- 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 that use standard calling conventions.

The conventions observed by the DEC OSF/1 compiler system are more complicated than those of some other compiler systems, mostly to enhance the speed of each procedure call. Specifically:

- The C compiler uses the full, general calling sequence only when necessary; whenever possible, it omits unneeded portions of the sequence. For example, the C compiler does not use a register as a frame pointer if it is unnecessary to do so.
- The C compiler and the debugger observe certain implicit rules instead of communicating by means of 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. This technique enables the debugger to tolerate the lack of a register containing a frame pointer at execution time.
- The linker performs code optimizations based on information that is not available at compile time. For example, the linker can, in some cases, replace the general calling sequence to a procedure with a single instruction.

6.2 Program Model

A program consists of an executable image and zero or more shared images. Each image has an independent text and data area.

Each data segment contains a global offset table (GOT), which contains address constants for procedures and data locations that the text segment references. The GOT provides the means to access arbitrary 64-bit addresses and allows the text segment to be position independent. The size of the GOT is limited only by the maximum image size. However, because only 64KB can be addressed by a single memory-format instruction, the GOT is segmented into one or more sections of 64KB or less.

In addition to providing efficient access to the GOT, the `gp` register is also used to access global data within ±2GB of the global pointer. This area of memory is known as the global data area.

A static executable image is not a special case in the program model. It is simply an executable image that uses no shared libraries. However, it is possible for the linker to perform code optimizations. In particular, if a static executable image's GOT is less than or equal to 64KB (that is, it has only one segment), the code to load, save, and restore the `gp` register is not necessary because all procedures will access the same GOT segment.

6.3 General Coding Concerns

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

- Usable and restricted registers
- Control of section and location counters with directives
- Stack frame requirements on entering and exiting a procedure

Another general coding consideration is the use of data structures to communicate between high-level language procedures and assembly procedures. In most cases, this communication is handled by means of simple variables: pointers, integers, booleans, and single- and double-precision real numbers. Describing the details of the various high-level data structures that can also be used (arrays, records, sets, and so on) is beyond the scope of this manual.

6.3.1 Register Use

The main processor has 32 64-bit integer registers. The uses and restrictions of these registers are described in Table 6-1.

The floating-point coprocessor has 32 floating-point registers. Each register can hold either a single precision (32 bit) or a double precision (64 bit) value. Refer to Table 6-2 for details.

Table 6-1: Integer Registers

Register Name	Software Name (from regdef.h)	Use
`$0`	`v0`	Used for expression evaluations and to hold the integer function results. Not preserved across procedure calls.
`$1-8`	`t0-t7`	Temporary registers used for expression evaluations. Not preserved across procedure calls.
`$9-14`	`s0-s5`	Saved registers. Preserved across procedure calls.
`$15` or `$fp`	`s6` or `fp`	Contains the frame pointer (if needed); otherwise, a saved register.
`$16-21`	`a0-a5`	Used to pass the first six integer type actual arguments. Not preserved across procedure calls.
`$22-25`	`t8-t11`	Temporary registers used for expression evaluations. Not preserved across procedure calls.
`$26`	`ra`	Contains the return address. Preserved across procedure calls.

Table 6-1: (continued)

Register Name	Software Name (from regdef.h)	Use
`$27`	`pv` or `t12`	Contains the procedure value and used for expression evaluation. Not preserved across procedure calls.
`$28` or `$at`	`AT`	Reserved for the assembler. Not preserved across procedure calls.
`$29` or `$gp`	`gp`	Contains the global pointer. Not preserved across procedure calls.
`$30` or `$sp`	`sp`	Contains the stack pointer. Preserved across procedure calls.
`$31`	zero	Always has the value 0.

Table 6-2: Floating-Point Registers

Register Name	Use
`$f0-f1`	Used to hold floating-point type function results (`$f0`) and complex type function results (`$f0` has the real part, `$f1` has the imaginary part). Not preserved across procedure calls.
`$f2-f9`	Saved registers. Preserved across procedure calls.
`$f10-f15`	Temporary registers used for expression evaluation. Not preserved across procedure calls.
`$f16-f21`	Used to pass the first six single or double precision actual arguments. Not preserved across procedure calls.
`$f22-f30`	Temporary registers used for expression evaluations. Not preserved across procedure calls.
`$f31`	Always has the value 0.0.

6.3.2 Using Directives to Control Sections and Location Counters

Assembled code and data are stored in the object file sections shown in Figure 6-1. Each section has an implicit location counter that begins at zero and increments by one for each byte assembled in the section. Location control directives (`.align`, `.data`, `.rdata`, `.sdata`, `.space`, and `.text`) can be used to control what is stored in the various sections and to adjust location counters.

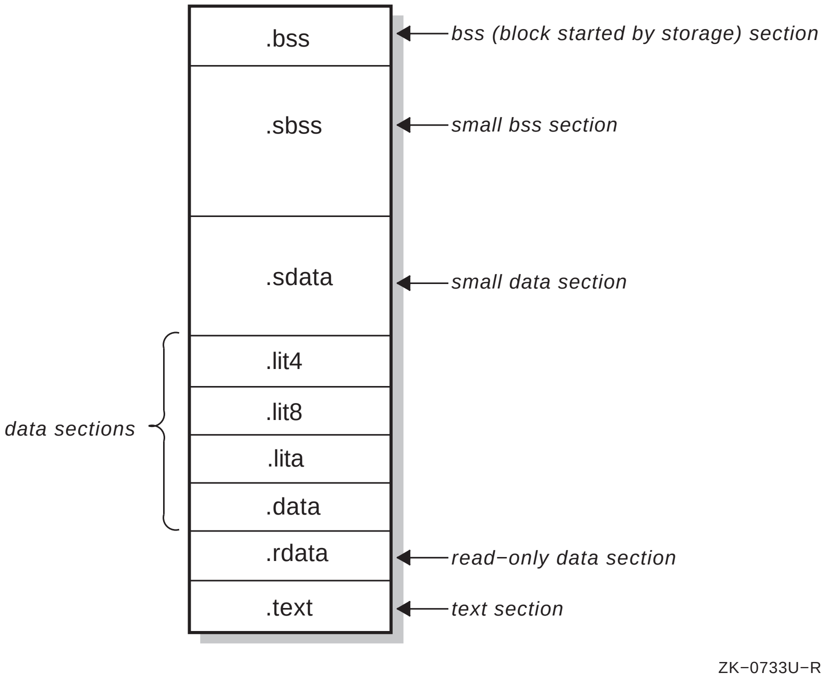
The assembler always generates the text section before other sections. Additions to the text section are done in 4-byte units.

The bss section holds zero-initialized data. If a `.lcomm` directive defines a variable, the assembler assigns that variable to either the bss (block started by symbol) section or the sbss (small bss) section, depending on the variable's size.

The default size for sbss variables is eight or fewer bytes. You can change the size using the `-G` compilation option for the C compiler or the assembler. Items smaller than or equal to the specified size go in sbss. Items greater than the specified size go in the bss section.

At run time, the `$gp` register points into the area of memory occupied by the `.lita` section. The `.lita` section is used to hold address literals for 64-bit addressing.

Figure 6-1: Section and Location Counters



See Chapter 7 for more information on section data.

6.3.3 The Stack Frame

The C compiler classifies each procedure into one of the following categories:

- *Nonleaf procedures.* These procedures call other procedures.
- *Leaf procedures.* These procedures do not themselves call other procedures. Leaf procedures are of two types: those that require stack storage for local variables and those that do not.

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

To write a program with proper stack frame usage and debugging capabilities, you should observe the conventions presented in the following list of steps. Steps 1 through 6 describe what must be done on procedure entry, step 7 describes how to pass parameters, and steps 8 through 12 describe what must be done on procedure exit:

1. Regardless of the type of procedure, you should include a `.ent` directive and an entry label for the procedure:

   ```
           .ent    procedure_name
   procedure_name:
   ```

   The `.ent` directive generates information for the debugger, and the entry label is the procedure name.

2. If you are writing a procedure that references static storage, calls other procedures, uses constants greater than 31 bits in size, or uses floating constants, you must load the `gp` register with the global pointer value for the procedure:

   ```
           ldgp    $gp,0($27)
   ```

   Register `$27` contains the procedure value (the address of this procedure as supplied by the caller).

3. If you are writing a leaf procedure that does not use the stack, skip to step 4. For a nonleaf procedure or a leaf procedure that uses the stack, you must adjust the stack size by allocating all of the stack space that the procedure requires:

   ```
           lda     $sp,-framesize($sp)
   ```

   The *`framesize`* operand is the size of frame required, in bytes, and must be a multiple of 16. You must allocate space on the stack for the following items:

   - Local variables.
   - Saved general registers. Space should be allocated only for those registers saved. For nonleaf procedures, you must save register `$26`, which is used in the calls to other procedures from this procedure. If you use registers `$9` to `$15`, you must also save them.

- Saved floating-point registers. Space should be allocated only for those registers saved. If you use registers `$f2` to `$f9`, 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 procedure; this does not include space for the first six arguments, which are always passed in registers.

Note

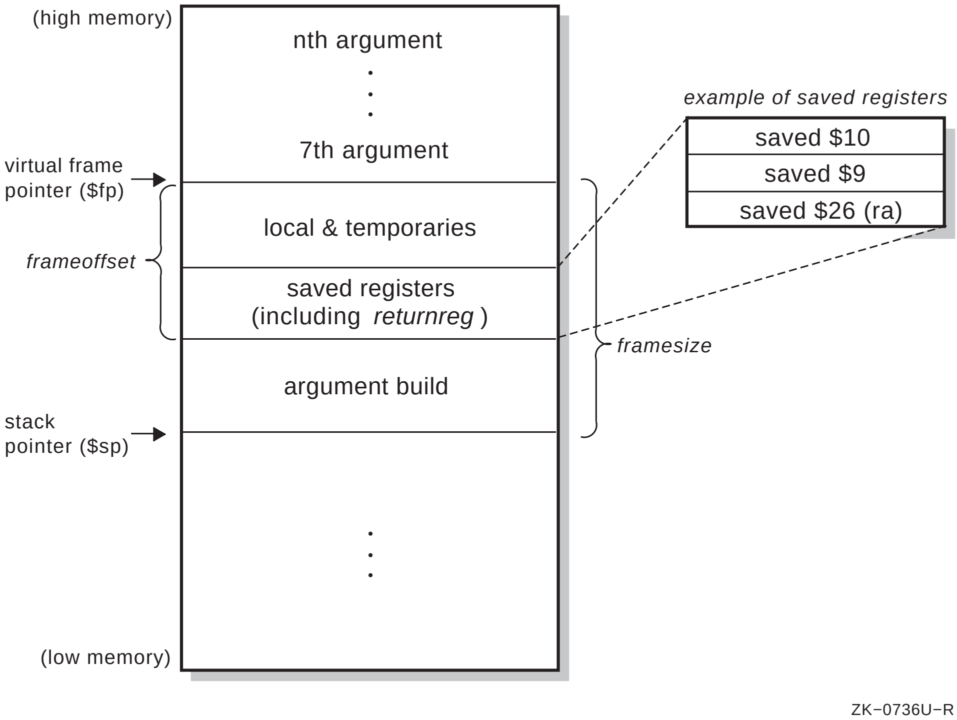
Once you have modified register `$sp`, you should not modify it again in the remainder of the procedure.

4. To generate information used by the debugger and exception handler, you must include a `.frame` directive:

```
.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 3). Figure 6-2 illustrates the stack components.

Figure 6-2: Stack Organization



The *returnreg* argument for the `.frame` directive specifies the register that contains the return address (usually register `$26`). These usual values may change if you use a varying stack pointer or are specifying a kernel trap procedure.

5. If the procedure is a leaf procedure that does not use the stack, skip to step 11. Otherwise, you must save the registers that you allocated space for in step 3.

   Saving the general registers requires the following operations:

   - Specify which registers are to be saved using the following `.mask` directive:

     ```
     .mask   bitmask,frameoffset
     ```

     The bit setting in *bitmask* indicate which registers are to be saved. For example, if register `$9` is to be saved, bit 9 in *bitmask* must be set to 1. The value for *frameoffset* is the offset (negative) from the virtual frame pointer to the start of the register save area.

   - Use the following `stq` instruction to save the registers specified in the `mask` directive:

```
stq     reg,framesize+frameoffset+N($sp)
```

The value of *N* is the size of the argument build area for the first register and is incremented by 8 for each successive register. If the procedure is a nonleaf procedure, the return address is the first register to be saved. For example, a nonleaf procedure that saves register `$9` and `$10` would use the following `stq` instructions:

```
stq     $26,framesize+frameoffset($sp)
stq     $9,framesize+frameoffset+8($sp)
stq     $10,framesize+frameoffset+16($sp)
```

(Figure 6-2 illustrates the order in which the registers in the preceding example would be saved.)

Then, save any floating-point registers that you allocated space for in step 3:

```
.fmask  bitmask,frameoffset
stt     reg,framesize+frameoffset+N($sp)
```

Saving floating-point registers is identical to saving integer registers except you use the `.fmask` directive instead of `.mask`, and the storage operations involve floating-point single- or double-precision data. (The previous discussion about how to save integer registers applies here as well.)

6. The final step in creating the procedure's prologue is to mark its end as follows:

   ```
   .prologue flag
   ```

   The *`flag`* is set to 1 if the prologue contains an `ldgp` instruction (see step 2); otherwise, it is set to 0.

7. This step describes parameter passing: how to access arguments passed into your procedure and how to pass arguments correctly to other procedures. For information on high-level language specific constructs (call-by-name, call-by-value, string or structure passing), see the programmer's guides for the high-level languages you are attempting to interface with.

   General registers `$16` to `$21` and floating-point registers `$f16` to `$f21` are used for passing the first six arguments. If any of the first six arguments are nonfloating-point arguments, they are passed in general registers. If any of the first six arguments are floating-point arguments, they are passed in floating-point registers.

   Stack space is used for passing the seventh and subsequent arguments. The stack space allocated to each argument is an 8-byte multiple and is aligned on an 16-byte boundary.

Table 6-3 summarizes the location of procedure arguments in the register or stack.

Table 6-3: Argument Locations

Argument Number	Integer Register	Floating-Point Register	Stack
1	$16 (a0)	$f16	
2	$17 (a1)	$f17	
3	$18 (a2)	$f18	
4	$19 (a3)	$f19	
5	$20 (a4)	$f20	
6	$21 (a5)	$f21	
7-*n*			0($sp)..(*n*-7)*8($sp)

8. On procedure exit, you must restore registers that were saved in step 5. To restore general purpose registers:

```
ldq     reg,framesize+frameoffset+N($sp)
```

To restore the floating-point registers:

```
ldt     reg,framesize+frameoffset+N($sp)
```

(Refer to step 5 for a discussion of the value of *N*.)

9. Get the return address:

```
ldq     $26,framesize+frameoffset($sp)
```

10. Clean up the stack:

```
lda     $sp,framesize($sp)
```

11. Return:

```
ret     $31,($26),1
```

12. End the procedure:

```
.end    procedurename
```

6.3.4 Examples

The following examples show procedures written in C and equivalent procedures written in assembly language.

Example 6-1 shows a nonleaf procedure. Notice that it creates a stackframe and saves its return address. It saves its return address because it must put a new return address into register `$26` when it makes a procedure call.

Example 6-1: Nonleaf Procedure

```
int
nonleaf(i, j)
  int i, *j;
  {
  int abs();
  int temp;

  temp = i - *j;
  return abs(temp);
  }

        .globl  nonleaf
 #    1 int
 #    2 nonleaf(i, j)
 #    3   int i, *j;
 #    4   {
        .ent    nonleaf 2
nonleaf:
        ldgp    $gp, 0($27)
        lda     $sp, -16($sp)
        stq     $26, 0($sp)
        .mask   0x04000000, -16
        .frame  $sp, 16, $26, 0
        .prologue      1
        addl    $16, 0, $18
 #    5   int abs();
 #    6   int temp;
 #    7
 #    8   temp = i - *j;
        ldl     $1, 0($17)
        subl    $18, $1, $16
 #    9   return abs(temp);
        jsr     $26, abs
        ldgp    $gp, 0($26)
        ldq     $26, 0($sp)
        lda     $sp, 16($sp)
        ret     $31, ($26), 1
        .end    nonleaf
```

Example 6-2 shows a leaf procedure that does not require stack space for local variables. Notice that it does not create a stackframe and does not save a return address.

Example 6-2: Leaf Procedure Without Stack Space for Local Variables

```
int
leaf(p1, p2)
  int p1, p2;
  {
  return (p1 > p2) ? p1 : p2;
  }

        .globl  leaf
 #    1 leaf(p1, p2)
 #    2   int p1, p2;
 #    3   {
        .ent    leaf 2
leaf:
        ldgp    $gp, 0($27)
        .frame  $sp, 0, $26, 0
        .prologue       1
        addl    $16, 0, $16
        addl    $17, 0, $17
 #    4   return (p1 > p2) ? p1 : p2;
        bis     $16, $16, $18
        .set     noat
        cmplt   $18, $17, $28
        cmovne  $28, $17, $18
        .set     at
        bis     $18, $18, $0
        ret     $31, ($26), 1
        .end    leaf
```

Example 6-3 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.

Example 6-3: Leaf Procedure With Stack Space for Local Variables

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

        .globl  leaf_storage
 #    1 int
 #    2 leaf_storage(i)
 #    3   int i;
```

Example 6-3: (continued)

```
 #    4   {
        .ent    leaf_storage 2
leaf_storage:
        ldgp    $gp, 0($27)
        lda     $sp, -80($sp)
        .frame  $sp, 80, $26, 0
        .prologue       1
        addl    $16, 0, $1
 #    5   int a[16];
 #    6   int j;
 #    7   for (j = 0; j < 10; j++)
        ldil    $2, 48
        stl     $2, 16($sp)
        ldil    $3, 49
        stl     $3, 20($sp)
        ldil    $0, 2
        lda     $16, 24($sp)
$32:
 #    8     a[j] = '0' + j;
        addl    $0, 48, $4
        stl     $4, 0($16)
        addl    $0, 49, $5
        stl     $5, 4($16)
        addl    $0, 50, $6
        stl     $6, 8($16)
        addl    $0, 51, $7
        stl     $7, 12($16)
        addl    $0, 4, $0
        addq    $16, 16, $16
        subq    $0, 10, $8
        bne     $8, $32
 #    9   return a[i];
        mull    $1, 4, $22
        addq    $22, $sp, $0
        ldl     $0, 16($0)
        lda     $sp, 80($sp)
        ret     $31, ($26), 1
        .end    leaf_storage
```

6.4 Developing Code for Procedure Calls

The rules and parameter requirements for passing control and exchanging data between procedures written in assembly language and procedures written in other languages are varied and complex. The simplest approach to coding an interface between an assembly procedure and a procedure written in a high-level language is to do the following:

- Use the high-level language to write a skeletal version of the procedure 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, using the code in the listing as a guideline, write your assembly-language code.

Section 6.4.1 and Section 6.4.2 describe techniques you can use to create interfaces between procedures written in assembly language and procedures written in a high-level language. The examples show 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. In writing your particular interface, you should write and compile realistic examples of the code you want to write in assembly language.

6.4.1 Calling a High-Level Language Procedure

This section describes the steps you could use in writing an assembly language procedure that calls `atof`(3), a procedure written in C that converts ASCII characters to numbers. The steps are as follows:

1. Write a C program that calls `atof.` Pass global variables instead of local variables; this makes them easy to recognize in the assembly-language version of the C program (and ensures that optimization does not remove any of the code on the grounds that it has no effect).

