Tru64 UNIX

Assembly Language Programmer's Guide

Part Number: AA-RH9LC-TE

August 2000

Product Version: Tru64 UNIX Version 5.1 or higher

This manual describes the assembly language supported by the Tru64 UNIX compiler system.

Hewlett-Packard Company
Palo Alto, California

© 2000 Hewlett-Packard Company

Microsoft® and Windows® are trademarks of Microsoft Corporation in the U.S. and/or other countries. UNIX® and The Open Group™ are trademarks of The Open Group in the U.S. and/or other countries. All other product names mentioned herein may be trademarks of their respective companies.

Portions of this document © MIPS Computer Systems, Inc., 1990.

Confidential computer software. Valid license from Compaq Computer Corporation, a wholly owned subsidiary of Hewlett-Packard Company, required for possession, use, or copying. Consistent with FAR 12.211 and 12.212, Commercial Computer Software, Computer Software Documentation, and Technical Data for Commercial Items are licensed to the U.S. Government under vendor's standard commercial license.

None of Compaq, HP, or any of their subsidiaries shall be liable for technical or editorial errors or omissions contained herein. The information is provided "as is" without warranty of any kind and is subject to change without notice. The warranties for HP or Compaq products are set forth in the express limited warranty statements accompanying such products. Nothing herein should be construed as constituting an additional warranty.

Contents

About This Manual

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 1–3
1.3.1 Aligned Data Operations 1–4
1.3.2 Unaligned Data Operations 1–4
1.4 Exceptions 1–5
1.4.1 Main Processor Exceptions 1–5
1.4.2 Floating-Point Processor Exceptions 1–5

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 2–4
2.6 Statements 2–5
2.6.1 Labels 2–5
2.6.2 Null Statements 2–5
2.6.3 Keyword Statements 2–5
2.6.4 Relocation Operands 2–6
2.7 Expressions 2–8
2.7.1 Expression Operators 2–8
2.7.2 Expression Operator Precedence Rules 2–9
2.7.3 Data Types 2–10
2.7.4 Type Propagation in Expressions 2–11
2.8 Address Formats 2–12

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 3–6
3.2 Arithmetic Instructions 3–8
3.3 Logical and Shift Instructions 3–14
3.4 Relational Instructions 3–16
3.5 Move Instructions 3–17
3.6 Control Instructions 3–18
3.7 Byte-Manipulation Instructions 3–21
3.8 Special-Purpose Instructions 3–24

4 Floating-Point Instruction Set

4.1 Background Information on Floating-Point Operations 4–2
4.1.1 Floating-Point Data Types 4–2
4.1.2 Floating-Point Control Register 4–3
4.1.3 Floating-Point Exceptions 4–4
4.1.4 Floating-Point Rounding Modes 4–5
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 4–10
4.4 Floating-Point Relational Instructions 4–13
4.5 Floating-Point Move Instructions 4–14
4.6 Floating-Point Control Instructions 4–15
4.7 Floating-Point Special-Purpose Instructions 4–16

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 Coding 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

A Instruction Summaries

B 32-Bit Considerations

B.1 Canonical Form 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–3
C.4 Load and Store Instructions C–3
C.5 Integer Arithmetic Instructions C–4
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–1

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 1–3
4–1 Floating-Point Data Formats 4–3
4–2 Floating-Point Control Register 4–4

6–1 Sections and Location Counters for Nonshared Object Files ... 6–5
6–2 Stack Organization ... 6–8
6–3 Default Layout of Memory (User Program View) ... 6–18