   The following C program is an example of a program that calls `atof`:

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

2. Compile the program using the following compiler options:

```
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, examine the file `caller.s` (shown here). The comments in the listing show how the parameters are passed, the

execution of the call, and how the returned values are retrieved:

```
        .globl  c
        .data
c:
        .ascii  "3.1415\X00"
        .comm   d 8
        .comm   f 4
        .text
        .globl  caller
 #    1 char c[] = "3.1415";
 #    2 double d, atof();
 #    3 float f;
 #    4 caller()
 #    5   {
        .ent    caller 2
caller:
        ldgp    $gp, 0($27)
        lda     $sp, -16($sp)
        stq     $26, 0($sp)
        .mask   0x04000000, -16
        .frame  $sp, 16, $26, 0
        .prologue       1
 #    6   d = atof(c);
        lda     $16, c
        jsr     $26, atof
        ldgp    $gp, 0($26)
        stt     $f0, d
 #    7   f = (float)atof(c);
        lda     $16, c
        jsr     $26, atof
        ldgp    $gp, 0($26)
        cvtts   $f0, $f10
        sts     $f10, f
 #    8   }
        ldq     $26, 0($sp)
        lda     $sp, 16($sp)
        ret     $31, ($26), 1
        .end    caller
```

6.4.2 Calling an Assembly-Language Procedure

This section shows a technique to follow in writing an assembly language procedure that can be called by a procedure written in a high-level language. The technique consists of the following steps:

1. Using a high-level language, write a facsimile of the assembly-language procedure you want to call. In the body of the procedure, write statements that use the same arguments you intend to use in the final assembly-language procedure. Copy the arguments to global variables instead of local variables to make it easy for you to read the resulting assembly-language listing.

The following C program is a facsimile of the assembly-language program:

```
typedef char str[10];
typedef int boolean;

float global_r;
int global_i;
str global_s;
boolean global_b;

boolean callee(float *r, int i, str s)
  {
  global_r = *r;
  global_i = i;
  global_s[0] = s[0];
  return i == 3;
  }
```

2. Compile the program using the following compiler options:

   ```
   cc -S -O callee.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, examine the file `callee.s` (shown here). The comments in the listing shows how the parameters are passed, the execution of the call, and how the returned values are retrieved:

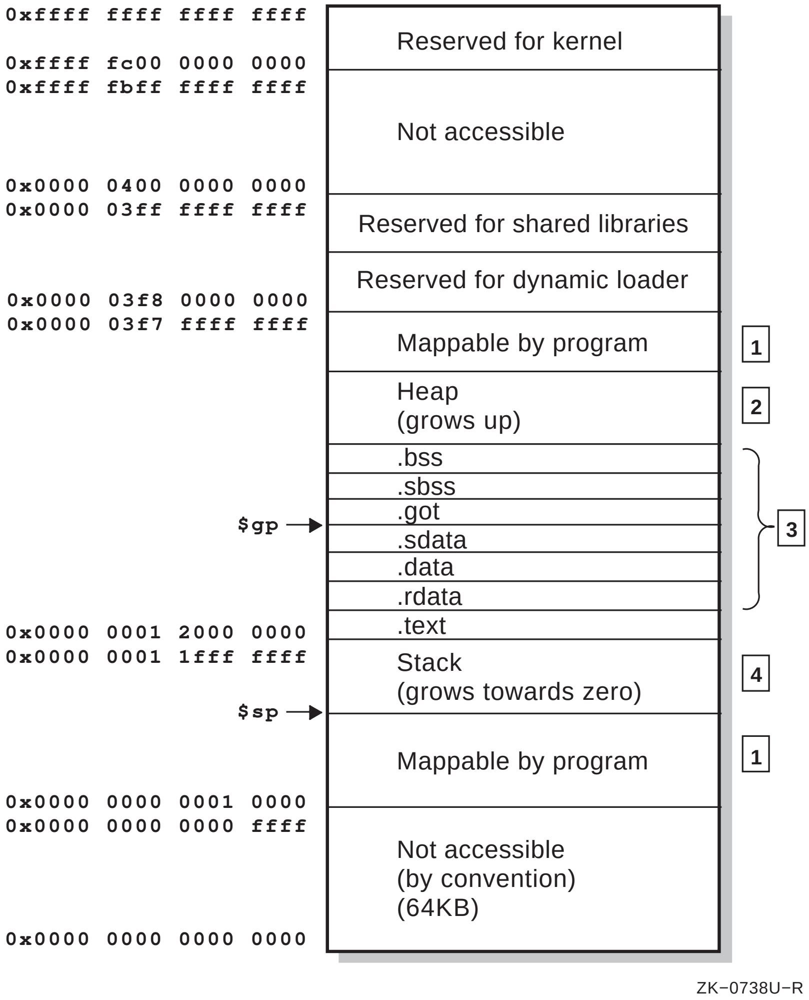
   ```
           .comm   global_r 4
           .comm   global_i 4
           .comm   global_s 10
           .comm   global_b 4
           .text
           .globl  callee
    #   10   {
           .ent    callee 2
   callee:
           ldgp    $gp, 0($27)
           .frame  $sp, 0, $26, 0
           .prologue       1
           addl    $17, 0, $17
    #   11   global_r = *r;
           lds     $f10, 0($16)
           sts     $f10, global_r
    #   12   global_i = i;
           stl     $17, global_i
    #   13   global_s[0] = s[0];
           ldq_u   $1, 0($18)
           extbl   $1, $18, $1
           .set     noat
           lda     $28, global_s
           ldq_u   $2, 0($28)
   ```

```
        insbl   $1, $28, $3
        mskbl   $2, $28, $2
        bis     $2, $3, $2
        stq_u   $2, 0($28)
        .set     at
#   14    return i == 3;
        cmpeq   $17, 3, $0
        ret     $31, ($26), 1
        .end    callee
```

6.5 Memory Allocation

The default memory allocation scheme used by the Alpha AXP system 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 linker and assembler use the scheme shown in Figure 6-3.

Figure 6-3: Layout of Memory (User Program View)



1 Not allocated until a user requests it, as in System V shared memory regions.

2 The heap is reserved for `sbrk` and `brk` system calls, and it is not always present.

3 All data is stored in the following sections:

- bss (block started by symbol) – Uninitialized data with a size greater than the value specified by the `-G` command line option.

- sbss (small bss) – Data less than or equal to the `-G` command line option. (Eight is the default value for the `-G` option.)
- got (global offset table) – Address constants merged from `.lita` sections.
- sdata (small data) – Data initialized and specified for the `.sdata` section.
- data (data) – Data initialized and specified for the `.data` section.
- rdata (read-only data) – Data initialized and specified for the `.rdata` section.

4 Used for local data in C programs.

Object Files 7

This chapter provides details on how compiler system object files are formatted and processed.

The chapter addresses the following topics:

- The components that make up the object file and the differences between the object-file format used by the DEC OSF/1 compiler system and the System V common object file format (COFF). (See Section 7.1.)
- The headers and sections of the object file. (Detailed information is given on the logic followed by the assembler and linker in handling relocation entries.) (See Section 7.2.)
- The format of object files (OMAGIC, NMAGIC, and ZMAGIC). (See Section 7.3.)
- Information used by the system loader in loading object files at run time. (See Section 7.4.)
- Archive files. (See Section 7.5.)
- The symbols defined by the linker. (See Section 7.6.)

7.1 Object File Overview

The assembler and the linker generate object files that have sections ordered as shown in Figure 7-1. Sections in an object file that do not contain data are omitted, except for the file header, optional header, and section header, which are always present.

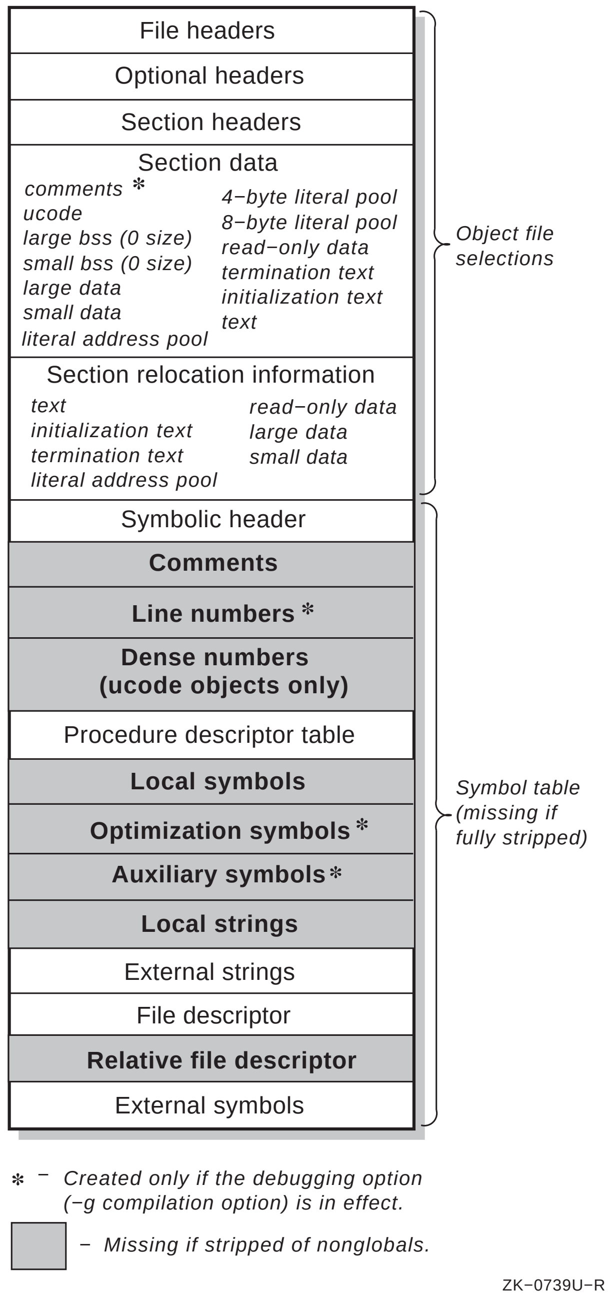
The sections of the symbol table (shown in Figure 7-1) that appear in the final object file can vary:

- The line numbers table, optimization symbols table, and auxiliary symbols table appear only when a debugging option is in effect (when the user specifies one of the `-g1`, `-g2`, or `-g3` compilation options).
- When you specify the `-x` option (strip nonglobals) for the link-edit phase, the linker updates the procedure descriptor table and strips the following tables from the object file: Line Number, Local Symbols, Optimization Symbols, Auxiliary Symbols, Local Strings, and Relative File Descriptor.

- The linker 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 linker designed to work with the compiler system should lay out the object file sections in the order shown in Figure 7-1. The linker can process object files that are ordered differently, but performance may be degraded.

Figure 7-1: Object File Format



The standard System V COFF (common object file format) differs from the DEC OSF/1 compiler system format in the following ways:

- The file header definition is based on the 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.
  - All tables that specify symbolic information have their file pointers and number of entries in the symbolic header. See Chapter 8 for information about the symbolic header.
- The definition of the optional header has the same format as specified in the System V header file `aouthdr.h`, except the following fields have been added: `bss_start`, `gprmask`, `fprmask`, and `gp_value` (see Table 7-4).
- The definition of the section header has the same format as the System V header file `scnhdr.h`, except the line number fields are used for global pointers (see Table 7-6).

The definition of the section relocation information is similar to UNIX 4.3 BSD, which has local relocation types. Section 7.2.5 provides information on differences between local and external relocation entries.

7.2 Object File Sections

The following sections describe the components of an object file. Headers are informational and provide the means for navigating the object file. Sections contain program instructions or data (or both).

7.2.1 File Header

The format of the file header is shown in Table 7-1. The file header and all of the fields described in this section are defined in `filehdr.h`.

Table 7-1: File Header Format

Declaration Type	Field	Description
`unsigned short`	`f_magic`	Target-machine magic number (see Table 7-2)
`unsigned short`	`f_nscns`	Number of sections
`int`	`f_timdat`	Time and date stamp
`long`	`f_symptr`	File pointer to symbolic header (see Chapter 8 for a description of the symbolic header)
`int`	`f_nsyms`	Size of symbolic header
`unsigned short`	`f_opthdr`	Size of optional header

Table 7-1: (continued)

Declaration Type	Field	Description
`unsigned short`	`f_flags`	Flags (see Table 7-3)

The magic number in the `f_magic` field in the file header specifies the target machine on which an object file can execute. Table 7-2 shows the octal values and mnemonics for the magic numbers.

Table 7-2: File Header Magic Numbers

Symbol	Value	Description
`ALPHAMAGIC`	0603	Machine-code object file
`ALPHAUMAGIC`	0617	Ucode object file

The `f_flags` field in the file header describes the object file characteristics. Table 7-3 lists the flags and gives their hexadecimal values and their meanings. The table notes those flags that do not apply to compiler system object files.

Table 7-3: File Header Flags

Symbol	Value	Description
`F_RELFLG`	0x0001	Relocation information stripped from file.
`F_EXEC`	0x0002	File is executable (that is, no unresolved external references).
`F_LNNO`	0x0004	Line numbers stripped from file.
`F_LSYMS`	0x0008	Local symbols stripped from file.
`F_MINMAL`[a]	0x0010	Minimal object file (`.m` file) output of `fextract`.
`F_UPDATE`[a]	0x0020	Fully bound update file, output of ogen.
`F_SWABD`[a]	0x0040	File whose bytes were swabbed (in names).
`F_AR16WR`[a]	0x0080	File has the byte ordering of an AR16WR machine (for example, PDP-11/70).
`F_AR32WR`[a]	0x0100	File has the byte ordering of an AR32WR machine (for example, VAX).
`F_AR32W`[a]	0x0200	File has the byte ordering of an AR32W machine (for example, 3b, maxi, MC68000).
`F_PATCH`[a]	0x0400	File contains ‘‘patch’’ list in optional header.
`F_NODF`[a]	0x0400	(Minimal file only.) No decision functions for replaced functions.
`F_MIPS_NO_SHARED`	0x1000	Cannot be dynamically shared.

Table 7-3: (continued)

Symbol	Value	Description
F_MIPS_SHARABLE	0x2000	A dynamically shared object.
F_MIPS_CALL_SHARED	0x3000	Dynamic executable.
F_MIPS_NO_REORG	0x4000	Do not reorder sections.
F_MIPS_NO_REMOVE	0x8000	Do not remove nops.

Table Note:

a. Not used by compiler system object modules.

7.2.2 Optional Header

The linker 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 Section 7.4.

Table 7-4 shows the format of the optional header (defined in the header file `aouthdr.h`).

Table 7-4: Optional Header Definitions

Declaration	Field	Description
short	magic	Object-file magic numbers (See Table 7-5)
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
int	gprmask	General-purpose register mask
int	fprmask	Floating-point register mask
long	gp_value	The gp value used for this object

Table 7-5 shows the octal values of the `magic` field for the optional header; the header file `aouthdr.h` contains the macro definitions.

Table 7-5: Optional Header Magic Numbers

Symbol	Value	Description
OMAGIC	0407	Impure Format. The text is not write-protected or shareable; 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. The Alpha AXP system provides write-protection for the text segment. (Other systems using COFF may not provide write-protection.) The object can be either dynamic or static.

See Section 7.3 for information on the format of OMAGIC, NMAGIC, and ZMAGIC files.

7.2.3 Section Headers

Table 7-6 shows the format of the section header (defined in the header file `scnhdr.h`).

Table 7-6: Section Header Format

Declaration	Field	Description
`char`	`s_name[8]`	Section name (see Table 7-7)
`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`	For `.pdata`, indicates the number of entries contained in the section; otherwise, reserved.
`unsigned short`	`s_nreloc`	Number of relocation entries
`unsigned short`	`s_nlnno`	Number of global pointer tables
`int`	`s_flags`	Flags (see Table 7-8)

Table 7-7 shows the defined section names for the `s_name` field of the section header.

Table 7-7: Section Header Constants for Section Names

Declaration	Field Contents	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
_LITA	".lita"	Literal address pool 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
_UCODE	".ucode"	ucode section
_GOT[a]	".got"	Global offset table
_DYNAMIC[a]	".dynamic"	Dynamic linking information
_DYNSYM[a]	".dynsym"	Dynamic linking symbol table
_REL_DYN[a]	".rel.dyn"	Relocation information
_DYNSTR[a]	".dynstr"	Dynamic linking strings
_HASH[a]	".hash"	Symbol hash table
_MSYM[a]	".msym"	Additional dynamic linking symbol table
_CONFLICT[a]	".conflict"	Additional dynamic linking information
_REGINFO[a]	".reginfo"	Register usage information
_XDATA	".xdata"	Exception scope table
_PDATA	".pdata"	Exception procedure table

Table Notes:

a. These sections exist only in ZMAGIC-type files and are used during dynamic linking.

Table 7-8 shows the defined hexadecimal values for the `s_flags` field. (Those flags that are not used by compiler system object files are noted in the table.)

Table 7-8: Format of s_flags Section Header Entry

Symbol	Value	Description
STYP_REG	0x00	Regular section: allocated, relocated, loaded
STYP_DSECT[a]	0x01	Dummy section: not allocated, relocated, not loaded

Table 7-8: (continued)

Symbol	Value	Description
STYP_NOLOAD[a]	0x02	Noload section: allocated, relocated, not loaded
STYP_GROUP[a]	0x04	Grouped section: formed of input sections
STYP_PAD[a]	0x08	Padding section: not allocated, not relocated, loaded STYP_COPY[a]~0x10~T{ Copy section (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	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_GOT[b]	0x1000	Global offset table
STYP_DYNAMIC[b]	0x2000	Dynamic linking information
STYP_DYNSYM[b]	0x4000	Dynamic linking symbol table
STYP_REL_DYN[b]	0x8000	Dynamic relocation information
STYP_DYNSTR[b]	0x10000	Dynamic linking symbol table
STYP_HASH[b]	0x20000	Dynamic symbol hash table
STYP_MSYM[b]	0x80000	Additional dynamic linking symbol table
STYP_CONFLICT[b]	0x100000	Additional dynamic linking information
STYP_REGINFO[b]	0x200000	Register usage information
STYP_FINI	0x01000000	.fini section text
STYP_COMMENT	0x02000000	Comment section
STYP_XDATA	0x02400000	Exception scope table
STYP_PDATA	0x02800000	Exception procedure table
STYP_LITA	0x04000000	Address literals only
STYP_LIT8	0x08000000	8-byte literals only
STYP_LIT4	0x10000000	4-byte literals only
S_NRELOC_OVFL	0x20000000	`s_nreloc` overflowed, the value is in `r_vaddr` of the first entry
STYP_INIT	0x80000000	Section initialization text only

Table Notes:

a. Not used by compiler system object modules.

b. These sections exist only in ZMAGIC type files and are used during dynamic linking.

The `S_NRELOC_OVFL` flag 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.

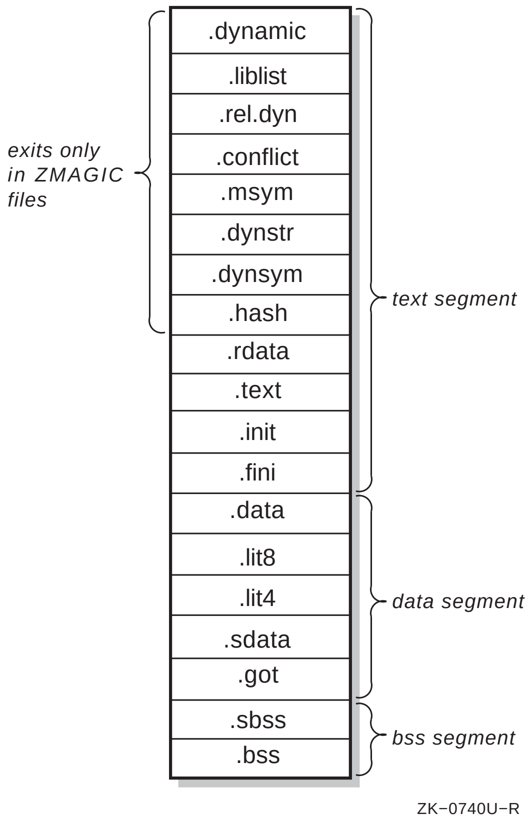
Note

For performance reasons, the linker uses the `s_flags` entry instead of `s_name` to determine the type of section. However, the linker does correctly fill in the `s_name` entry.

7.2.4 Section Data

Object files contain instructions and data. The instructions and data are segregated into sections according to their use. Figure 7-2 shows the layout of section data in object files.

Figure 7-2: Organization of Section Data



The `.dynamic`, `.liblist`, `.rel.dyn`, `.conflict`, `.msym`, `.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 9. The following table describes the uses of the other sections:

Section Name	**Use**
`.ucode`	Intermediate code
`.bss`	Block started by symbol
`.sbss`	Small block started by symbol
`.sdata`	Small data
`.lit4`	4-byte literal pool
`.lit8`	8-byte literal pool
`.data`	Data
`.fini`	Process termination text
`.init`	Initialization text

Section Name	Use
`.text`	Machine instructions to be executed
`.rdata`	Read-only data

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.

As indicated in Figure 7-2, the sections are grouped into segments:

- The text segment contains the `.rdata`, `.text`, `.init`, and `.fini` sections in all files except ZMAGIC files, which contain additional sections. (The `rdata` section can go in either the text or data segment, depending on the object file type.)
- The data segment contains the `.rdata`, `.data`, `.lit8`, `.lit4`, `.sdata`, and `.got` sections. (The `.got` section appears as a `.lita` section in static objects.)
- The bss segment contains the `.sbss` and `.bss` sections.

A section is described by and referenced through the section header (see Section 7.2.3); the optional header (see Section 7.2.2) provides the same information for segments.

The linker references the data shown in Figure 7-2 as both 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.

7.2.5 Section Relocation Information

Program instructions and data may contain addresses that must be adjusted when the object file is linked. Relocations locate the addresses within the section and indicate how they are to be adjusted.

7.2.5.1 Relocation Table Entry

Table 7-9 shows the format of an entry in the relocation table (defined in the header file `reloc.h`).

Table 7-9: Format of a Relocation Table Entry

Declaration	Field	Description
long	r_vaddr	Address (virtual) of an item to be relocated.
unsigned	r_symndx	For an external relocation entry, r_symndx is an index into external symbols. For a local relocation entry, r_symndx is the number of the section containing the symbol.
unsigned	r_type:8	Relocation type (see Table 7-11).
unsigned	r_extern:1	Set to 1 for an external relocation entry. Set to 0 for a local relocation entry.
unsigned	r_offset:6	For R_OP_STORE, r_offset is the bit offset of a field within a quadword.
unsigned	r_reserved:11	Must be zero.
unsigned	r_size:6	For R_OP_STORE, r_size is the bit size of a field.

The setting of r_extern and the contents of r_symndx vary for external and local relocation entries:

- For external relocation entries, r_extern is set to 1 and r_symndx is the index into external symbols. In this case, the value of the symbol is used as the value for relocation (see Figure 7-3).
- For local relocation entries, r_extern is set to 0, and r_symndx contains a constant that refers to a section (see Figure 7-4). In this case, the starting address of the section to which the constant refers is used as the value for relocation.

Table 7-10 gives the section numbers for r_symndx; the reloc.h file contains the macro definitions.

Table 7-10: Section Numbers for Local Relocation Entries

Symbol	Value	Description
R_SN_TEXT	1	.text 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_INIT	7	.init section

Table 7-10: (continued)

Symbol	Value	Description
R_SN_LIT8	8	.lit8 section
R_SN_LIT4	9	.lit4 section
R_SN_XDATA	10	.xdata section
R_SN_PDATA	11	.pdata section
R_SN_FINI	12	.fini section
R_SN_LITA	13	.lita section
R_SN_ABS	14	for R_OP_*xxxx* constants

Table 7-11 shows valid symbolic entries for the r_type field (defined in the header file reloc.h).

Table 7-11: Relocation Types

Symbol	Value	Description
R_ABS	0x0	Relocation already performed.
R_REFLONG	0x1	32-bit reference to the symbol's virtual address.
R_REFQUAD	0x2	64-bit reference to the symbol's virtual address.
R_GPREL32	0x3	32-bit displacement from the global pointer to the symbol's virtual address.
R_LITERAL	0x4	Reference to a literal in the literal address pool as an offset from the global pointer.
R_LITUSE	0x5	Identifies usage of a literal address previously loaded into a register. The r_symndx field identifies the specific usage of the register. Table 7-12 lists the valid usage types.
R_GPDISP	0x6	Identifies an lda/ldah instruction pair that is used to initialize a procedure's global-pointer register. The r_vaddr field identifies one instruction of the pair. The r_symndx contains a byte offset, which when added to the r_vaddr field, results in the address of the other instruction of the pair.
R_BRADDR	0x7	21-bit branch reference to the symbol's virtual address.
R_HINT	0x8	14-bit jsr hint reference to the symbol's virtual address.
R_SREL16	0x9	16-bit self-relative reference to the symbol's virtual address.
R_SREL32	0xa	32-bit self-relative reference to the symbol's virtual address.
R_SREL64	0xb	64-bit self-relative reference to the symbol's virtual address.

Table 7-11: (continued)

Symbol	Value	Description
R_OP_PUSH	0xc	Push symbol's virtual address on relocation expression stack.
R_OP_STORE	0xd	Pop value from the relocation expression stack and store at the symbol's virtual address. The r_size field determines the number of bits stored. The r_offset field designates the bit offset from the symbol to the target.
R_OP_PSUB	0xe	Pop value from the relocation expression stack and substract the symbol's virtual address. The result is pushed on the relocation expression stack.
R_OP_PRSHIFT	0xf	Pop value from the relocation expression stack and shift right by the symbol's value. The result is pushed on the relocation expression stack.
R_GPVALUE	0x10	Specifies a new *gp* value is to be used starting with the address specified by the r_vaddr field. The *gp* value is the sum of the optional header's gp_value field and the r_symndx field. The r_extern field must be zero.

Table 7-12 shows valid symbolic entries for the symbol index (r_symndx) field for the relocation type R_LITUSE.

Table 7-12: Literal Usage Types

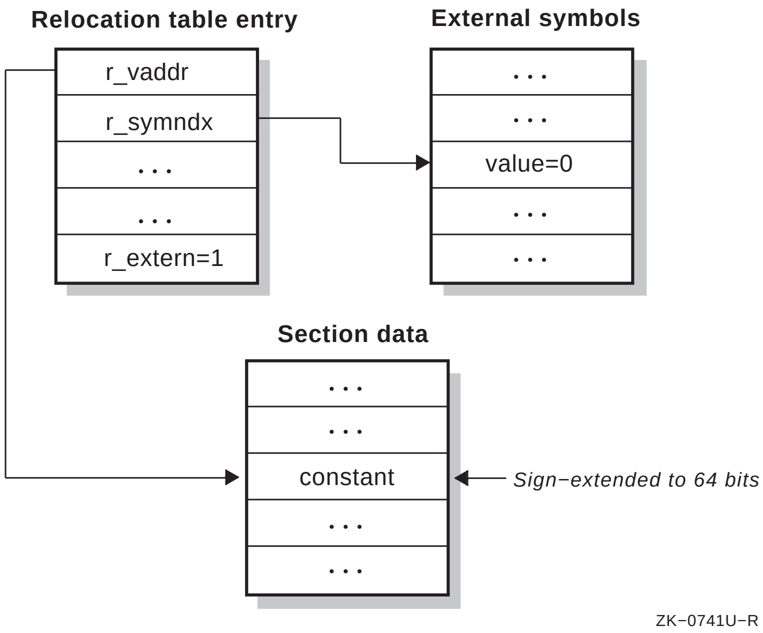
Symbol	Description
R_LU_BASE	A literal address is contained in the base register of a memory format instruction (except ldah).
R_LU_BYTOFF	A literal address is contained in the byte offset register (Rb) of a byte-manipulation instruction.
R_LU_JSR	A literal address is contained in the target register of a jsr instruction.

7.2.5.2 Assembler and Linker Processing of Relocation Entries

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

Local relocation entries must be used for symbols that are defined, and external relocation entries are used only for undefined symbols. Figure 7-3 gives an overview of the relocation table entry for an undefined external symbol.

Figure 7-3: Relocation Table Entry for Undefined External Symbols



The assembler creates this entry as follows:

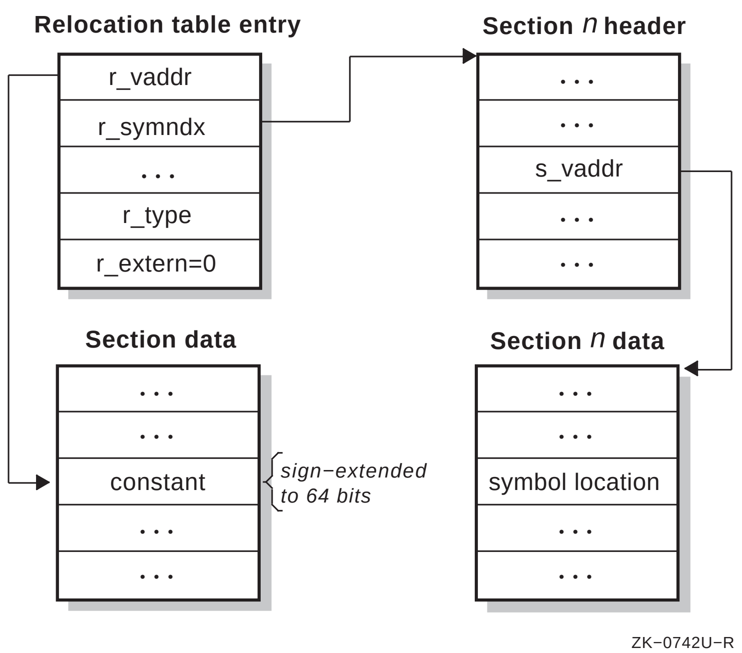
1. Sets `r_vaddr` to point to the item to be relocated.
2. Places a constant to be added to the value for relocation at the address for the item to be relocated (`r_vaddr`).
3. Sets `r_symndx` to the index of the external symbols entry that contains the symbol value (which is used as the value for relocation).
4. Sets `r_type` to the constant for the type of relocation types. Table 7-11 shows the valid constants for the relocation type.
5. 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. For relocation types other than `R_HINT`, the linker flags this as an error if the width of the constant is less than a full quadword and an overflow occurs after relocation.

When the linker determines that an external symbol is defined, it changes the relocation table entry for the symbol to a local relocation entry. Figure 7-4 gives an overview of the new entry.

Figure 7-4: Relocation Table Entry for a Local Relocation Entry



To change this entry from an external relocation entry to a local relocation entry, the linker performs the following steps:

1. Picks up the constant from the address to be relocated (`r_vaddr`).
2. If the width of the constant is less than 64 bits, sign-extends the constant to 64 bits.

3. Adds the value for relocation (the value of the symbol) to the constant and places it back in the address to be relocated.
4. Sets `r_symndx` to the section number that contains the external symbol.
5. Sets `r_extern` to 0.

The following list presents examples that show the use of external relocation entries:

– **Example 1: 64-Bit Reference — R_REFQUAD**

  This example shows assembly statements that set the value at location `b` to the global data value `y`.

```
    .globl y
    .data
b:  .quad y # R_REFQUAD relocation type at address b for
            # symbol y
```

  In processing this statement, the assembler generates a relocation entry of type `R_REFQUAD` for the address `b` and the symbol `y`. After determining the address for the symbol `y`, the loader adds the 64-bit address of `y` to the 64-bit value at location `b` and places the sum in location `b`.

  The loader handles 32-bit addresses (`R_REFLONG`) in the same manner, except it checks for overflow after determining the relocation value.

– **Example 2: 21-Bit Branch — R_BRADDR**

  This example shows assembly statements that call routine `x` from location `c`.

```
    .text
x:  #routine x
    ...
c:  bsr x # R_BRADDR relocation type at address c for symbol x
```

  In processing these statements, the assembler generates a relocation entry of type `R_BRADDR` for the address and the symbol `x`. After determining the address for the routine, the loader subtracts the address `c+4` to form the displacement to the routine. Then, the loader adds this result to the 21 low-order bits (sign-extended and multiplied by 4) of the instruction at address `c`, and after checking for overflow, places the result (divided by 4) back into the 21 low-order bits at address `c`.

  `R_BRADDR` relocation entries are produced for the assembler's `br` (branch) and `bsr` (branch subroutine) instructions.

If the entry is a local relocation type, the target of the branch instruction is assembled in the instruction at the address to be relocated. Otherwise, the instruction's displacement field contains a signed offset from the external symbol.

– **Example 3: 32-bit GP-Relative Reference — R_GPREL32**

This example shows assembly language statements that set the value at location a to the offset from the global pointer to the global data value z.

```
    .globl z
    .data
a:  .gprel32 z # R_GPREL32 relocation type at address a for
               # symbol z
```

In processing this statement, the assembler generates a relocation entry of type R_GPREL32 for the address a and the symbol z. After determining the address for the symbol z, the loader adds the 64-bit displacement of z from the the global pointer to the signed 32-bit value at location a, and places the sum in location a. The loader checks for overflow when performing the above operation.

– **Example 4: Literal Address Reference — R_LITERAL**

This example shows an assembly language statement that loads the address of the symbol y into register 22.

```
    lda $22, y
```

In processing this statement, the assembler generates the following code:

```
    .lita
x:  .quad y # R_REFQUAD relocation type at address x for
            # symbol y

    .text
h:  ldq $22, n($gp) # R_LITERAL relocation type at address h
                    # for symbol x
```

The assembler uses the difference between the address for the symbol x and the value of the global pointer as the value of the displacement (n) for the instruction. The linker gets the value of the global pointer used by the assembler from gp_value in the optional header (see Table 7-4).

– **Example 5: Literal Usage Reference — R_LITUSE**

This example shows an assembly language statement that loads the

32-bit value stored at address y into register 22.

```
ldl $22, y
```

In processing this statement, the assembler generates the following code:

```
    .lita
x:  .quad y # R_REFQUAD relocation type at address x for
            # symbol y

    .text
h:  ldq $at, n($gp) # R_LITERAL relocation type at address h
                    # for symbol x
i:  ldl $22, 0($at) # R_LITUSE relocation type at address i;
                    # r_symndx == R_LU_BASE
```

The assembler uses the difference between the address for the symbol x and the value of the global pointer as the value of the displacement (n) for the ldq instruction. The linker gets the value of the global pointer used by the assembler from gp_value in the optional header (see Table 7-4).

– **Example 6: GP Displacement Reference — R_GPDISP**

This example shows an assembly language statement that reloads the value of the global pointer after a call to procedure x.

```
# call to procedure x returns here with return address in ra
ldgp $gp, 0(ra)
```

In processing this statement, the assembler generates the following code:

```
j:  lda $at, <gp_disp>[0:15](ra) # R_GPDISP relocation type
                                 # at address j;
                                 # r_symndx contains byte offset
                                 # from address j to address k
k:  ldah $gp, <gp_disp>[16:31]($at)
```

The assembler determines the 32-bit displacement from the address of the ldgp instruction to the global pointer and stores it into the offset fields of the lda and ldah instructions. The linker gets the value of the global pointer used by the assembler from gp_value in the optional header (see Table 7-4).

– **Example 7: JSR Hint — R_HINT**

This example shows an assembly language statement that makes an indirect jump through register 24 and specifies to the branch-prediction logic that the target of the jsr is the address of the symbol x.

```
    # get address of procedure to call into register 24
m:  jsr ra, ($24), x # R_HINT relocation type at address m
                     # for symbol x
```

In processing this statement, the assembler generates a relocation entry of type R_HINT for the address m and the symbol x.

7.3 Object-File Formats (OMAGIC, NMAGIC, ZMAGIC)

This section describes the object-file formats created by the linker: Impure (OMAGIC), Shared Text (NMAGIC), and Demand Paged (ZMAGIC) formats. To understand this section, you should be familiar with the format and contents of the text, data, and bss segments as described in Section 7.2.4.

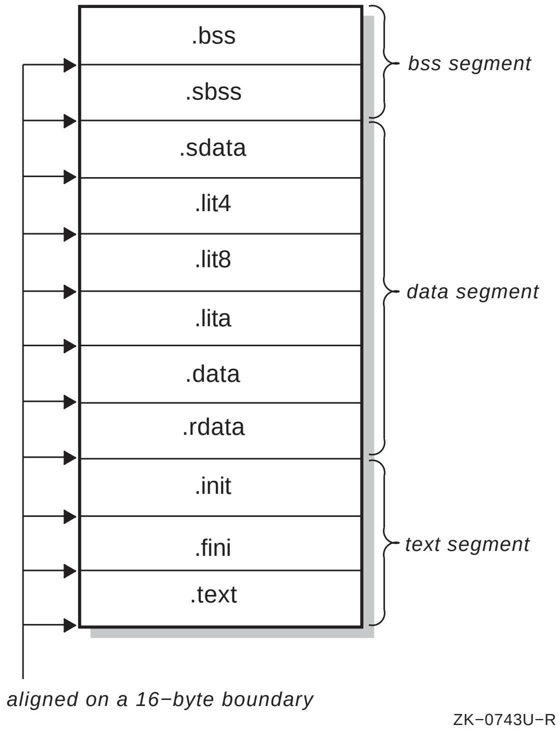
The following constraints are imposed on the address at which an object can be loaded and the boundaries of its segments; the operating system can dictate additional constraints.

- Segments must not overlap.
- Space should be reserved for the stack, which starts just below the base of the text segment and grows through lower addresses; that is, the value of each subsequent address is less than that of the previous address.
- For ZMAGIC and NMAGIC files, the default text segment address is 0x120000000, with the data segment starting at 0x140000000.
- For OMAGIC files, the default text segment address is 0x10000000, with the data segment following the text segment.
- For OMAGIC files, the `-B` *num* option (specifying a bss segment origin) cannot be specified; the default, which specifies that the bss segment is to follow the data segment, must be used.
- Segments must be aligned on 8-megabyte boundaries.

7.3.1 Impure Format (OMAGIC) Files

An OMAGIC file has the format shown in Figure 7-5.

Figure 7-5: Layout of OMAGIC Files in Virtual Memory



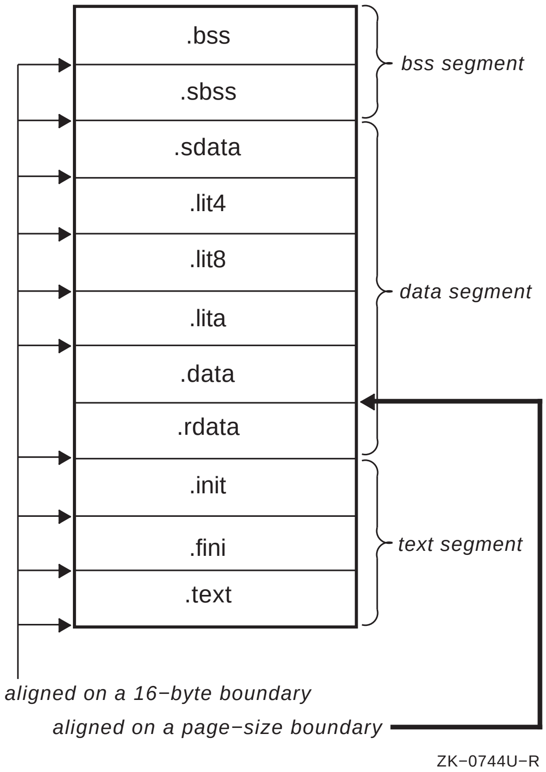
The OMAGIC format has the following characteristics:

- Each section follows the other in virtual address space aligned on a 16-byte boundary.
- The sections are not blocked.
- Text, data and bss segments can be placed anywhere in the virtual address space using the linker's `-T`, `-D` and `-B` options.
- The addresses specified for the segments must be rounded to 16-byte boundaries.

7.3.2 Shared Text (NMAGIC) Files

An NMAGIC file has the format shown in Figure 7-6.

Figure 7-6: Layout of NMAGIC Files in Virtual Memory



An NMAGIC file has the following characteristics:

- The virtual address of the `.data` section is on a *`pagesize`* boundary.
- The sections are not blocked.
- Each section follows the other in virtual address space aligned on a 16-byte boundary.
- Only the start of the text and data segments, using the linker'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.

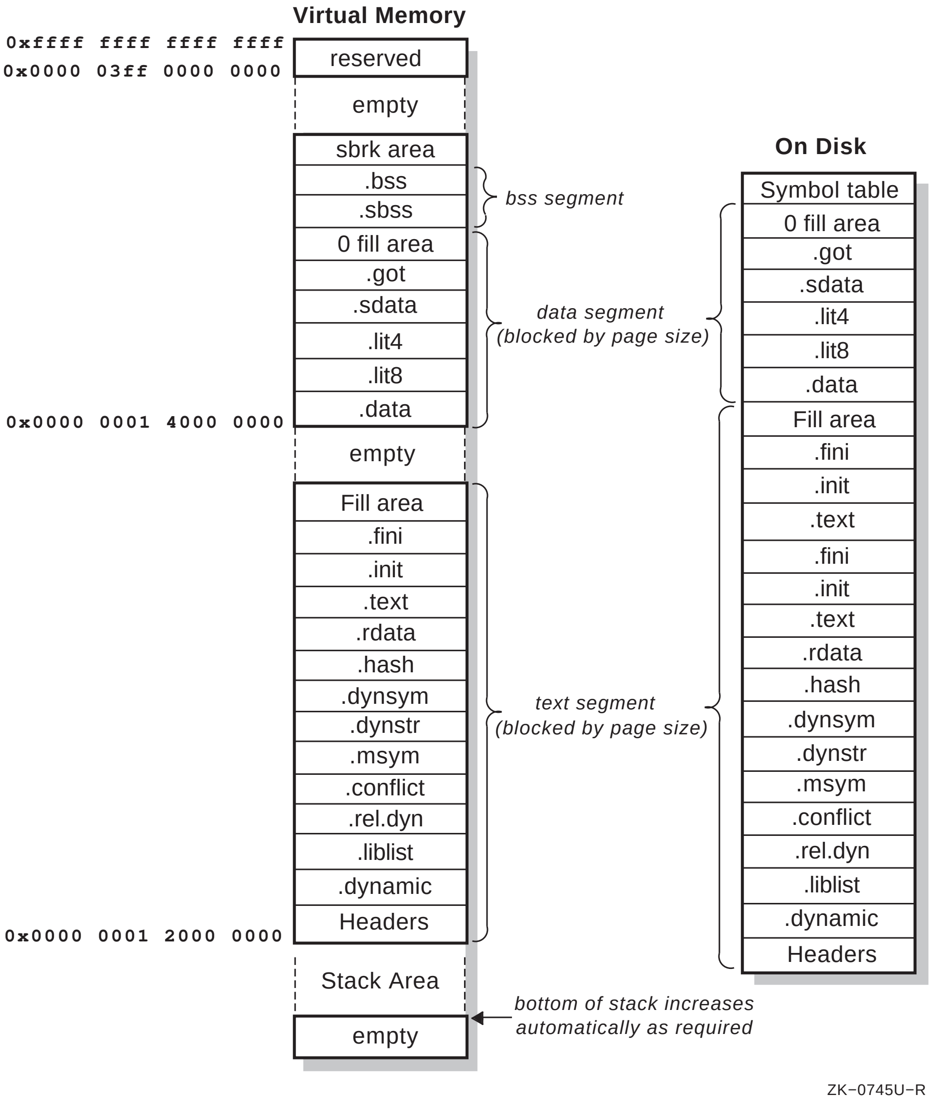
7.3.3 Demand Paged (ZMAGIC) Files

A ZMAGIC file is a demand paged file. Figure 7-7 shows the format of a ZMAGIC file as it appears in virtual memory and on disk.