Tables

2–1 Backslash Conventions ... 2–4
2–2 Expression Operators ... 2–9
2–3 Operator Precedence ... 2–10
2–4 Data Types ... 2–10
2–5 Address Formats ... 2–12
3–1 Load and Store Formats ... 3–2
3–2 Load Instruction Descriptions ... 3–4
3–3 Store Instruction Descriptions ... 3–6
3–4 Arithmetic Instruction Formats ... 3–8
3–5 Arithmetic Instruction Descriptions ... 3–10
3–6 Logical and Shift Instruction Formats ... 3–14
3–7 Logical and Shift Instruction Descriptions ... 3–15
3–8 Relational Instruction Formats ... 3–17
3–9 Relational Instruction Descriptions ... 3–17
3–10 Move Instruction Formats ... 3–18
3–11 Move Instruction Descriptions ... 3–18
3–12 Control Instruction Formats ... 3–19
3–13 Control Instruction Descriptions ... 3–19
3–14 Byte-Manipulation Instruction Formats ... 3–21
3–15 Byte-Manipulation Instruction Descriptions ... 3–22
3–16 Special-Purpose Instruction Formats ... 3–24
3–17 Special-Purpose Instruction Descriptions ... 3–25
4–1 Qualifier Combinations for Floating-Point Instructions ... 4–8
4–2 Load and Store Instruction Formats ... 4–9
4–3 Load and Store Instruction Descriptions ... 4–10
4–4 Arithmetic Instruction Formats ... 4–10
4–5 Arithmetic Instruction Descriptions ... 4–12
4–6 Relational Instruction Formats ... 4–13
4–7 Relational Instruction Descriptions ... 4–14
4–8 Move Instruction Formats ... 4–14
4–9 Move Instruction Descriptions ... 4–15
4–10 Control Instruction Formats ... 4–15
4–11 Control Instruction Descriptions ... 4–16
4–12 Special-Purpose Instruction Formats ... 4–16
4–13 Control Register Instruction Descriptions ... 4–16
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
A–1	Main Instruction Set Summary	A–2
A–2	Floating-Point Instruction Set Summary	A–7
A–3	Rounding and Trapping Modes	A–10
D–1	Unprivileged PALcode Instructions	D–1
D–2	Privileged PALcode Instructions	D–2

About This Manual

This manual describes the assembly language supported by the HP Tru64 UNIX 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, in the assembly language for the Tru64 UNIX compiler system, some instructions correspond to multiple machine instructions.

The assembler's primary purpose is to produce object modules from the assembly instructions generated by some high-level language compilers. As a result, the assembler lacks many functions that are normally present in assemblers designed to produce object modules from source programs coded in assembly language. It also includes some functions that are not found in such assemblers because of special requirements associated with the high-level language compilers.

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 — for example, a low-level I/O driver — in a way that might not be possible in C, Fortran-77, Pascal, or another high-level language.
- 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.

New and Changed Features

The major technical changes to the manual are as follows:

- The following directives are no longer supported and their descriptions have been deleted from *Chapter 5*: `.alias`, `.bgnb`, `.endb`, `.gjsrlive`, `.gjsrsaved`, `.livereg`, `.noalias`, `.ugen`, and `.vreg`.

- Descriptions of the following new directives have been added to *Chapter 5*: `.ident`, `.tlscomm`, `.tlsdate`, and `.tlslcomm`.

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.
Appendix A	Summarizes all assembler instructions.
Appendix B	Describes issues related to the processing of 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 system.

Related Documents

The following manuals provide additional information on many of the topics addressed in this manual:

- *Programmer's Guide*
- *The Alpha Architecture Reference Manual, 3rd Edition* (Butterworth-Heinemann Press, ISBN:1-55558-202-8)
- *Calling Standard for Alpha Systems*
- *Object File / Symbol Table Format Specification* (This manual is available as an HTML or PDF document on the documentation CD-ROM; it is not available in hardcopy.)

Icons on Tru64 UNIX Printed Manuals

The printed version of the Tru64 UNIX documentation uses letter icons on the spines of the manuals to help specific audiences quickly find the manuals

that meet their needs. (You can order the printed documentation from HP.) The following list describes this convention:

G	Manuals for general users
S	Manuals for system and network administrators
P	Manuals for programmers
R	Manuals for reference page users

Some manuals in the documentation help meet the needs of several audiences. For example, the information in some system manuals is also used by programmers. Keep this in mind when searching for information on specific topics.

The *Documentation Overview* provides information on all of the manuals in the Tru64 UNIX documentation set.

Reader's Comments

HP welcomes any comments and suggestions you have on this and other Tru64 UNIX manuals.

You can send your comments in the following ways:

- Fax: 603-884-0120 Attn: UBPG Publications, ZKO3-3/Y32
- Internet electronic mail: `readers_comment@zk3.dec.com`

  A Reader's Comment form is located on your system in the following location:

  `/usr/doc/readers_comment.txt`

Please include the following information along with your comments:

- The full title of the manual and the order number. (The order number appears on the title page of printed and PDF versions of a manual.)
- The section numbers and page numbers of the information on which you are commenting.
- The version of Tru64 UNIX that you are using.
- If known, the type of processor that is running the Tru64 UNIX software.