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 (Table 7-1, Table 7-4, and Table 7-6, respectively) rounded to 16 bytes is included in blocking of the text segment.
- The text segment starts by default at 0x120000000.
- Only the start of the text and data segments, using the linker's `-T` and `-D` options can be specified for a demand paged format file and must be a multiple of the pagesize.

Figure 7-7: Layout of ZMAGIC Files



7.3.4 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.

7.4 Loading Object Files

The linker 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 a description of the sections and how they are formatted.

The sections are grouped into segments, which are described in the optional header. To load an object file for execution, the 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 the segments 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 following macro in the header file `a.out.h`:

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

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 7-4) 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 7-6) 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 linker 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 7-4).

The other fields in the optional header are `gprmask` and `fprmask`, 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 when a context-switch operation occurs.

7.5 Archive Files

The linker can link object files in archives created by the archiver. The archiver and the format of the archives are based on the System V portable archive format. To improve performance, the format of the archives symbol table was changed to 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 `ranhash`(3x). The archive format definition is in the header file `ar.h`.

7.6 Linker Defined Symbols

Certain symbols are reserved and their values are defined by the linker. A user program can reference these symbols, but cannot define them; an error is generated if a user program attempts to define one of these symbols. Table 7-13 lists the names and values of these symbols; the header file `sym.h` contains their preprocessor macro definitions.

Table 7-13: Linker Defined Symbols

Symbol	Value	Description
`_ETEXT`	"_etext"	First location after text segment.
`_EDATA`	"_edata"	First location after data segment.
`_END`	"_end"	First location after bss segment.
`_FTEXT`[a]	"_ftext"	First location of text segment.
`_FDATA`[a]	"_fdata"	First location of data segment.
`_FBSS`[a]	"_fbss"	First location of the bss segment.
`_GP`	"_gp"	gp value stored in optional header.
`_PROCEDURE_TABLE`	"_procedure_table"	Run-time procedure table.
`_PROCEDURE_TABLE_SIZE`	"_procedure_table_size"	Run-time procedure table size.
`_PROCEDURE_STRING_TABLE`	"_procedure_string_table"	String table for run-time procedure.
`_COBOL_MAIN`	"_cobol_main"	First COBOL main symbol.
`_WEAK_ETEXT`[b]	"etext"	Weak symbol for first location after text segment.
`_WEAK_EDATA`	"edata"	Weak symbol for first location after data segment.
`_WEAK_END`[b]	"end"	Weak symbol for first location after bss segment.
	"_BASE_ADDRESS"[c]	Base address of file.

Table 7-13: (continued)

Symbol	Value	Description
	"_DYNAMIC_LINK"[c]	1 if creating a dynamic executable, 0 otherwise.
	"_DYNAMIC"[c]	Address of .dynamic section.
	"_GOT_OFFSET"[c]	Address of .got section for dynamic executable.

Table Notes:

a. Compiler system only.

b. Not defined with `-std`.

c. No symbol entry. Not defined in `sym.h`.

The dynamic linker also reserves and defines certain symbols; see Chapter 9 for more information.

The first three symbols in Table 7-13 (`_ETEXT`, `_EDATA`, and `_END`) come from the standard UNIX system linker. The remaining symbols are compiler-system specific.

The linker 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 main routine.

The following symbols relate to the run-time procedure table:

– `_PROCEDURE_TABLE`

– `_PROCEDURE_TABLE_SIZE`

– `_PROCEDURE_STRING_TABLE`

The run-time procedure table is used by the exception systems in languages that have exception-handling capabilities built into them. 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 linker fills in `exception_info` with the address of the external table. Otherwise, it fills 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 run-time 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` describe the routines in that library.

Symbol Table 8

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

- Symbol Table Overview – this section gives the purpose of the Symbol Table, a summary of its components, and their relationship to each other.
- Format of Symbol Table Entries[1] – this section shows the structures of symbol table entries and the values you assign them through the symbol table routines.

8.1 Symbol Table Overview

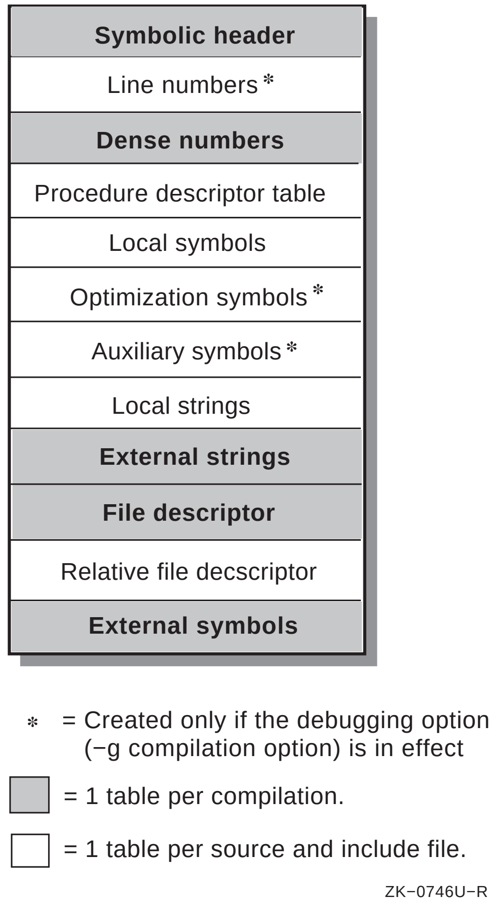
The symbol table is created by the compiler front-end as a stand-alone file. The purpose of the table is to provide information needed by the linker and the debugger to perform their respective functions. At the option of the user, the linker includes information from the symbol table in the final object file for use by the debugger. (See Figure 7-1 for details about object file format.)

[1] Third Eye Software, Inc. owns the copyright (dated 1984) to the format and nomenclature of the symbol table used by the compiler system as documented in this chapter.

Third Eye Software, Inc. grants reproduction and use rights to all parties, PROVIDED that this comment is maintained in the copy.

Third Eye makes no claims about the applicability of this symbol table to a particular use.

Figure 8-1: The Symbol Table Overview



The elements that make up the symbol table are shown in Figure 8-1. The compiler front-end creates one group of tables (the shaded areas in Figure 8-1) 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 linker interact with the symbol table as summarized in the following list:

– 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 and optimization symbols, and updates the local symbol table, external symbol table, and procedure descriptor table.

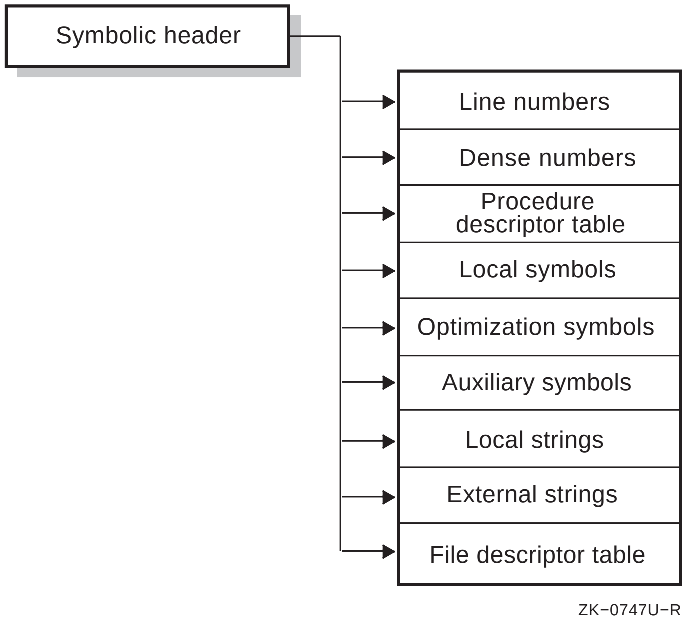
– The linker eliminates duplicate information in the external symbol table and the external string table, removes tables with duplicate information, updates the local symbol table with relocation information, and creates the relative file descriptor table.

The major elements of the symbol 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) contains the sizes and locations (as an offset from the beginning of the file) of the subtables that make up the symbol table. Figure 8-2 shows the symbolic relationship of the header to the other tables.

Figure 8-2: Functional Overview of the Symbolic Header



Line Number Table

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 instructions to the source lines and vice versa.

Dense Number Table

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 the local symbol table. The table facilitates symbol look-up for the assembler, optimizer, and code generator by allowing direct table access to be used instead of hashing.

Procedure Descriptor Table

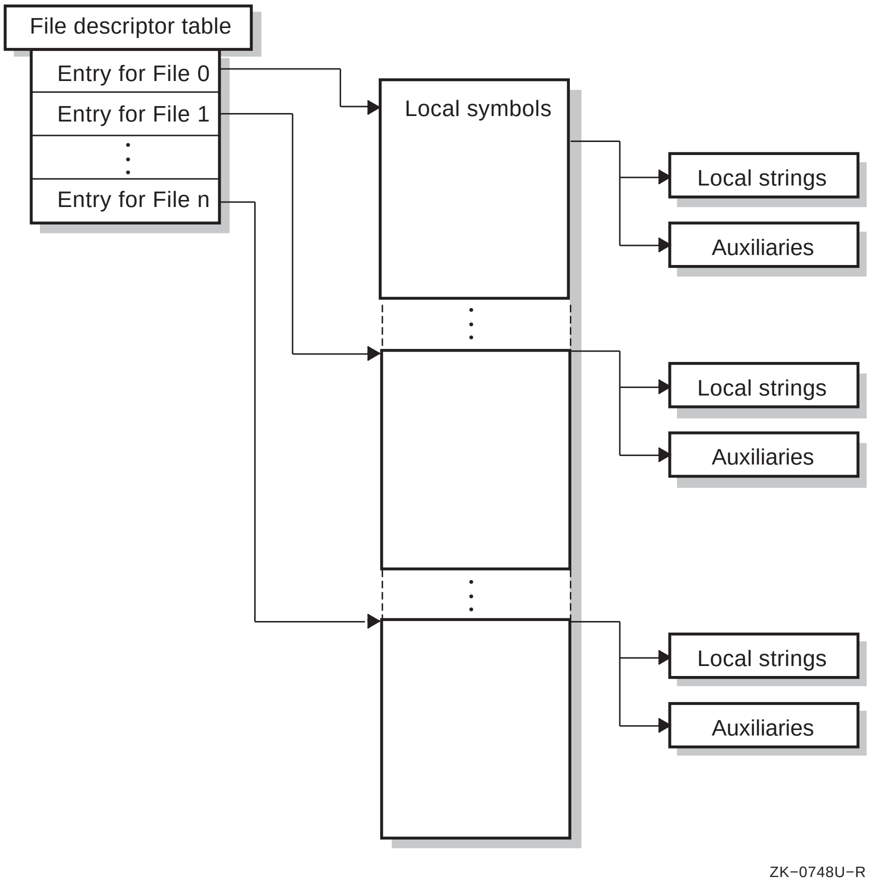
The procedure descriptor table contains register and frame information, and offsets into other tables that provide detailed information on the procedure. The compiler front-end creates the table and links it to the local symbol table. The assembler enters information on registers and frames. The debugger uses the entries in determining the line numbers for procedures and the frame information for stack traces.

Local Symbol Table

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

A unique local symbol 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 the local symbol table can reference related information in the local string table and auxiliary symbol table. This relationship is shown in Figure 8-3.

Figure 8-3: Logical Relationship Between the File Descriptor Table and Local Symbols



Optimization Symbol Table
To be defined at a future date.

Auxiliary Symbol Table
The auxiliary symbol tables contain data type information specific to one language. Each entry is linked to an entry in the Local Symbol Table. The entry in the local symbol table 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 one of the debugging options (`-g` compilation options) is in effect.

Local String Table
The local string tables contain the names of local symbols.

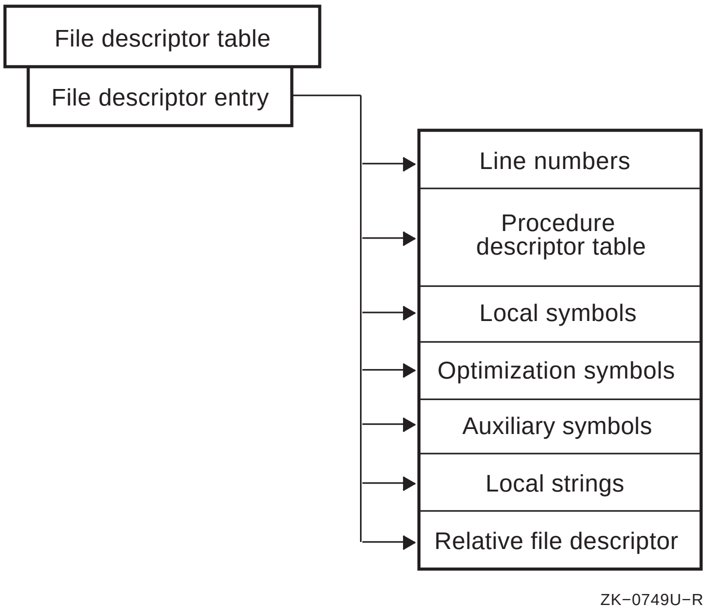
External String Table

The external string table contains the names of external symbols.

File Descriptor Table

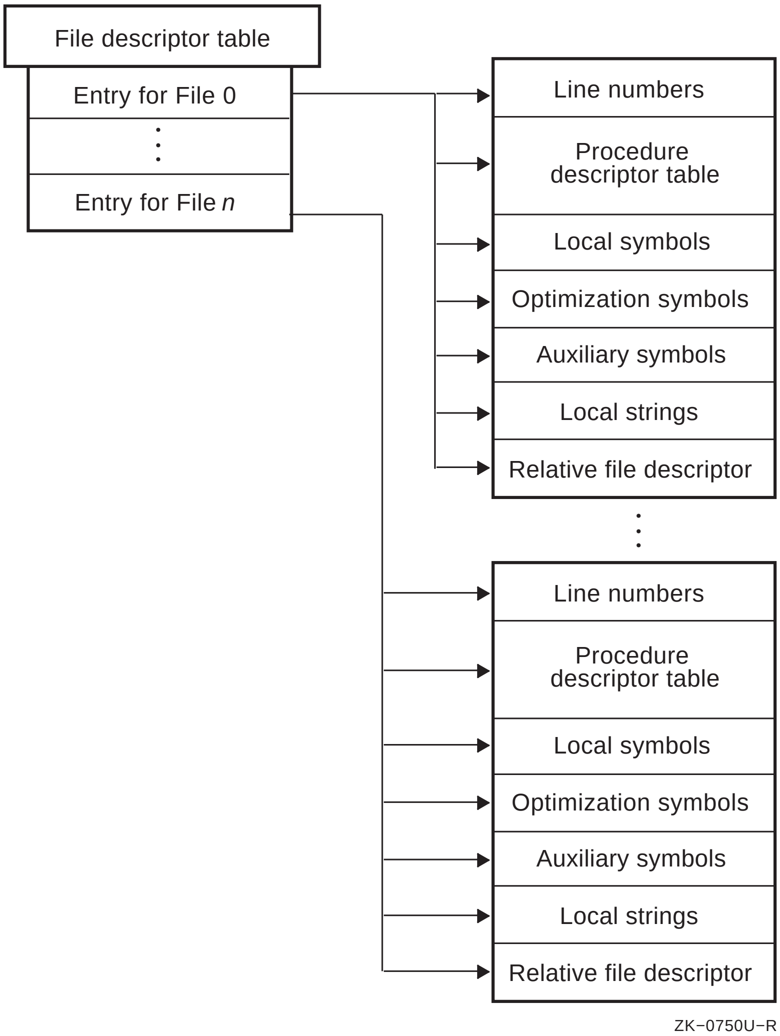
The file descriptor table contains one entry each for each source file and each of its include files. The entry is composed of pointers to a group of subtables related to the file. The structure of an entry is shown in Table 8-12, and the physical layout of the subtables is shown in Figure 8-4.

Figure 8-4: 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 the file descriptor table and its subtables is shown in Figure 8-5.

Figure 8-5: Logical Relationship Between the File Descriptor Table and Other Tables



Relative File Descriptor Table

Each file in the symbol table contains a relative file descriptor for each file it was compiled with (including itself and include files). The relative file descriptor maps the index of each file at compile time to its index after linking. All file indices inside the local symbols and auxiliary table must be mapped through the relative file descriptor table for the file they occur in. A missing file descriptor table implies the identity function.

External Symbol Table

The external symbol table contains global symbols entered by the compiler front-end. The symbols are defined in one module and referenced in one or more other modules. The assembler updates the entries, and the linker merges the symbols and resolves their addresses.

8.2 Format of Symbol Table Entries

The symbol table is comprised of several subtables. The symbolic header is the "directory" for the subtables; it locates the subtables and gives their sizes.

The following sections describe the symbolic header and the subtables.

8.2.1 Symbolic Header

The structure of the symbolic header is shown below in Table 8-1; the `sym.h` header file contains the header declaration.

Table 8-1: Format of the Symbolic Header

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

Table 8-1: (continued)

Declaration	Name	Description
long	cbExtOffset	Index to the start of external symbols

The lower byte of the vstamp field contains LS_STAMP and the upper byte contains MS_STAMP (see the stamp.h header file). These values are defined in the stamp.h file.

The iMax fields and the cbOffset fields must be set to zero if one of the tables shown in Table 8-1 is not present.

The magic field must contain the constant magicSym, which is also defined in symconst.h.

8.2.2 Line Number Table

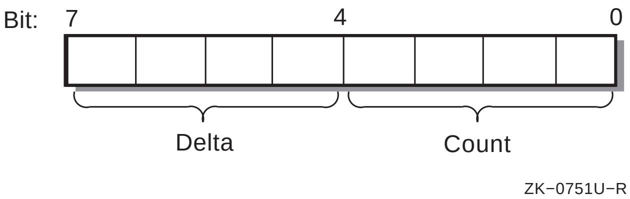
Table 8-2 shows the format of an entry in the line number table; the sym.h header file contains its declaration.

Table 8-2: Format of a Line Number Entry

Declaration	Name
int	LINER
int *	pLINER

The line number section in the symbol table is rounded to the nearest 4-byte boundary.

Line numbers map executable instructions to source lines; one line number is stored for each instruction associated with a source line. Line numbers are stored as integers in memory and in packed format on disk. The layout of a line number entry on disk is as follows:

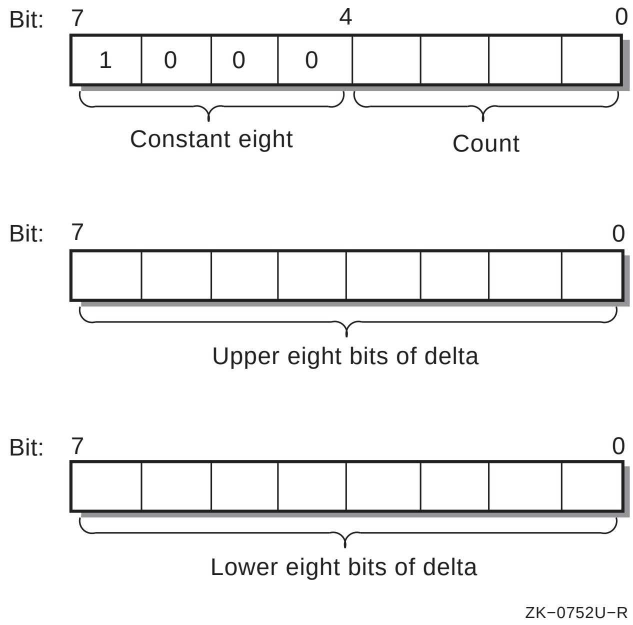


The compiler assigns a line number only to those lines of source code that generate executable instructions.

The uses of the delta and count fields are as follows:

- Delta is a 4-bit field with a value in the range -7 to 7. It defines the number of source lines between the current source line and the previous line generating executable instructions. The delta value of the first line number entry is the displacement from the `lnLow` field in the procedure descriptor table.
- Count is a 4-bit field with a value in the range 0 to 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 when the delta value is zero.

An extended format of the line number entry is used when the delta value is outside the range -7 to 7. The layout of an extended line number entry on disk is as follows:



Note

Between two source lines that produce executable code, the compiler allows a maximum of 32,767 comment lines, blank lines, continuation lines, and other lines not producing executable instructions.

The following source listing can be used to show how the compiler assigns line numbers:

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

```
19  } /* end main */
```

The compiler generates line numbers only for the lines 3, 6, 8, 18, and 20; the other lines are either blank or contain comments.

The following table shows the `LINER` entries for each source line:

Source Line	LINER Contents	Meaning
3	`03`	Delta 0, count 3
6	`35`	Delta 3, count 5
8	`28`	Delta 2, count 8
18[a]	`87 00 0a`	Delta 10, count 7
20	`23`	Delta 2, count 3

Table Note:

a. Extended format (count is greater than 7 lines).

The compiler generates the following instructions for the example program:

```
[main.c:   3] 0x0:    27bb0001        ldah    gp, 1(t12)
[main.c:   3] 0x4:    23bd80d0        lda     gp, -32560(gp)
[main.c:   3] 0x8:    23deffe0        lda     sp, -32(sp)
[main.c:   3] 0xc:    b75e0008        stq     ra, 8(sp)
[main.c:   6] 0x10:   a61d8010        ldq     a0, -32752(gp)
[main.c:   6] 0x14:   22108000        lda     a0, -32768(a0)
[main.c:   6] 0x18:   a77d8018        ldq     t12, -32744(gp)
[main.c:   6] 0x1c:   6b5b4000        jsr     ra, (t12), printf
[main.c:   6] 0x20:   27ba0001        ldah    gp, 1(ra)
[main.c:   6] 0x24:   23bd80b0        lda     gp, -32592(gp)
[main.c:   8] 0x28:   a61d8020        ldq     a0, -32736(gp)
[main.c:   8] 0x2c:   a77d8028        ldq     t12, -32728(gp)
[main.c:   8] 0x30:   6b5b4000        jsr     ra, (t12), fgetc
[main.c:   8] 0x34:   27ba0001        ldah    gp, 1(ra)
[main.c:   8] 0x38:   23bd809c        lda     gp, -32612(gp)
[main.c:   8] 0x3c:   b41e0018        stq     v0, 24(sp)
[main.c:   8] 0x40:   44000401        bis     v0, v0, t0
[main.c:   8] 0x44:   48203f41        extqh   t0, 0x1, t0
[main.c:   8] 0x48:   48271781        sra     t0, 0x38, t0
[main.c:   8] 0x4c:   40203402        addq    t0, 0x1, t1
[main.c:   8] 0x50:   f440000a        bne     t1, 0x7c
[main.c:  18] 0x54:   a61d8010        ldq     a0, -32752(gp)
[main.c:  18] 0x58:   22108020        lda     a0, -32736(a0)
[main.c:  18] 0x5c:   44000411        bis     v0, v0, a1
[main.c:  18] 0x60:   4a203f51        extqh   a1, 0x1, a1
[main.c:  18] 0x64:   4a271791        sra     a1, 0x38, a1
[main.c:  18] 0x68:   a77d8018        ldq     t12, -32744(gp)
[main.c:  18] 0x6c:   6b5b4000        jsr     ra, (t12), printf
[main.c:  18] 0x70:   27ba0001        ldah    gp, 1(ra)
[main.c:  18] 0x74:   23bd8060        lda     gp, -32672(gp)
[main.c:  18] 0x78:   c3ffffeb        br      zero, 0x28
```

```
[main.c:  20] 0x7c:   47ff0400        bis     zero, zero, v0
[main.c:  20] 0x80:   a75e0008        ldq     ra, 8(sp)
[main.c:  20] 0x84:   23de0020        lda     sp, 32(sp)
[main.c:  20] 0x88:   6bfa8001        ret     zero, (ra), 1
```

8.2.3 Procedure Descriptor Table

Table 8-3 shows the format of an entry in the procedure descriptor table; the `sym.h` header file contains its declaration.

Table 8-3: Format of a Procedure Descriptor Table Entry

Declaration	Name	Description
unsigned long	adr	Memory address of start of procedure
long	cbLineOffset	Byte offset for this procedure from the base of the file descriptor entry
int	isym	Start of local symbols
int	iline[a]	Procedure's line numbers
int	regmask	Saved register mask
int	regoffset[b]	Saved register offset
int	iopt	Procedure's optimization symbol entries
int	fregmask	Save floating-point register mask
int	fregoffset	Save floating-point register offset
int	frameoffset	Frame size
int	lnLow	Lowest line in the procedure
int	lnHigh	Highest line in the procedure
unsigned	gp_prologue : 8[c]	Byte size of GP prologue
unsigned	gp_used : 1	True if the procedures uses gp
unsigned	reg_frame : 1	True if register frame procedure
unsigned	reserved : 14	
unsigned	localoff	Offset of local variables from vfp
short	framereg	Frame pointer register
short	pcreg	Index or reg of return program counter

Table Notes:

a. If the value of `iline` is NULL and the `cycm` field in the file descriptor table is zero, the `iline` field is indexed to the actual table.

b. If the value of `reg_frame` is 1, the `regoffset` field contains the register number of the register in which the return address is stored.

c. If the value of `gp_prologue` is zero and `gp_used` is 1, a gp prologue is present but has been scheduled into the procedure prologue.

8.2.4 Local Symbol Table

Table 8-4 shows the format of an entry in the local symbol table; the `sym.h` header file contains its declaration.

Table 8-4: Format of a Local Symbol Table Entry

Declaration	Name	Description
`long`	`value`[a]	Value of symbol.
`int`	`iss`[b]	Index into local strings of symbol name.
`unsigned`	`st : 6`[c]	Symbol type.
`unsigned`	`sc : 5`[d]	Storage class.
`unsigned`	`reserved : 1`	
`unsigned`	`index : 20`[e]	Index into local or auxiliary symbols.

Table Notes:

a. An integer representing an address, size, offset from a frame pointer. The value is determined by the symbol type, as illustrated in Table 8-5.

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

c. The symbol type (`st`) defines the symbol. The valid `st` Constants are given in Table 8-6. These constants are defined in `symconst.h`.

d. The storage class (`sc`), where applicable, explains how to access the symbol type in memory. The valid `sc` constants are given in Table 8-7. These constants are defined in `symconst.h`.

e. An offset into either the local symbol table or auxiliary symbol tables, depending of the storage type (`st`) as shown in Table 8-5. The compiler uses `isymBase` in the file descriptor entry as the base for an entry in the local symbol table and `iauxBase` for an entry in the auxiliary symbol table.

Table 8-5: 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[a]	iaux	Address
stStatic	scD/B[a]	iaux	Address
stParam	scAbs	iaux	Frame offset[b]
	scRegister	iaux	Register number
	scVar	iaux	Frame offset[b]
	scVarRegister	iaux	Register number
stLocal	scAbs	iaux	Frame offset[b]
	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[c]
union	scInfo	iaux	Bit offset[c]
stBlock			
enumeration	scInfo	isymMac[d]	Max enumeration
structure	scInfo	isymMac	Size
text block	scText	isymMac	Relative address[e]
common block	scCommon	isymMac	Size
variant	scVariant	isymMac	isymTag[f]
variant arm	scInfo	isymMac	iauxRanges[g]
union	scInfo	isymMac	Size
stEnd			
enumeration	scInfo	isymStart[h]	0
file	scText	isymStart	Relative address[e]
procedure	scText	isymStart	Relative address[e]
structure	scInfo	isymStart	0
text block	scText	isymStart	Relative address[e]
union	scInfo	isymStart	0
common block	scCommon	isymStart	0
variant	scVariant	isymStart	0
variant arm	scInfo	isymStart	0
stTypedef	scInfo	iaux	0

Table Notes:

a. scD/B is the storage class determined by the assembler, either data, sdata, bss, or sbss.

b. *frame offset* is the offset from the virtual frame pointer.

c. *bit offset* is computed from the beginning of the procedure.

d. `isymMac` is the `isym` of the corresponding `stEnd` symbol plus 1.

e. *relative address* is the relative displacement from the beginning of the procedure.

f. `isymTag` is the `isym` to the symbol that is the tag for the variant.

g. `iauxRanges` is the `iaux` to the ranges for the variant arm.

h. `isymStart` is the `isym` of the corresponding begin block (for example, `stBlock`, `stFile`, or `stProc`).

The linker ignores all symbols except the types that it will relocate: `stLabel`, `stStatic`, `stProc`, and `stStaticProc`. Other symbols are used only by the debugger and need to be entered in the table only when one of the debugging options (`-g` compilation options) is in effect.

8.2.4.1 Symbol Type (st) Constants

Table 8-6 gives the allowable constants that can be specified in the `st` field of entries in the local symbol table; the `symconst.h` header file contains the declaration for the constants.

Table 8-6: Symbol Type (st) Constants

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	Constant
`stStaParam`	16	Fortran static parameters
`stBase`	17	C++ base class
`stVirtBase`	18	C++ virtual base class
`stTag`	19	C++ tag
`stInter`	20	C++ interlude

Table 8-6: (continued)

8.2.4.2 Storage Class (sc) Constants

Table 8-7 gives the allowable constants that can be specified in the `sc` field of entries in the local symbol table; the `symconst.h` header file contains the declaration for the constants.

Table 8-7: Storage Class Constants

Constant	Value	Description
`scNil`	0	Dummy entry
`scText`	1	Text symbol
`scData`	2	Initialized data symbol
`scBss`	3	Uninitialized 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
	7	Reserved
`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	Small data (load time only)
`scSBss`	14	Small common (load time only)
`scRData`	15	Read only data (load time only)
`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
`scSUndefined`	21	Small undefined
`scInit`	22	`init` section symbol
	23	Reserved
`scXData`	24	Exception handling data
`scPData`	25	Exception procedure section
`scFini`	26	`fini` section symbol
`scMax`	32	

8.2.5 Optimization Symbol Table

Reserved for future use.

8.2.6 Auxiliary Symbol Table

Table 8-8 shows the format of an entry, which is a union, in the auxiliary symbol table; the `sym.h` file contains its declaration.

Table 8-8: Auxiliary Symbol Table Entries

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

Table Notes:

a. Type Information Record. Table 8-9 shows the format of a `ti` entry; the `sym.h` file contains its declaration.

b. Relative File Index. The compiler 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 symbol table or auxiliary symbol table.

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

d. Range Count. Used in describing case variants. Gives the number of elements that are separated by commas in a case variant.

Table 8-9: Format of an Type Information Record Entry

Declaration	Name	Description
`unsigned`	`fBitfield : 1`	Set if bit width is specified.
`unsigned`	`continued : 1`	Next auxiliary entry has `tq` information.
`unsigned`	`bt : 6`	Basic type.
`unsigned`	`tq4 : 4`	Type qualifier (`tq0` – `tq5`).
`unsigned`	`tq5 : 4`	
`unsigned`	`tq0 : 4`	

Table 8-9: (continued)

Declaration	Name	Description
`unsigned`	`tq1 : 4`	
`unsigned`	`tq2 : 4`	
`unsigned`	`tq3 : 4`	

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

- `fbitfield` is set if the basic type (`bt`) is of nonstandard width.
- `bt` (for basic type) specifies the type of the symbol (for example, integer, real, complex, or structure). The valid entries for this field are shown in Table 8-10; 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 Table 8-11; the `sym.h` file contains its declaration.

Table 8-10: Basic Type (bt) Constants

Constant	Value	Default Size[a]	Description
`btNil`	0	0	Undefined, void
`btAdr32`	1	32	Address (32 bits)
`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
`btLong32`	8	32	Long (32 bits)
`btULong32`	9	32	Unsigned long (32 bits)
`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 by means of a typedef; `rndx` points at a `stTypedef` symbol
`btRange`	16	32	Subrange of integer
`btSet`	17	32	Pascal sets

Table 8-10: (continued)

Constant	Value	Default Size[a]	Description
btComplex	18	64	Fortran complex
btDComplex	19	128	Fortran double complex
btIndirect	20		Indirect definition; rndx points to an entry in the auxiliary symbol table that contains a TIR (type information record)
btVoid	26	n/a	Void
btPtrMem	27	64	C++ Pointer to member
btVptr	28	n/a	C++ Virtual function table
btClass	29	n/a	C++ Class (record)
btLong64	30	64	Address (64 bits)
btLong	30	64	Synonym for btLong64
btULong64	31	64	Unsigned long (64 bits)
btULong	31	64	Synonym for btUlong64
btLongLong	32	64	Long long (64 bits)
btULongLong	33	64	Unsigned long long (64 bits)
btAdr64	34	64	Address (64 bits)
btAdr	34	64	Synonym for btAdr64
btMax	64		

Table Notes:

a. Size in bits.

Table 8-11: 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
tqConst	6	Constant
tqRef	7	Reference
tqMax	8	Number of type qualifiers

8.2.7 File Descriptor Table

Table 8-12 shows the format of an entry in the file descriptor table; the `sym.h` file contains its declaration.

Table 8-12: Format of File Descriptor Entry

Declaration	Name	Description
`unsigned long`	`adr`	Memory address of start of file
`long`	`cbLineOffset`	Byte offset from header or file lines
`long`	`cbLine`	Size of lines for the file
`long`	`cbSs`	Number of bytes in local strings
`int`	`rss`	Source file name
`int`	`issBase`	Start of local strings
`int`	`isymBase`	Start of local symbol entries
`int`	`csym`	Count of local symbol entries
`int`	`ilineBase`	Start of line number entries
`int`	`cline`	Count of line number entries
`int`	`ioptBase`	Start of optimization symbol entries
`int`	`copt`	Count of optimization symbol entries
`int`	`ipdFirst`	Start of procedure descriptor table
`int`	`cpd`	Count of procedures descriptors
`int`	`iauxBase`	Start of auxiliary symbol entries
`int`	`caux`	Count of auxiliary symbol entries
`int`	`rfdBase`	Index into relative file descriptors
`int`	`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`	Not used
`unsigned`	`glevel : 2`	Level this file was compiled with
`unsigned`	`reserved : 22`	Reserved for future use

8.2.8 External Symbol Table

The external symbol table has the same format as the local symbol table, except an offset (`ifd`) field has been added to the file descriptor table. This field is used to locate information associated with the symbol in an auxiliary symbol table. Table 8-13 shows the format of an entry in the external symbol table; the `sym.h` file contains its declaration.

Table 8-13: External Symbol Table Entries

Declaration	Name	Description
`SYMR`	`asym`	Same as local symbol table
`unsigned short`	`weakext : 1`	Symbol is weak external
`unsigned short`	`reserved : 15`	Reserved for future use
`int`	`ifd`	Pointer to entry in file descriptor table

Program Loading and Dynamic Linking 9

Executable files and shared library 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 executable and shared object files.

This chapter contains the following major sections:

- Object File Considerations (Section 9.1) – provides background information on factors that influence linking and loading operations.
- Program Loading (Section 9.2) – describes the loading process.
- Dynamic Linking (Section 9.3) – describes dynamic linking and loading.

9.1 Object File Considerations

The following sections describe several general factors that are involved in the linking and loading process.

9.1.1 Structures

The following object file structures contain information that is used in linking and loading operations:

- File Header – The file header identifies a file as an object file and additionally indicates whether the object is a static executable, a shared executable, or a shared library.
- Optional Header – The optional header immediately follows the file header and identifies the size, location and virtual addresses of the object's segments.
- Section Headers – Section headers describe the individual sections that comprise the object's segments. Section headers are normally not used in program loading; however, the section headers are used to locate the dynamic section in shared executable files and shared libraries.

See Chapter 7 for further details on file headers, optional headers, and section headers.

9.1.2 Base Addresses

Executable files 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 during dynamic linking.

During program loading, 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.

9.1.3 Segment Access 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, the segments are assigned access permissions, which are determined by the type of segment and type of program image. Table 9-1 shows the access permissions for the various segment and image types.

Table 9-1: Segment Access Permissions

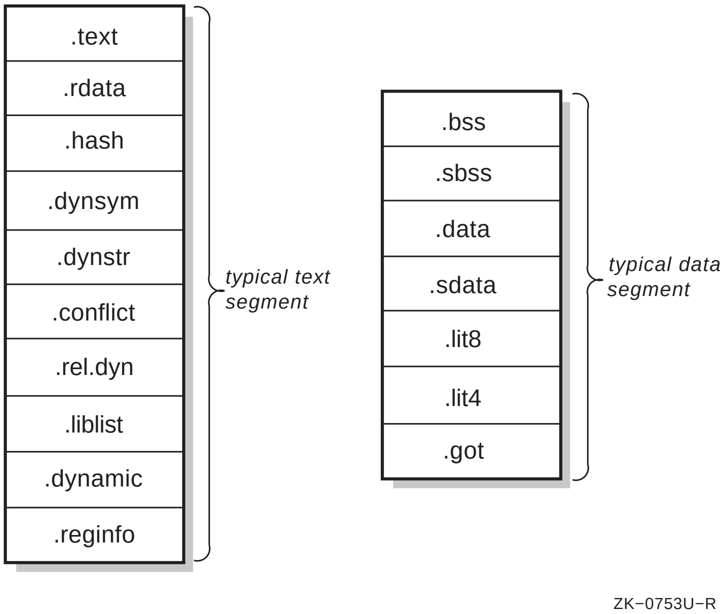
Image	Segment	Access Permissions
OMAGIC	text, data, bss	Read, Write, Execute
NMAGIC	text	Read, Execute
NMAGIC	data, bss	Read, Write, Execute
ZMAGIC	text	Read, Execute
ZMAGIC	data, bss	Read, Write, Execute

9.1.4 Segment Contents

An object file segment can 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. Figure 9-1 illustrates typical segment contents for an executable or shared object. The order of sections within a segment may vary.

Text segments contain instructions and read-only data, and data segments contain writable data. Text segments and data segments typically include the sections shown in Figure 9-1.

Figure 9-1: Text and Data Segments of Object Files



9.2 Program Loading

As the system creates or augments a process image, it logically copies a file's segment to a virtual memory segment. When the system physically reads the file depends on the programs execution behavior, system load, and other factors. A process does not require a physical page unless it references the logical page during execution. Processes commonly leave many pages unreferenced. Therefore, delaying physical reads frequently obviates them, thus 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 the text and data segments must be aligned on 64KB (0x10000) or larger power of 2 boundaries. File offsets must be aligned on 8KB (0x2000) or larger power of 2 boundaries.

Because the page size can be larger than the alignment restrictions of a segment's file offset, up to four file pages can hold text or data that is not logically part of the segment (depending on page size and file system block size). The contents of the various file pages are as follows:

- The first text page contains the COFF file header, section headers, 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 may contain file information not relevant to the running process.

Logically, the system enforces the memory permissions as if each segment were complete and separate; segment's addresses are adjusted to ensure that each logical page in the address space has a single set of permissions.

The end of the data segment requires special handling for uninitialized data, which must be set to zero. 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 contents of the executable file.

9.3 Dynamic Linking

An executable file is loaded at fixed addresses; the system creates its segments using the virtual addresses from the optional header. The system transfers control directly to the entry point of the executable file.

An executable file that uses dynamic linking requires one or more shared objects to be loaded in addition to the executable file. Instead of loading the executable file, the system loads the dynamic loader, which in turn loads the executable file and its shared objects.

9.3.1 Dynamic Loader

When building an executable file that uses dynamic linking, the linker adds the flag `F_MIPS_CALL_SHARED` to the `f_flags` field of the file header. This flag tells the system to invoke the dynamic loader to load the executable file. Typically, the dynamic loader requested is `/sbin/loader`, the default loader. The `exec` function and the dynamic loader cooperate to create the process image. Creating the process image involves the following operations:

- 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
- Transferring control to the program, making it appear that the program received control directly from `exec`

To assist the dynamic loader, the linker also constructs the following data items for shared objects and executable files:

– A number of dynamic sections (.dynamic, .rel.dyn, and .dynstr) hold various data items, including a structure that holds the addresses of other dynamic linking information.

– The .hash section contains a symbol hash table.

– The .got section contains the global offset table.

These data items are located in loadable segments and are available during execution.

Shared objects may be located at virtual addresses that are different from the addresses in the optional header. The dynamic loader 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. (See Section 9.3.5 for information about lazy binding.)

The following sections describe the various dynamic linking sections. The C language definitions are in the header files elf_abi.h and elf_mips.h.

9.3.2 Dynamic Section (.dynamic)

The dynamic section acts as a table of contents for information, relating to dynamic linking, within the object. Dynamic sections are present only in shared objects and shared executables (objects that use shared libraries).

The dynamic section is located by its section header. This section header is identified by its name (.dynamic) or its section type (STYP_DYNAMIC) in the flags field (s_flags).