The Tru64 UNIX Publications group cannot respond to system problems or technical support inquiries. Please address technical questions to your local system vendor or to the appropriate HP technical support office. Information provided with the software media explains how to send problem reports to HP.

Conventions

file	Italic (slanted) type indicates variable values, placeholders, and function argument names.
[|] { | }	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.

1

Architecture-Based Considerations

This chapter describes programming considerations that are determined by the Alpha 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 systems and describes how memory organization affects them. See Section 6.3 for information on register use and linkage.

Alpha systems have the following types of registers:

- Integer registers (Section 1.1.1)
- Floating-point registers (Section 1.1.2)

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 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.

The `$gp` register (integer register `$29`) is available as a general register on some non-Alpha compiler systems when the `-G 0` compilation option is specified. It is not available as a general register on Alpha 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 by certain PALcode instructions. 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 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:

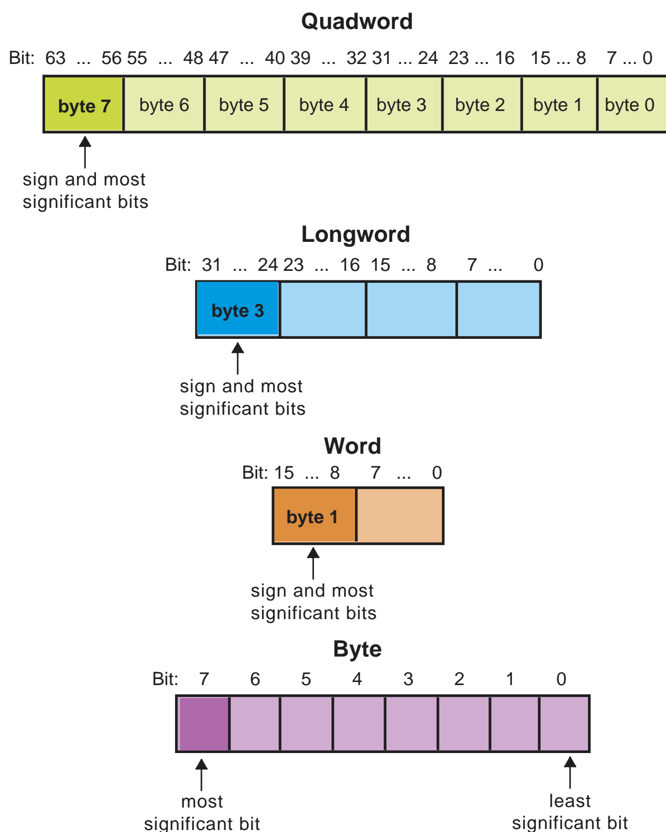
- Big-endian systems store the sign bit in the lowest address byte.
- Little-endian systems store the sign bit in the highest address byte.

Alpha 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. (Bit numbering is a software convention; no assembler instructions depend on it.)

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 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

The Alpha system detects some exceptions directly, and other exceptions are signaled as a result of specific tests that are inserted by the assembler.

The following sections describe exceptions that you may encounter during the execution of assembly programs. Only those exceptions that occur most frequently are described.

1.4.1 Main Processor Exceptions

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

- Address error exceptions 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 occur when arithmetic operations compute signed values and the destination lacks the precision to store the result.
- Bus exceptions occur when an address is invalid for the executing process.
- Divide-by-zero exceptions 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 include the following:
  - Magnitude subtraction of infinities, for example, (+INF) - (+INF).
  - Multiplication of 0 by INF with any signs.
  - Division of 0 by 0 or INF by INF with any signs.
  - Conversion of a binary floating-point number to an integer format, that is, only in those cases in which the conversion produces an overflow or an operand value of infinity or NaN. (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 occur when a divisor is zero.
- Overflow exceptions occur when a rounded floating-point result exceeds the destination format's largest finite number.

- Underflow exceptions 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 occur if the infinitely precise result differs from the rounded result.

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

2
Lexical Conventions

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 undefined identifier is 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 (Section 2.4.1)
- Floating-point constants (Section 2.4.2)
- String constants (Section 2.4.3)

2.4.1 Scalar Constants

The assembler interprets all scalar constants as two's complement numbers. Scalar constants can be any of the digits 0123456789abcdefABCDEF.