The dynamic section 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;
```

The structure and union members in the preceding definition provide the following information:

d_tag
: Indicates how the d_un field is to be interpreted.

d_val
: Represents integer values.

d_ptr
: Represents program virtual addresses. A file’s virtual addresses may not

match the memory virtual addresses during execution. The dynamic loader 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 section.

The `d_tag` requirements for executable and shared object files are summarized in Table 9-2. ‘‘Mandatory’’ indicates that 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.

Table 9-2: Dynamic Array Tags (d_tag)

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
DT_STRTAB	5	d_ptr	mandatory	mandatory
DT_SYMTAB	6	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	ignored	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	unspecified

The uses of the various dynamic array tags are as follows:

DT_NULL
: Marks the end of the array.

DT_NEEDED

Contains the string table offset of a null terminated string that is the name of the 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 these entries is not significant.

DT_PLTRELSZ
: 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
: Contains an address associated with either the procedure linkage table and/or the global offset table.

DT_HASH
: Contains the address of the symbol hash table.

DT_STRTAB
: Contains the address of the string table.

DT_SYMTAB
: Contains the address of the symbol table with `Elf32_Sym` entries.

DT_RELA
: 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 linker 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 entry is present, the dynamic structure must also contain `DT_RELASZ` and `DT_RELAENT` entries. When relocation is mandatory for a file, either `DT_RELA` or `DT_REL` may be present.

DT_RELASZ
: Contains the size in bytes of the `DT_RELA` relocation table.

DT_RELAENT
: Contains the size in bytes of a `DT_RELA` relocation table entry.

DT_STRSZ
: Contains the size in bytes of the string table.

DT_SYMENT
: Contains the size in bytes of a symbol table entry.

DT_INIT
: Contains the address of the initialization function.

DT_FINI
: Contains the address of the termination function.

DT_SONAME
: 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
: Contains the string table offset of a null-terminated library search path string. The offset is an index into the table indicated in the `DT_STRTAB` entry.

DT_SYMBOLIC
: If this entry 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
: Contains the address of the relocation table. An object file can have multiple relocation sections. When the linker 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 loader sees a single table. When the dynamic loader 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 entry is present, the dynamic structure must contain the `DT_RELSZ` entry.

DT_RELSZ
: Contains the size in bytes of the relocation table pointed to by the `DT_REL` entry.

DT_RELENT
: Contains the size in bytes of a `DT_REL` entry.

DT_PLTREL
: 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
: Used for debugging. The contents of this entry are not specified.

DT_TEXTREL
: If this entry is not present, no relocation entry should cause a modification to a nonwritable segment. If this entry is present, one or more relocations might request modifications to a nonwritable segment.

DT_JMPREL

If this entry 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. (See Section 9.3.5 for information about lazy binding.)

DT_LOPROC through DT_HIPROC

Reserved for processor-specific semantics.

Table 9-3: Processor-Specific Dynamic Array 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

The uses of the various processor-specific dynamic array tags are as follows:

DT_MIPS_RLD_VERSION

Holds an index into the object file's string table, which holds the version of the run-time linker interface. The version is currently 1.

DT_MIPS_TIME_STAMP

Contains a 32-bit time stamp.

DT_MIPS_CHECKSUM

Contains a value that is the sum of all external strings and COMMON sizes.

DT_MIPS_IVERSION

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
Contains a set of 1-bit flags. 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

DT_MIPS_BASE_ADDRESS
Contains the base address.

DT_MIPS_CONFLICT
Contains the address of the `.conflict` section.

DT_MIPS_LIBLIST
Contains the address of the `.liblist` section.

DT_MIPS_LOCAL_GOTNO
Contains the number of local GOT entries.

DT_MIPS_CONFLICTNO
Contains the number of entries in the `.conflict` section and is mandatory if there is a `.conflict` section.

DT_MIPS_RLD_MAP
Contains the address of the quadword that contains a pointer to the dynamic linker’s object list.

DT_MIPS_SYMTABNO
Indicates the number of entries in the `.dynsym` section.

DT_MIPS_LIBLISTNO
Contains the number of entries in the `.liblist` section.

DT_MIPS_UNREFEXTNO
Holds an index into the dynamic symbol table. The index is the entry of the first external symbol that is not referenced within the object.

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

DT_MIPS_HIPAGENO
Holds the number of page table entries in the global offset table. A page table entry here refers to 64KB of data space. This entry is used by the profiling tools and is optional.

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

9.3.3 Shared Object Dependencies

When the linker 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 that require the dynamic linker to include the appropriate shared object files in the process image. To accomplish this, executable files and shared object files must describe their dependencies.

The dependencies, indicated by the `DT_NEEDED` entries of the dynamic structure, indicate which 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 includes only one instance of the object in the process image.

Names in the dependency list are copies of either the `DT_SONAME` strings or the pathnames of the shared objects used to build the object file. If the linker builds an executable file from a shared object with a `DT_SONAME` entry of `liba` and another shared object with the pathname `/usr/lib/libz`, the executable file contains `liba` and `/usr/lib/libz` in its dependency list.

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

1. The dynamic array tag `DT_RPATH` may give a string that holds a list of directories separated by colons, such as `/usr/newlib:/usr/local/lib`. The dynamic linker searches these directories in order and, if a library is not located, then searches the current directory.
2. 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.`
3. 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.

The following environment variables are defined:

_RLD_PATH	Path to dynamic linker (rld)
_RLD_ARGS	Argument 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.

9.3.4 Global Offset Table (.got)

Position-independent code cannot, in general, contain absolute virtual addresses. Global offset tables (GOTs) 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 – local and external:

- 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 quadword of the table, the second symbol corresponds to the second quadword, and so on. Each quadword 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 Section 9.3.11 for information about quickstart processing.)

After the system creates memory segments for a loadable object file, the dynamic loader may process the relocation entries. The only relocation entries remaining are type R_MIPS_REL64 or R_MIPS_REL32, referring to local entries in the GOT and data items containing addresses. The dynamic loader 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 linker builds an object file, the dynamic loader 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 loader processes all necessary relocations before giving control to the process image, thus ensuring the absolute addresses are available during execution.

The zero (first) 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 using *lazy binding* to resolve text symbols (see Section 9.3.5 for information about resolving text symbols using lazy binding).

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.

The DT_MIPS_LOCAL_GOTNO and DT_PLTGOT entries of the dynamic section describe the attributes of the global offset table.

9.3.5 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 the 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 deferred through a technique known as lazy binding.

Using the lazy binding technique, the linker builds a stub for each called function and allocates GOT entries that initially point to the stubs. 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:
    ldq  t12, .got_index(gp)
    lda  $at, .dynsym_index(zero)
    nop
    jmp  t12, (t12)
```

The stub code loads register `t12` with an entry from the GOT. The entry loaded into register `t12` is the address of the procedure in the dynamic linker that handles lazy binding. The stub code also loads register `$at` with the index into the `.dynsym` section of the referenced external symbol. The code then transfers control to the dynamic linker and loads register `t12` with the address following the stub. 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 in other than a `jsr` instruction. In the exception case, the program uses the address of the stub instead of the actual address of the function. Determining which case is in effect is based on the following processing:

- The linker 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 nonzero, only `jsr` references were made to the function and nothing needs to be done to the GOT entry.
    - If the field is zero, some other kind of reference was made 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. Two situations, however, make lazy binding undesirable for some applications:

- 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.

– 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.

9.3.6 Dynamic Symbol Section (.dynsym)

The dynamic symbol section provides information on all external symbols, either imported or exported from an object.

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

Undefined symbols of type `STT_FUNC` that have been referenced only by `jsr` instructions may contain nonzero values in 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 symbol 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 file cannot be preempted. The symbol table in the executable is always searched first to resolve any symbol references.

The dynamic symbol section contains an array of entries of the following type:

```
typedef struct {
    Elf32_Word      st_name;
    Elf32_Addr      st_value;
    Elf32_Word      st_size;
    unsigned char   st_info;
    unsigned char   st_other;
    Elf32_Half      st_shndx;
} Elf32_Sym;
```

The structure members in the preceding definition provide the following information:

`st_name`
Contains the offset of the symbol’s name in the dynamic string section.

`st_value`
Contains the value of the symbol for those symbols defined within the object; otherwise, contains the value zero.

st_size
: Identifies the size of symbols with *common* storage allocation; otherwise, contains the value zero.

st_info
: Identifies both the binding and the type of the symbol. The macros ELF32_ST_BIND and ELF32_ST_TYPE are used to access the individual values.

A symbol's binding determines the linkage visibility and behavior. The binding is encoded in the st_info field and can have one of the following values:

Value	Description
STB_LOCAL	Indicates that the symbol is local to the object.
STB_GLOBAL	Indicates that the symbol is visible to other objects.
STB_WEAK	Indicates that the symbol is a weak global symbol.

A symbol's type identifies its use. The type is encoded in the st_info field and can have one of the following values:

Value	Description
STT_NOTYPE	Indicates that the symbol has no type or its type is unknown.
STT_OBJECT	Indicates that the symbol is a data object.
STT_FUNC	Indicates that the symbol is a function.
STT_SECTION	Indicates that the symbol is associated with a program section.
STT_FILE	Indicates that the symbol as the name of a source file.

st_other
: Currently holds a value of zero and has no defined meaning.

st_shndx
: Identifies the section to which this symbol is related.

All symbols are defined relative to some program section. The st_shndx field identifies the section and can have one of the following values:

Value	Description
SHN_UNDEF	Indicates that the symbol is undefined.
SHN_ABS	Indicates that the symbol has an absolute value.
SHN_COMMON	Indicates that the symbol has common storage (unallocated).
SHN_ACOMMON	Indicates that the symbol has common storage (allocated).

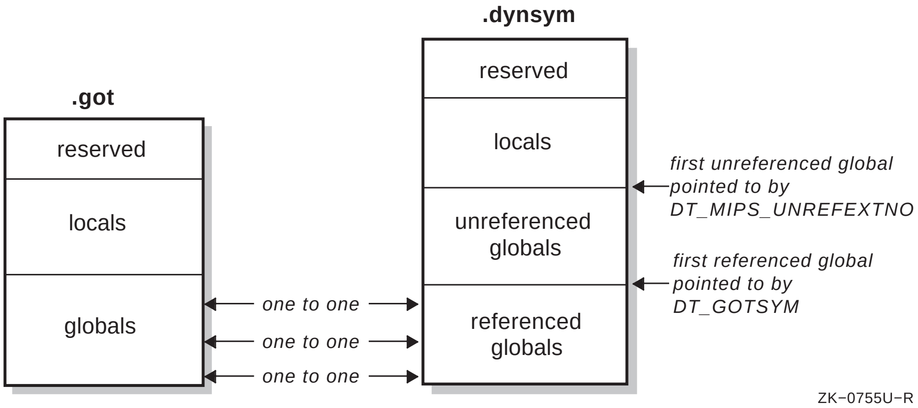
Value	Description
SHN_TEXT	Indicates that the symbol is in a text segment.
SHN_DATA	Indicates that the symbol is in a data segment.

The entries of the dynamic symbol section are ordered as follows:

- A single null entry.
- Symbols local to the object.
- Unreferenced global symbols, that is, symbols that are defined within the object but not referenced.
- Referenced global symbols. These symbols are ordered by the value of their `st_value` field (lowest value first). These symbols also correspond one-to-one with the GOT entries for global symbols.

Figure 9-2 shows the layout of the `.dynsym` section and its relationship to the `.got` section.

Figure 9-2: Relationship Between .dynsym and .got



The `DT_SYMENT` and `DT_SYMTAB` entries of the dynamic section describe the attributes of the dynamic symbol table.

9.3.7 Dynamic Relocation Section (.rel.dyn)

The dynamic relocation section describes all locations within the object that must be adjusted if the object is loaded at an address other than its linked base address.

Only one dynamic relocation section is used to resolve addresses in data items 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` and `R_MIPS_REL64` relocation entries are supported in the dynamic relocation section.

The dynamic relocation section is an array of entries of the following type:

```
typedef struct {
    Elf32_Addr      r_offset;
    Elf32_Word      r_info;
} Elf32_Rel;
```

The structure members in the preceding definition provide the following information:

`r_offset`
: Identifies the location within the object to be adjusted.

`r_info`
: Identifies the relocation type and the index of the symbol that is referenced. The macros `ELF32_R_SYM` and `ELF32_R_TYPE` access the individual attributes. The relocation type must be either `R_MIPS_REL32` or `R_MIPS_REL64`.

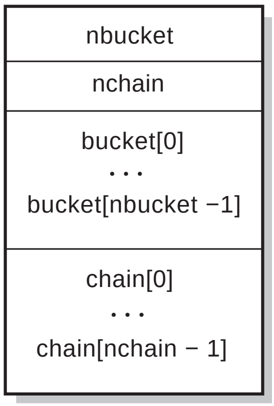
The entries of the dynamic relocation section are ordered by symbol index value.

The `DT_REL` and `DT_RELSZ` entries of the dynamic section describe the attributes of the dynamic relocation section.

9.3.8 Hash Table Section (.hash)

A hash table of `Elf32_Word` entries supplies symbol table access. Figure 9-3 shows the contents of a hash table.

Figure 9-3: Hash Table Section



ZK−0756U−R

The entries in the hash table contain the following information:

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

  The hashing function accepts a symbol name and returns a value that can 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`.

The `DT_HASH` entry of the dynamic section contains the address of the hash table section.

9.3.9 Dynamic String Section (.dynstr)

The dynamic string section is the repository for all strings referenced by the dynamic linking sections. Strings are referenced by a byte offset within the dynamic string section. The end of the string is denoted by a byte containing the value zero.

The DT_STRTAB and DT_STRSZ entries of the dynamic section describe the attributes of the dynamic string section.

9.3.10 Initialization and Termination Functions

After the dynamic loader 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 objects initialization occurs before the executable file gains control.

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

Shared objects designate initialization and termination functions through the 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 atexit termination processing normally is done, it is not guaranteed to have executed when the process terminates. In particular, the process does not execute the termination processing if it calls _exit or if the process terminates because it received a signal that it neither caught nor ignored.

9.3.11 Quickstart

The assembler supports several sections that 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.

9.3.11.1 Shared Object List (.liblist)

A shared object list section is an array of Elf32_Lib structures that 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;
```

The structure members in the preceding definition provide the following information:

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

`l_time_stamp`
: Contains a 32-bit time stamp. The value can be combined with the `l_checksum` value and the `l_version` string to form a unique identifier for this shared object.

`l_checksum`
: Contains the sum of all externally visible symbols' string names and common sizes.

`l_version`
: Specifies the interface version. Its value is a string table index. The interface version is a 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`
: Specifies a set of 1-bit flags.

The `l_flags` field can have one or both of the following flags set:

`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.

The DT_MIPS_LIBLIST and DT_MIPS_LIBLISTNO entries of the dynamic section describe the attributes of the shared object list section.

9.3.11.2 Conflict Section (.conflict)

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_Word Elf32_Conflict;
```

The DT_MIPS_CONFLICT and DT_MIPS_CONFLICTNO entries of the dynamic section describe the attributes of the conflict section.

9.3.11.3 Ordering of Sections

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; it must correspond to the .dynsym section.
- The .rel.dyn section must have all local entries first, followed by the external entries. Within these subsections, the entries must be ordered by symbol index. This groups each symbol's relocations together.

Instruction Summaries A

The tables in this appendix summarize the assembly-language instruction set:

– Table A-1 summarizes the main instruction set.

– Table A-2 summarizes the floating-point instruction set.

– Table A-3 summarizes the rounding and trapping modes supported by some floating-point instructions.

Most of the assembly-language instructions translate into single instructions in machine code.

The tables in this appendix show the format of each instruction in the main instruction set and the floating-point instruction set. The tables list the instruction names and the forms of operands that can be used with each instruction. The specifiers used in the tables to identify operands have the following meanings:

Operand Specifier	Description
address	A symbolic expression whose effective value is used as an address.
b_reg	Base register. A register containing a base address to which is added an offset (or displacement) value to produce an effective address.
d_reg	Destination register. A register that receives a value as a result of an operation.
d_reg/s_reg	One register that is used as both a destination register and a source register.
label	A label that identifies a location in a program.
no_operands	No operands are specified.
offset	An immediate value that is added to the contents of a base register to calculate an effective address.
palcode	A value that determines the operation performed by a PAL instruction.
s_reg, s_reg1, s_reg2	Source registers. Registers whose contents are to be used in an operation.
val_expr	An expression whose value is used as an absolute value.

Operand Specifier	Description
val_immed	An immediate value that is to be used in an operation.
jhint	An address operand that provides a hint of where a `jmp` or `jsr` instruction will transfer control.
rhint	An immediate operand that provides software with a hint about how a `ret` or `jsr_coroutine` instruction is used.

The tables in this appendix are segmented into groups of instructions that have the same operand options; the operands specified within a particular segment of the table apply to all of the instructions contained in that segment.

Table A-1: Main Instruction Set Summary

Instruction	Mnemonic	Operands
Load Address	`lda`	*d_reg,address*
Load Byte	`ldb`	
Load Byte Unsigned	`ldbu`	
Load Word	`ldw`	
Load Word Unsigned	`ldwu`	
Load Sign Extended Longword	`ldl`	
Load Sign Extended Longword Locked	`ldl_l`	
Load Quadword	`ldq`	
Load Quadword Locked	`ldq_l`	
Load Quadword Unaligned	`ldq_u`	
Load Unaligned Word	`uldw`	
Load Unaligned Word Unsigned	`uldwu`	
Load Unaligned Longword	`uldl`	
Load Unaligned Quadword	`uldq`	
Store Byte	`stb`	*s_reg,address*
Store Word	`stw`	
Store Longword	`stl`	
Store Longword Conditional	`stl_c`	
Store Quadword	`stq`	
Store Quadword Conditional	`stq_c`	
Store Quadword Unaligned	`stq_u`	
Store Unaligned Word	`ustw`	
Store Unaligned Longword	`ustl`	
Store Unaligned Quadword	`ustq`	
Load Address High	`ldah`	*d_reg,offset(b_reg)*
Load Global Pointer	`ldgp`	

Table A-1: (continued)

Instruction	Mnemonic	Operands
Load Immediate Longword	`ldil`	*d_reg,val_expr*
Load Immediate Quadword	`ldiq`	
Branch if Equal to Zero	`beq`	*s_reg,label*
Branch if Not Equal to Zero	`bne`	
Branch if Less Than Zero	`blt`	
Branch if Less Than or Equal to Zero	`ble`	
Branch if Greater Than Zero	`bgt`	
Branch if Greater Than or Equal to Zero	`bge`	
Branch if Low Bit is Clear	`blbc`	
Branch if Low Bit is Set	`blbs`	
Branch	`br`	*d_reg,label*
Branch to Subroutine	`bsr`	*label*
Jump	`jmp`	*d_reg,(s_reg),jhint*
Jump to Subroutine	`jsr`	*d_reg,(s_reg)*
		(s_reg),jhint
		(s_reg)
		d_reg,address
		address
Return from Subroutine	`ret`	*d_reg,(s_reg),rhint*
Jump to Subroutine Return	`jsr_coroutine`	*d_reg,(s_reg)*
		d_reg,rhint
		d_reg
		(s_reg),rhint
		(s_reg)
		rhint
		no_operands
Clear	`clr`	*d_reg*
Absolute Value Longword	`absl`	*s_reg,d_reg*
Absolute Value Quadword	`absq`	*d_reg/s_reg*
Move	`mov`	*val_immed,d_reg*
Negate Longword (without overflow)	`negl`	
Negate Longword (with overflow)	`neglv`	
Negate Quadword (without overflow)	`negq`	
Negate Quadword (with overflow)	`negqv`	
Logical Complement (NOT)	`not`	

Table A-1: (continued)

Instruction	Mnemonic	Operands
Sign-Extension Longword	`sextl`	(see previous page)
Add Longword (without overflow)	`addl`	*s_reg1,s_reg2,d_reg*
Add Longword (with overflow)	`addlv`	*d_reg/s_reg1,s_reg2*
Add Quadword (without overflow)	`addq`	*s_reg1,val_immed,d_reg*
Add Quadword (with overflow)	`addqv`	*d_reg/s_reg1,val_immed*
Scaled Longword Add by 4	`s4addl`	
Scaled Quadword Add by 4	`s4addq`	
Scaled Longword Add by 8	`s8addl`	
Scaled Quadword Add by 8	`s8addq`	
Compare Signed Quadword Equal	`cmpeq`	
Compare Signed Quadword Less Than	`cmplt`	
Compare Signed Quadword Less Than or Equal	`cmple`	
Compare Unsigned Quadword Less Than	`cmpult`	
Compare Unsigned Quadword Less Than or Equal	`cmpule`	
Multiply Longword (without overflow)	`mull`	
Multiply Longword (with overflow)	`mullv`	
Multiply Quadword (without overflow)	`mulq`	
Multiply Quadword (with overflow)	`mulqv`	
Subtract Longword (without overflow)	`subl`	
Subtract Longword (with overflow)	`sublv`	
Subtract Quadword (without overflow)	`subq`	
Subtract Quadword (with overflow)	`subqv`	
Scaled Longword Subtract by 4	`s4subl`	
Scaled Quadword Subtract by 4	`s4subq`	
Scaled Longword Subtract by 8	`s8subl`	
Scaled Quadword Subtract by 8	`s8subq`	
Unsigned Quadword Multiply High	`umulh`	
Divide Longword	`divl`	
Divide Longword Unsigned	`divlu`	
Divide Quadword	`divq`	
Divide Quadword Unsigned	`divqu`	
Longword Remainder	`reml`	
Longword Remainder Unsigned	`remlu`	
Quadword Remainder	`remq`	
Quadword Remainder Unsigned	`remqu`	
Logical Product (AND)	`and`	
Logical Sum (OR)	`bis`	

Table A-1: (continued)

Instruction	Mnemonic	Operands
Logical Sum (OR)	`or`	(see previous page)
Logical Difference (XOR)	`xor`	
Logical Product with Complement (ANDNOT)	`bic`	
Logical Product with Complement (ANDNOT)	`andnot`	
Logical Sum with Complement (ORNOT)	`ornot`	
Logical Equivalence (XORNOT)	`eqv`	
Logical Equivalence (XORNOT)	`xornot`	
Move if Equal to Zero	`cmoveq`	
Move if Not Equal to Zero	`cmovne`	
Move if Less Than Zero	`cmovlt`	
Move if Less Than or Equal to Zero	`cmovle`	
Move if Greater Than Zero	`cmovgt`	
Move if Greater Than or Equal to Zero	`cmovge`	
Move if Low Bit Clear	`cmovlbc`	
Move if Low Bit Set	`cmovlbs`	
Shift Left Logical	`sll`	
Shift Right Logical	`srl`	
Shift Right Arithmetic	`sra`	
Compare Byte	`cmpbge`	
Extract Byte Low	`extbl`	
Extract Word Low	`extwl`	
Extract Longword Low	`extll`	
Extract Quadword Low	`extql`	
Extract Word High	`extwh`	
Extract Longword High	`extlh`	
Extract Quadword High	`extqh`	
Insert Byte Low	`insbl`	
Insert Word Low	`inswl`	
Insert Longword Low	`insll`	
Insert Quadword Low	`insql`	
Insert Word High	`inswh`	
Insert Longword High	`inslh`	
Insert Quadword High	`insqh`	
Mask Byte Low	`mskbl`	
Mask Word Low	`mskwl`	
Mask Longword Low	`mskll`	
Mask Quadword Low	`mskql`	
Mask Word High	`mskwh`	
Mask Longword High	`msklh`	
Mask Quadword High	`mskqh`	
Zero Bytes	`zap`	
Zero Bytes NOT	`zapnot`	

Table A-1: (continued)

Instruction	Mnemonic	Operands
Call Privileged Architecture Library	`call_pal`	*palcode*
Prefetch Data	`fetch`	*address*
Prefetch Data, Modify Intent	`fetch_m`	
Read Process Cycle Counter	`rpcc`	*d_reg*
No Operation	`nop`	*no_operands*
Universal No Operation	`unop`	
Trap Barrier	`trapb`	
Exception Barrier	`excb`	
Memory Barrier	`mb`	
Write Memory Barrier	`wmb`	

A number of the floating-point instructions in Table A-2 support qualifiers that control rounding and trapping modes. Table notes identify the qualifiers that can be used with a particular instruction. Qualifiers are appended as suffixes to the particular instructions that support them, for example, the instruction `cvtdg` with the `sc` qualifier would be coded `cvtdgsc`. The qualifier suffixes consist of one or more characters, with each character identifying a particular rounding or trapping mode. Table A-3 defines the rounding or trapping modes associated with each character.

Table A-2: Floating-Point Instruction Set Summary

Instruction	Mnemonic	Operands
Load F_Floating	`ldf`	*d_reg,address*
Load G_Floating (Load D_Floating)	`ldg`	
Load S_Floating (Load Longword)	`lds`	
Load T_Floating (Load Quadword)	`ldt`	
Store F_Floating	`stf`	*s_reg,address*
Store G_Floating (Store D_Floating)	`stg`	
Store S_Floating (Store Longword)	`sts`	
Store T_Floating (Store Quadword)	`stt`	
Load Immediate F_Floating	`ldif`	*d_reg,val_expr*
Load Immediate D_Floating	`ldid`	
Load Immediate G_Floating	`ldig`	
Load Immediate S_Floating	`ldis`	
Load Immediate T_Floating	`ldit`	

Table A-2: (continued)

Instruction	Mnemonic	Operands
Branch Equal to Zero	fbeq	*s_reg,label*
Branch Not Equal to Zero	fbne	*label*
Branch Less Than Zero	fblt	
Branch Less Than or Equal to Zero	fble	
Branch Greater Than Zero	fbgt	
Branch Greater Than or Equal to Zero	fbge	
Floating Clear	fclr	*d_reg*
Floating Move	fmov	*s_reg,d_reg*
Floating Negate	fneg	*d_reg/s_reg*
Floating Absolute Value	fabs	
Negate F_Floating[c]	negf	
Negate G_Floating[c]	negg	
Negate S_Floating[d]	negs	
Negate T_Floating[d]	negt	
Copy Sign	cpys	*s_reg1,s_reg2,d_reg*
Copy Sign Negate	cpysn	*d_reg/s_reg1,s_reg2*
Copy Sign and Exponent	cpyse	
Move if Equal to Zero	fcmoveq	
Move if Not Equal to Zero	fcmovne	
Move if Less Than Zero	fcmovlt	
Move if Less Than or Equal to Zero	fcmovle	
Move if Greater Than Zero	fcmovgt	
Move if Greater Than or Equal to Zero	fcmovge	
Add F_Floating[a]	addf	
Add G_Floating[a]	addg	
Add S_Floating[b]	adds	
Add T_Floating[b]	addt	
Compare G_Floating Equal[c]	cmpgeq	
Compare G_Floating Less Than[c]	cmpglt	
Compare G_Floating Less Than or Equal[c]	cmpgle	
Compare T_Floating Equal[d]	cmpteq	
Compare T_Floating Less Than[d]	cmptlt	
Compare T_Floating Less Than or Equal[d]	cmptle	
Compare T_Floating Unordered[d]	cmptun	
Divide F_Floating[a]	divf	
Divide G_Floating[a]	divg	
Divide S_Floating[b]	divs	
Divide T_Floating[b]	divt	
Multiply F_Floating[a]	mulf	
Multiply G_Floating[a]	mulg	
Multiply S_Floating[b]	muls	
Multiply T_Floating[b]	mult	

Table A-2: (continued)

Instruction	Mnemonic	Operands
Subtract F_Floating[a]	subf	(see previous page)
Subtract G_Floating[a]	subg	
Subtract S_Floating[b]	subs	
Subtract T_Floating[b]	subt	
Convert Quadword to Longword[e]	cvtql	*s_reg,d_reg*
Convert Longword to Quadword	cvtlq	*d_reg/s_reg*
Convert G_Floating to Quadword[f]	cvtgq	
Convert T_Floating to Quadword[g]	cvttq	
Convert Quadword to F_Floating[h]	cvtqf	
Convert Quadword to G_Floating[h]	cvtqg	
Convert Quadword to S_Floating[i]	cvtqs	
Convert Quadword to T_Floating[i]	cvtqt	
Convert D_Floating to G_Floating[a]	cvtdg	
Convert G_Floating to D_Floating[a]	cvtgd	
Convert G_Floating to F_Floating[a]	cvtgf	
Convert T_Floating to S_Floating[b]	cvtts	
Convert S_Floating to T_Floating[c]	cvtst	
Move From FP Control Register	mf_fpcr	*d_reg*
Move To FP Control Register	mt_fpcr	*s_reg*
Floating No Operation	fnop	*no_operands*

Table notes:

a. c, u, uc, s, sc, su, suc
b. c, m, d, u, uc, um, ud, su, suc, sum, sud, sui, suic, suim, suid
c. s
d. su
e. sv, v
f. c, v, vc, s, sc, sv, svc
g. c, v, vc, sv, svc, svi, svic, d, vd, svd, svid
h. c
i. c, m, d, sui, suic, suim, suid

Table A-3: Rounding and Trapping Modes

Suffix	Description
(no suffix)	Normal rounding
`c`	Chopped rounding
`d`	Dynamic rounding
`m`	Minus infinity rounding
`s`	Software completion
`u`	Underflow trap enabled
`v`	Integer overflow trap enabled
`i`	Inexact trap enabled

32-Bit Considerations B

The Alpha architecture is a quadword (32-bit) architecture, with limited backward compatibility for longword operations. The Alpha design philosophy for longword operations is to use the quadword instructions wherever possible and to include specialized longword instructions for high-frequency operations.

B.1 Canonical Form

Longword operations deal with longword data stored in canonical form in quadword registers. The canonical form has the longword data in the low 32 bits (0-31) of the register, with bit 31 replicated in the high 32 bits (32-63). Note that the canonical form is the same for both signed and unsigned longword data.

B.1.1 Canonical Form Operands

To create a canonical form operand from longword data, use the `ldl`, `ldl_l`, or `uldl` instruction.

To create a canonical form operand from a constant, use the `ldil` instruction. The `ldil` instruction is a macro instruction that expands into a series of instructions, including the `lda` and `ldah` instructions.

B.2 Longword Instructions

The Alpha architecture includes the following longword instructions:

- Load Longword (`ldl`)
- Load Longword Locked (`ldl_l`)
- Store Longword (`stl`)
- Store Longword Conditional (`stl_c`)
- Add Longword (`addl, addlv`)
- Subtract Longword (`subl, sublv`)
- Multiply Longword (`mull, mullv`)

– Scaled Longword Add (`s4addl, s8addl`)
– Scaled Longword Subtract (`s4subl, s8subl`)

In addition, the assembler provides the following longword macro instructions:

– Divide Longword (`divl, divlu`)
– Remainder Longword (`reml, remlu`)
– Negate Longword (`negl, neglv`)
– Unaligned Load Longword (`uldl`)
– Load Immediate Longword (`ldil`)
– Absolute Value Longword (`absl`)
– Sign-Extension Longword (`sextl`)

All longword instructions, with the exception of `stl` and `stl_c`, generate results in canonical form.

All longword instructions that have source operands produce correct results regardless of whether the data items in the source registers are in canonical form.

See Chapter 3 for a detailed description of the longword instructions.

B.3 Quadword Instructions for Longword Operations

The following quadword instructions, if presented with two canonical longword operands, produce a canonical longword result:

– Logical AND (`and`)
– Logical OR (`bis`)
– Logical Exclusive OR (`xor`)
– Logical OR NOT (`ornot`)
– Logical Equivalence (`eqv`)
– Conditional Move (`cmov`*xx*)
– Compare (`cmp`*xx*)
– Conditional Branch (`b`*xx*)
– Arithmetic Shift Right (`sra`)

Note that these instructions, unlike the longword instructions, must have operands in canonical form to produce correct results.

See Chapter 3 for a detailed description of the quadword instructions.

B.4 Logical Shift Instructions

No instructions, either machine or macro, exist for performing logical shifts on canonical longwords.

To perform a logical shift left, the following instruction sequence can be used:

```
sll  $rx, xx, $ry    # noncanonical result
addl $ry, 0, $ry     # sign-extend bit-31
```

To perform a logical shift right, the following instruction sequence can be used:

```
zap  $rx, 0xf0, $ry  # noncanonical result
srl  $ry, xx, $ry    # if xx >= 1, bring in zeros
addl $ry, 0, $ry     # sign-extend bit-31
```

Note that the `addl` instruction is not needed if the shift count in the previous sequence is guaranteed to be non-zero.

B.5 Conversions to Quadword

A signed longword value in canonical form is also a proper signed quadword value and no conversions are needed.

An unsigned longword value in canonical form is not a proper unsigned quadword value. To convert an unsigned longword to a quadword, the following instruction sequence can be used:

```
zap  $rx, 0xf0, $ry  # clear bits 32-63
```

B.6 Conversions to Longword

To convert a quadword value to either a signed or unsigned longword, the following instruction sequence can be used:

```
addl $rx, 0, $ry     # sign-extend bit-31
```

Basic Machine Definition C

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

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

C.1 Implicit Register Use

Register `$28` (`$at`) is reserved as a temporary register for use by the assembler.

Some assembly-language instructions require additional temporary registers. For these instructions, the assembler uses one or more of the general-purpose temporary registers (`t0` – `t12`). The following table lists the instructions that require additional temporary registers and the specific registers that they use:

Instruction	Registers Used
ldb	AT,t9
ldbu	AT,t9
ldw	AT,t9
ldwu	AT,t9
stb	AT,t9,t10
stw	AT,t9,t10
ustw	AT,t9,t10,t11,t12
ustl	AT,t9,t10,t11,t12
ustq	AT,t9,t10,t11,t12
uldw	AT,t9,t10
uldwu	AT,t9,t10
uldl	AT,t9,t10
uldq	AT,t9,t10
divl	AT,t9,t10,t11,t12

Instruction	Registers Used
divq	AT,t9,t10,t11,t12
divlu	AT,t9,t10,t11,t12
divqu	AT,t9,t10,t11,t12
reml	AT,t9,t10,t11,t12
remq	AT,t9,t10,t11,t12
remlu	AT,t9,t10,t11,t12
remqu	AT,t9,t10,t11,t12

The registers that equate to the software names (from `regdef.h`) in the preceding table are as follows:

Software Name	Register
AT	$28 or $at
t9	$23
t10	$24
t11	$25
t12 or pv	$27

Note

The `div` and `rem` instructions destroy the contents of `t12` only if the third operand is a register other than `t12`. See Section C.5 for more details.

C.2 Addresses

If you use an address as an operand and it references a data item that does not have an absolute address in the range -32768 to 32767, the assembler may generate a machine-code instruction to load the address of the data (from the literal address section) into `$at`.

The assembler's `ldgp` (load global pointer) instruction generates an `lda` and `ldah` instruction. The assembler requires the `ldgp` instruction because `ldgp` couples relocation information with the instruction.

C.3 Immediate Values

If you use an immediate value as an operand and the immediate value falls outside the range -32768 to 32767 for the `ldil` and `ldiq` instructions or the range 0 – 255 for other instructions, multiple machine instructions are generated to load the immediate value into the destination register or `$at`.

C.4 Load and Store Instructions

There are no single machine-code instructions for loading and storing unaligned data or data less than 32 bits. The following assembler instructions generate multiple machine instructions:

- Load Byte (`ldb`)
- Load Byte Unsigned (`ldbu`)
- Load Word (`ldw`)
- Load Word Unsigned (`ldwu`)
- Unaligned Load Word (`uldw`)
- Unaligned Load Word Unsigned (`uldwu`)
- Unaligned Load Longword (`uldl`)
- Unaligned Load Quadword (`uldq`)
- Store Byte (`stb`)
- Store Word (`stw`)
- Unaligned Store Word (`ustw`)
- Unaligned Store Longword (`ustl`)
- Unaligned Store Quadword (`ustq`)

Signed loads may require one more instruction than an unsigned load.

C.5 Integer Arithmetic Instructions

Multiply operations using constant powers of two are turned into `sll` or scaled add instructions.

There are no machine instructions for performing integer division (`divl`, `divlu`, `divq`, and `divqu`) or remainder operations (`reml`, `remlu`, `remq`, and `remqu`). The machine instructions generated for these assembler instructions depend on the operands specified on the instructions.

Division and remainder operations involving constant values are replaced by an instruction sequence that depends on the data type of the numerator and the value of the constant.

Division and remainder operations involving nonconstant values are replaced with a procedure call to a library routine to perform the operation. The library routines are in the C run-time library (libc). The library routines use a nonstandard parameter passing mechanism. The first operand is passed in register `t10` and the second operand is passed in `t11`. The result is returned in `t12`. If the operands specified are other than those just described, the assembler moves them to the correct registers. The library routines expect the return address in `t9`; therefore, a routine that uses divide instructions does not need to save register `ra` just because it uses divide instructions.

The `absl` and `absq` (absolute value) instructions generate two machine instructions.

C.6 Floating-Point Load Immediate Instructions

There are no floating-point instructions that accept an immediate value (except for 0.0). Whenever the assembler encounters a floating-point load immediate instruction, the immediate value is stored in the data section and a load instruction is generated to load the value.

C.7 One-to-One Instruction Mappings

Some assembler instructions generate single machine instructions. The following table lists these assembler instructions and their equivalent machine instructions:

Assembler Instruction		**Machine Instruction**	
andnot	$rx,$ry,$rz	bic	$rx,$ry,$rz
clr	$rx	bis	$31,$31,$rx
fabs	$fx,$fy	cpys	$f31,$fx,$fy
fclr	$fx	cpys	$f31,$f31,$fx
fmov	$fx,$fy	cpys	$fx,$fx,$fy
fneg	$fx,$fy	cpysn	$fx,$fx,$fy
fnop		cpys	$f31,$f31,$f31
mov	$rx,$ry	bis	$rx,$rx,$ry
mov	*val_immed*,$rx	bis	$31,*val_immed*,$rx
negf	$fx,$fy	subf	$f31,$fx,$fy
negfs	$fx,$fy	subfs	$f31,$fx,$fy
negg	$fx,$fy	subg	$f31,$fx,$fy
neggs	$fx,$fy	subgs	$f31,$fx,$fy
negl	$rx,$ry	subl	$31,$rx,$ry
neglv	$rx,$ry	sublv	$31,$rx,$ry
negq	$rx,$ry	subq	$31,$rx,$ry
negqv	$rx,$ry	subqv	$31,$rx,$ry
negs	$fx,$fy	subs	$f31,$fx,$fy
negssu	$fx,$fy	subssu	$f31,$fx,$fy
negt	$fx,$fy	subt	$f31,$fx,$fy
negtsu	$fx,$fy	subtsu	$f31,$fx,$fy
nop		bis	$31,$31,$31
not	$rx,$ry	ornot	$31,$rx,$ry
or	$rx,$ry,$rz	bis	$rx,$ry,$rz

Assembler Instruction		Machine Instruction	
sextl	$rx,$ry	addl	$rx,0,$ry
unop		ldq_u	$31,0($sp)
xornot	$rx,$ry,$rz	eqv	$rx,$ry,$rz

PALcode Instruction Summaries D

This appendix summarizes the Privileged Architecture Library (PALcode) instructions that are required to support an Alpha AXP system.

By including the file `pal.h` (use `#include <alpha/pal.h>`) in your assembly language program, you can use the symbolic names for the PALcode instructions.

D.1 Unprivileged PALcode Instructions

Table D-1 describes the unprivileged PALcode instructions.

Table D-1: Unprivileged PALcode Instructions

Symbolic Name	Number	Operation and Description
PAL_bpt	0x80	Break Point Trap – switches mode to kernel mode, builds a stack frame on the kernel stack, and dispatches to the breakpoint code.
PAL_bugchk	0x81	Bugcheck – switches mode to kernel mode, builds a stack frame on the kernel stack, and dispatches to the breakpoint code.
PAL_callsys	0x83	System call – switches mode to kernel mode, builds a callsys stack frame, and dispatches to the system call code.
PAL_gentrap	0xaa	Generate Trap – switches mode to kernel, builds a stack frame on the kernel stack, and dispatches to the gentrap code.
PAL_imb	0x86	I-Stream Memory Barrier – makes the I-cache coherent with main memory.
PAL_rduniq	0x9e	Read Unique – returns the contents of the process unique register.
PAL_wruniq	0x9f	Write Unique – writes the process unique register.

D.2 Privileged PALcode Instructions

The privileged PALcode instructions can be called only from kernel mode. They provide an interface to control the privileged state of the machine.

Table D-2 describes the privileged PALcode instructions.

Table D-2: Privileged PALcode Instructions

Symbolic Name	Number	Operation and Description
`PAL_halt`	0x00	Halt Processor – stops normal instruction processing. Depending on the halt action setting, the processor can either enter console mode or the restart sequence.
`PAL_rdps`	0x36	Read Process Status – return the current process status.
`PAL_rdusp`	0x3a	Read User Stack Pointer – reads the user stack pointer while in kernel mode and returns it.
`PAL_rdval`	0x32	Read System Value – reads a 64-bit per-processor value and returns it.
`PAL_rtsys`	0x3d	Return from System Call – pops the return address, the user stack pointer, and the user global pointer from the kernel stack. It then saves the kernel stack pointer, sets mode to user mode, enables interrupts, and jumps to the address popped off the stack.
`PAL_rti`	0x3f	Return from Trap, Fault, or Interrupt – pops certain registers from the kernel stack. If the new mode is user mode, the kernel stack is saved and the user stack is restored.
`PAL_swpctx`	0x30	Swap Privileged Context – saves the current process data in the current process control block (PCB). Then it switches to the PCB and loads the new process context.
`PAL_swpipl`	0x35	Swap IPL – returns the current IPL value and sets the IPL.
`PAL_tbi`	0x33	TB Invalidate – removes entries from the instruction and data translation buffers when the mapping entries change.

Table D-2: (continued)

Symbolic Name	Number	Operation and Description
PAL_whami	0x3c	Who Am I – returns the process number for the current processor. The processor number is in the range 0 to the number of processors minus one (0..numproc-1) that can be configured into the system.
PAL_wrfen	0x2b	Write Floating-Point Enable – writes a bit to the floating-point enable register.
PAL_wrkgp	0x37	Write Kernel Global Pointer – writes the kernel global pointer internal register.
PAL_wrusp	0x38	Write User Stack Pointer – writes a value to the user stack pointer while in kernel mode.
PAL_wrval	0x31	Write System Value – writes a 64-bit per-processor value.
PAL_wrvptptr	0x2d	Write Virtual Page Table Pointer – writes a pointer to the virtual page table pointer (vptptr).

Index

A

absl instruction, 3–10, 3–11
absq instruction, 3–10, 3–11
addf instruction, 4–11, 4–13
addg instruction, 4–11, 4–13
addl instruction, 3–10, 3–12
addlv instruction, 3–10, 3–12
addq instruction, 3–10, 3–12
addqv instruction, 3–10, 3–12
addresses, C–2
addressing
  formats, 2–9
adds instruction, 4–11, 4–13
addt instruction, 4–11, 4–13
.aent directive, 5–2
.alias directive, 5–2
.align directive, 5–2
and instruction, 3–19
andnot instruction, 3–19, 3–20
archive files
  object files, 7–26
arithmetic instructions
  descriptions of, 3–10
  formats, 3–9
.ascii directive, 5–3
.asciiz directive, 5–3
assembler directives, 5–1 to 5–13
auxiliary symbol table, 8–5
auxiliary symbols, 8–18

B

base addresses
  calculation and use, 9–2
beq instruction, 3–24, 3–25
bge instruction, 3–24, 3–26
.bgnb directive, 5–3
bgt instruction, 3–24, 3–26
bic instruction, 3–19, 3–20
big endian
  byte ordering, 1–2
binding, lazy, 9–13
bis instruction, 3–19
blbc instruction, 3–24, 3–26
blbs instruction, 3–24, 3–26
ble instruction, 3–24, 3–26
blt instruction, 3–24, 3–26
bne instruction, 3–24, 3–25
br instruction, 3–25, 3–26
bsr instruction, 3–25, 3–26
.byte directive, 5–3
byte ordering
  big endian, 1–2
  little endian, 1–2

byte-manipulation instructions
descriptions of, 3–28
formats, 3–27

C

C programs
calling, 6–1
-S compilation option, 6–14
call_pal instruction, 3–33
calls
to position independent functions, 9–13
to programs in other languages, 6–1
chopped rounding (IEEE), 4–7
chopped rounding (VAX), 4–6
clr instruction, 3–10, 3–11
cmoveq instruction, 3–23
cmovge instruction, 3–23, 3–24
cmovgt instruction, 3–23, 3–24
cmovlbc instruction, 3–23, 3–24
cmovlbs instruction, 3–23, 3–24
cmovle instruction, 3–23
cmovlt instruction, 3–23
cmovne instruction, 3–23
cmpbge instruction, 3–28, 3–29
cmpeq instruction, 3–21, 3–22
cmpgeq instruction, 4–14
cmpgle instruction, 4–14, 4–15
cmpglt instruction, 4–14, 4–15
cmple instruction, 3–21, 3–22
cmplt instruction, 3–21, 3–22
cmpteq instruction, 4–14
cmptle instruction, 4–14, 4–15
cmptlt instruction, 4–14, 4–15
cmptun instruction, 4–14, 4–15
cmpule instruction, 3–21, 3–22
cmpult instruction, 3–21, 3–22
code optimization, 6–1
.comm directive, 5–4
comments, 2–1
compilation options
-S option, 6–14
.conflict section, 9–22
constants
floating-point, 2–2
scalar, 2–2
string, 2–3
control instructions
descriptions of, 3–25
formats, 3–24
counters, 6–4
cpys instruction, 4–15, 4–16
cpyse instruction, 4–15, 4–16
cpysn instruction, 4–15, 4–16
cvtdg instruction, 4–11, 4–13
cvtgd instruction, 4–11, 4–13
cvtgf instruction, 4–11, 4–13
cvtgq instruction, 4–11, 4–13
cvtlq instruction, 4–11, 4–13
cvtqf instruction, 4–11, 4–13
cvtqg instruction, 4–11, 4–13
cvtql instruction, 4–11, 4–13
cvtqs instruction, 4–11, 4–13
cvtqt instruction, 4–11, 4–13
cvtst instruction, 4–11, 4–13
cvttq instruction, 4–11, 4–13
cvtts instruction, 4–11, 4–13

D

.data directive, 5–4
data segments
sections contained in, 9–2
data types, 2–7
dense numbers, 8–3
.d_floating directive, 5–4
directives
assembler directives, 5–1 to 5–13
divf instruction, 4–11, 4–13
divg instruction, 4–11, 4–13
divl instruction, 3–10, 3–15
divlu instruction, 3–10, 3–16
divq instruction, 3–10, 3–16
divqu instruction, 3–10, 3–17
divs instruction, 4–11, 4–13
divt instruction, 4–11, 4–13
.double directive, 5–4
dynamic linking, 9–4
dynamic loader
use, 9–4
dynamic relocation section
See .rel.dyn section
dynamic rounding mode, 4–4
.dynamic section
contents, 9–5
ordering for quickstart, 9–22
dynamic string section
See .dynstr section
dynamic symbol section
See .dynsym section
.dynstr section, 9–19
.dynsym section, 9–15
relationship with .got section, 9–17

E

.edata directive, 5–4
.eflag directive, 5–4
.end directive, 5–5
.endb directive, 5–5
.endr directive, 5–5
.ent directive, 5–5
eqv instruction, 3–19, 3–20
.err directive, 5–5
excb instruction, 3–33
exceptions
floating-point, 1–6
main processor, 1–6
executable files
loading considerations, 9–3
offset alignment, 9–3
expression operators, 2–6
expressions
operator precedence rules, 2–7
type propagation rules, 2–8
extbl instruction, 3–28, 3–29
.extern directive, 5–5
external string table, 8–6
external symbol table, 8–21
external symbols, 8–8
extlh instruction, 3–28, 3–30
extll instruction, 3–28, 3–29
extqh instruction, 3–28, 3–30
extql instruction, 3–28, 3–29
extwh instruction, 3–28, 3–30
extwl instruction, 3–28, 3–29

F

fabs instruction, 4–11, 4–12
fbeq instruction, 4–17
fbge instruction, 4–17
fbgt instruction, 4–17
fble instruction, 4–17
fblt instruction, 4–17
fbne instruction, 4–17
fclr instruction, 4–11, 4–12
fcmoveq instruction, 4–15, 4–16
fcmovge instruction, 4–15, 4–16
fcmovgt instruction, 4–15, 4–16
fcmovle instruction, 4–15, 4–16
fcmovlt instruction, 4–15, 4–16
fcmovne instruction, 4–15, 4–16
fetch instruction, 3–33
fetch_m instruction, 3–33, 3–34
.f_floating directive, 5–5
file descriptor table, 8–6, 8–21
.file directive, 5–6
file header
  file header magic field (f_magic), 7–6
  flags (s_flags), 7–8
.float directive, 5–6
floating point
  rounding mode qualifiers, 4–7
  trapping mode qualifiers, 4–8
floating-point
  constants, 2–2
floating-point arithmetic instructions
  descriptions of, 4–12
  formats, 4–11
floating-point control instructions
  descriptions of, 4–17
  formats, 4–17
floating-point control register, 4–4
floating-point instruction set, 4–1
floating-point load instructions
  descriptions of, 4–10
  formats, 4–9
floating-point move instructions
  descriptions of, 4–16
  formats, 4–15
floating-point relational instructions
  descriptions of, 4–14
  formats, 4–14
floating-point rounding modes, 4–6
floating-point special instructions
  descriptions of, 4–18
  formats, 4–17
floating-point store instructions
  descriptions of, 4–10
  formats, 4–9
.fmask directive, 5–6
fmov instruction, 4–15, 4–16
fnop instruction, 4–18
FPCR
  floating-point control register, 4–4
.frame directive, 5–6
functions
  calling position-independent functions, 9–13

G

.g_floating directive, 5–6
.gjsrlive directive, 5–7
.gjsrsaved directive, 5–7
global offset table
  See .got section
.globl directive, 5–7
.got section, 9–12
  relationship with .dynsym section, 9–17

.got section (cont.)
relationship with .lita section, 7–12
.gprel32 directive, 5–7
.gretlive directive, 5–7

H

.hash section, 9–18
hash table section
See .hash section

I

identifiers, 2–1
immediate values, C–2
implicit register use, C–1
infinity
rounding toward plus or minus infinity, 4–6, 4–7
insbl instruction, 3–28, 3–30
inslh instruction, 3–28, 3–31
insll instruction, 3–28, 3–30
insqh instruction, 3–28, 3–31
insql instruction, 3–28, 3–31
instruction summaries, A–1
inswh instruction, 3–28, 3–31
inswl instruction, 3–28, 3–30
integer arithmetic instructions, C–3

J

jmp instruction, 3–25, 3–26
jsr instruction, 3–25, 3–26
jsr_coroutine instruction, 3–25, 3–27

K

keyword statements, 2–5

L

.lab directive, 5–7
label definitions, 2–5
language interfaces, 6–2
lazy binding, 9–13
.lcomm directive, 5–7
lda instruction, 3–2, 3–4
ldah instruction, 3–3, 3–7
ldb instruction, 3–2, 3–4
ldbu instruction, 3–2, 3–4
ldf instruction, 4–10
ldg instruction, 4–10
ldgp instruction, 3–3, 3–7
ldid instruction, 4–10
ldif instruction, 4–10
ldig instruction, 4–10
ldil instruction, 3–3, 3–7
ldiq instruction, 3–3, 3–6, 3–7
ldis instruction, 4–10
ldit instruction, 4–10
ldl instruction, 3–2, 3–5
ldl_l instruction, 3–2, 3–5
ldq instruction, 3–2, 3–5
ldq_l instruction, 3–2, 3–6
ldq_u instruction, 3–2, 3–6
lds instruction, 4–10
ldt instruction, 4–10
ldw instruction, 3–2, 3–4
ldwu instruction, 3–2, 3–4
.liblist section, 9–20
line number table, 8–3

linkage conventions
examples, 6–10
general, 6–3
language interfaces, 6–14
memory allocation, 6–17
linker defined symbols, 7–27
linking, dynamic, 9–4
.lita section, 6–5
relationship with .got section, 7–12
little endian
byte ordering, 1–2
.livereg directive, 5–8
load and store instructions, 3–2, C–3
load instructions
descriptions of, 3–3
formats, 3–2
loader, default, 9–4
loader, dynamic
use, 9–4
loading considerations, 9–3
loading programs, 9–3
.loc directive, 5–8
local string table, 8–5
local symbol table, 8–4
logical instructions
descriptions of, 3–19
formats, 3–18
.long directive, 5–8

M

.mask directive, 5–9
mb instruction, 3–33, 3–34
mf_fpcr instruction, 4–18
minus infinity
rounding toward (IEEE), 4–6
mnemonic
definition, 2–5
mov instruction, 3–23
move instructions
descriptions of, 3–23
formats, 3–22
mskbl instruction, 3–28, 3–31
msklh instruction, 3–28, 3–32
mskll instruction, 3–28, 3–31
mskqh instruction, 3–28, 3–32
mskql instruction, 3–28, 3–32
mskwh instruction, 3–28, 3–32
mskwl instruction, 3–28, 3–31
mt_fpcr instruction, 4–18
mulf instruction, 4–11, 4–13
mulg instruction, 4–11, 4–13
mull instruction, 3–10, 3–13
mullv instruction, 3–10, 3–13
mulq instruction, 3–10, 3–13
mulqv instruction, 3–10, 3–14
muls instruction, 4–11, 4–13
mult instruction, 4–11, 4–13

N

negf instruction, 4–11, 4–12
negg instruction, 4–11, 4–12
negl instruction, 3–10, 3–11
neglv instruction, 3–10, 3–11
negq instruction, 3–10, 3–11
negqv instruction, 3–10, 3–11
negs instruction, 4–11, 4–12
negt instruction, 4–11, 4–12
NMAGIC files, 7–6
segment access permissions, 9–2
.noalias directive, 5–9

nop instruction, 3–33, 3–34
normal rounding (IEEE)
unbiased round to nearest, 4–6
normal rounding (VAX)
biased, 4–6
not instruction, 3–19
null statements, 2–5

O

object file format, 7–1
object file types
demand paged (ZMAGIC) files, 7–24
impure format (OMAGIC) files, 7–21
shared text (NMAGIC) files, 7–22
object files
See also executable files
See also shared object files
archived object files, 7–26
data segment contents, 9–2
loading
boundary constraints, 7–21
description, 7–26
text segment contents, 9–2
OMAGIC files, 7–21
segment access permissions, 9–2
operator evaluation order
precedence rules, 2–7
operators, expression, 2–6
optimization
optimizing assembly code, 6–1
optimization symbol table, 8–5
.option directive, 5–9
optional header, 7–6
optional header magic field (magic), 7–6
or instruction, 3–19, 3–20
ornot instruction, 3–19, 3–20

P

PALcode
instruction summaries, D–1
performance
optimizing assembly code, 6–1
plus infinity
rounding toward (IEEE), 4–7
position independent functions
calling, 9–13
precedence rules
operator evaluation order, 2–7
procedure descriptor table, 8–4, 8–13
program loading, 9–3
program model, 6–2
program optimization, 6–1
program segments
access permissions, 9–2
.prologue directive, 5–9

Q

.quad directive, 5–9
quickstart, 9–20
section ordering constraints, 9–22

R

.rdata directive, 5–10
register use, 6–3
registers
floating-point, 1–2, 6–4
format, 1–3
general, 1–1
integer, 1–1, 6–3

relational instructions
descriptions of, 3–22
formats, 3–21
relative file descriptor table, 8–7
.rel.dyn section, 9–17
ordering for quickstart, 9–22
reml instruction, 3–10, 3–17
remlu instruction, 3–10, 3–17
remq instruction, 3–10, 3–18
remqu instruction, 3–10, 3–18
.repeat directive, 5–10
ret instruction, 3–25, 3–27
rounding mode
chopped rounding (IEEE), 4–7
chopped rounding (VAX), 4–6
dynamic rounding qualifier, 4–4
floating-point instruction qualifiers, 4–7
floating-point rounding modes, 4–6 to 4–7
FPCR control, 4–4
normal rounding (IEEE, unbiased), 4–6
normal rounding (VAX, biased), 4–6
rounding toward minus infinity (IEEE), 4–6
rounding toward plus infinity (IEEE), 4–7
rpcc instruction, 3–33, 3–34

S

-S compilation option, 6–14
.s files, 6–14
s4addl instruction, 3–10, 3–12
s4addq instruction, 3–10, 3–13
s4subl instruction, 3–10, 3–15
s4subq instruction, 3–10, 3–15
s8addl instruction, 3–10, 3–13
s8addq instruction, 3–10, 3–13
s8subl instruction, 3–10, 3–15
s8subq instruction, 3–10, 3–15
.save_ra directive, 5–10
scalar constants, 2–2
.sdata directive, 5–10
section data, 7–7, 7–10
section headers
flags (s_flags), 7–8
section name (s_name), 7–7
section relocation information
assembler and linker processing, 7–15
relocation entry, 7–12
relocation table entry, 7–16
segments
access permissions for program segments, 9–2
alignment of data segements, 9–3
alignment of text segments, 9–3
segments, text
sections contained in, 9–2
.set directive, 5–10
sextl instruction, 3–10, 3–12
.s_floating directive, 5–11
shared object files
dependencies, 9–11
dynamic linking, 9–4
dynamic section, 9–5
initialization and termination functions, 9–20
quickstart, 9–20
shared object list section
See .liblist section
shift instructions
descriptions of, 3–19
formats, 3–18
sll instruction, 3–19, 3–20
.space directive, 5–11

special-purpose instructions
descriptions of, 3–33
formats, 3–33
sra instruction, 3–19, 3–21
srl instruction, 3–19, 3–20
stack frame, 6–7
statements, 2–5
stb instruction, 3–3, 3–8
stf instruction, 4–10
stg instruction, 4–10
stl instruction, 3–8
stl_c instruction, 3–3, 3–8
store instructions
descriptions of, 3–8
formats, 3–2
stq instruction, 3–3, 3–8
stq_c instruction, 3–3, 3–9
stq_u instruction, 3–3, 3–9
string constants, 2–3
.struct directive, 5–11
sts instruction, 4–10
stt instruction, 4–10
stw instruction, 3–3, 3–8
subf instruction, 4–11, 4–13
subg instruction, 4–11, 4–13
subl instruction, 3–10, 3–14
sublv instruction, 3–10, 3–14
subq instruction, 3–10, 3–14
subqv instruction, 3–10, 3–14
subs instruction, 4–11, 4–13
subt instruction, 4–11, 4–13
symbol table, 8–1
format of entries, 8–8
line numbers in, 8–3
symbolic equate, 5–12
symbolic header, 8–3, 8–8

T

.text directive, 5–12
text segments
alignment, 9–3
sections contained in, 9–2
.t_floating directive, 5–12
trapb instruction, 3–33, 3–34
trapping mode
floating-point instruction qualifiers, 4–8
type propagation rules, 2–8

U

.ugen directive, 5–12
uldl instruction, 3–2, 3–7
uldq instruction, 3–2, 3–7
uldw instruction, 3–2, 3–6
uldwu instruction, 3–2
umulh instruction, 3–10, 3–15
unop instruction, 3–33, 3–34
ustl instruction, 3–3, 3–9
ustq instruction, 3–3, 3–9
ustw instruction, 3–3, 3–9

V

.verstamp directive, 5–12
.vreg directive, 5–12

W

.weakext directive, 5–12
wmb instruction, 3–33, 3–34
.word directive, 5–12

X

xor instruction, 3–19, 3–20
xornot instruction, 3–19, 3–20

Z

zap instruction, 3–28, 3–32
zapnot instruction, 3–28, 3–32
ZMAGIC files, 7–24
  segment access permissions, 9–2

How to Order Additional Documentation

Technical Support

If you need help deciding which documentation best meets your needs, call 800-343-4040 before placing your electronic, telephone, or direct mail order.

Electronic Orders

To place an order at the Electronic Store, dial 800-234-1998 using a 1200- or 2400-bps modem from anywhere in the USA, Canada, or Puerto Rico. If you need assistance using the Electronic Store, call 800-DIGITAL (800-344-4825).

Telephone and Direct Mail Orders

Your Location	Call	Contact
Continental USA, Alaska, or Hawaii	800-DIGITAL	Digital Equipment Corporation P.O. Box CS2008 Nashua, New Hampshire 03061
Puerto Rico	809-754-7575	Local Digital subsidiary
Canada	800-267-6215	Digital Equipment of Canada Attn: DECdirect Operations KAO2/2 P.O. Box 13000 100 Herzberg Road Kanata, Ontario, Canada K2K 2A6
International	—————	Local Digital subsidiary or approved distributor
Internal[a]	—————	SSB Order Processing - WMO/E15 *or* Software Supply Business Digital Equipment Corporation Westminster, Massachusetts 01473

[a] For internal orders, you must submit an Internal Software Order Form (EN-01740-07).

Reader's Comments

DEC OSF/1
Assembly Language Programmer's Guide
AA-PS31A-TE

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

Please rate this manual:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

What would you like to see more/less of? ______________________________

____

What do you like best about this manual? ______________________________

____

What do you like least about this manual? ______________________________

____

Please list errors you have found in this manual:

Page	Description
________	____
________	____
________	____
________	____
________	____

Additional comments or suggestions to improve this manual:

____

____

____

What version of the software described by this manual are you using? ____________

Name/Title ______________________________ Dept. ______________

Company ____________________________________ Date __________

Mailing Address ____

__________________ Email __________________ Phone ______________

Do Not Tear – Fold Here and Tape

digital™

No Postage
Necessary
if Mailed in the
United States

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
OPEN SOFTWARE PUBLICATIONS MANAGER
ZKO3–3/Y32
110 SPIT BROOK ROAD
NASHUA NH 03062–9987

Do Not Tear – Fold Here

Cut
Along
Dotted
Line

Reader's Comments

DEC OSF/1
Assembly Language Programmer's Guide
AA-PS31A-TE

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

Please rate this manual:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

What would you like to see more/less of? ______________________________________

____

What do you like best about this manual? ______________________________________

____

What do you like least about this manual? _____________________________________

____

Please list errors you have found in this manual:

Page	Description
________	____
________	____
________	____
________	____
________	____

Additional comments or suggestions to improve this manual:

____

____

____

What version of the software described by this manual are you using? _______________

Name/Title ________________________________ Dept. ________________

Company ____ Date __________

Mailing Address ____

______________________ Email ____________________ Phone ______________

Do Not Tear – Fold Here and Tape

digital™

No Postage
Necessary
if Mailed in the
United States

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
OPEN SOFTWARE PUBLICATIONS MANAGER
ZKO3−3/Y32
110 SPIT BROOK ROAD
NASHUA NH 03062−9987

Do Not Tear – Fold Here

Cut
Along
Dotted
Line