Scalar constants can be either decimal, hexadecimal, or octal constants:

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

A decimal integer that denotes the integral part of the floating-point value.

d2

A decimal integer that denotes the fractional part of the floating-point value.

d3

A decimal integer that denotes a power of 10.

The + symbol (plus sign) 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)
\X*nn*	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. Insert a 0 (zero) as the first character of the single-character hexadecimal number when this condition occurs.

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 keyword statement can include an optional label, an operation code (mnemonic or directive), and zero or more operands (with an optional comment following the last operand on the statement):

[*label*]: *opcode operand* [; *opcode operand*; ...] [# *comment*]

Some keyword statements also support relocation operands (see Section 2.6.4).

2.6.1 Labels

Labels can consist of label definitions or numeric values:

- A label definition consists of an identifier followed by a colon. (See Section 2.3 for the rules governing identifiers.) 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 numeric label is a single numeric value (1-255). Unlike label definitions, the value of a numeric label can be applied to any number of statements in a program. To reference a numeric label, put an `f` (forward) or a `b` (backward) immediately after the referencing digit in an instruction, for example, `br 7f` (which is a forward branch to numeric label 7). The reference directs the assembler to look for the nearest numeric 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 contains 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.

2.6.4 Relocation Operands

Relocation operands are generally useful in only two situations:

- In application programs in which the programmer needs precise control over scheduling
- In source code written for compiler development

Some macro instructions (for example, `ldgp`) require special coordination between the machine-code instructions and the relocation sequences given to the linker. By using the macro instructions, the assembler programmer relies on the assembler to generate the appropriate relocation sequences.

In some instances, the use of macro instructions may be undesirable. For example, a compiler that supports the generation of assembly language files may not want to defer instruction scheduling to the assembler. Such a compiler will want to schedule some or all of the machine-code instructions. To do this, the compiler must have a mechanism for emitting an object file's relocation sequences without using macro instructions. The mechanism for establishing these sequences is the relocation operand.

A relocation operand can be placed after the normal operand on an assembly language statement:

opcode operand relocation_operand

The *relocation_operand* has the following form:

!*relocation_type*! *sequence_number*

relocation_type

Any one of the following relocation types can be specified:

```
literal
lituse_base
lituse_bytoff
lituse_jsr
gpdisp
gprelhigh
gprellow
tlsliteral
tlshigh
tlslow
```

The relocation types must be enclosed within a pair of exclamation points (!) and are not case-sensitive. See the *Symbol Table/Object File Specification* manual for descriptions of the different types of relocation operations.

sequence_number

The sequence number is a numeric constant with a value range of 1 to 2147483647. The constant can be base 8, 10, or 16. Bases other than 10 require a prefix (see Section 2.4.1).

The following examples contain relocation operands in the source code:

- Example 1 — Referencing multiple `lituse_base` relocations:

```
# Equivalent C statement:
# sym1 += sym2  (Both external)

# Assembly statements containing macro instructions:
ldq   $1, sym1
ldq   $2, sym2
addq  $1, $2, $3
stq   $3, sym1

# Assembly statements containing machine-code instructions
# requiring relocation operands:
ldq   $1, sym1($gp)!literal!1
ldq   $2, sym2($gp)!literal!2

ldq   $3, sym1($1)!lituse_base!1
ldq   $4, sym2($1)!lituse_base!2
addq  $3, $4, $3
stq   $3, sym1($1)!lituse_base!1
```

  The assembler stores the `sym1` and `sym2` address constants in the `.lita` section.

  In this example, the code with relocation operands provides better performance than the other code because it saves on register usage and on the length of machine-code instruction sequences.

- Example 2 — Referencing an `ldgp` sequence that is scheduled inside a `lituse_base` relocation:

```
# Assembly statements containing macro instructions:
beq   $2, L
stq   $31, sym
ldgp  $gp, 0($27)

# Assembly statements containing machine-code instructions that
# require relocation operands:
ldq   $at, sym($gp)!literal!1
```

```
beq   $2, L           # crosses basic block boundary
ldah  $gp, 0($27)!gpdisp!2
stq   $31, sym($at)!lituse_base!1
lda   $gp, 0($gp)!gpdisp!2
```

In this example, the programmer has elected to schedule the load of the address of `sym` before the conditional branch.

- Example 3 — A routine call:

```
# Assembly statements containing macro instructions:
jsr   sym1
ldgp  $gp, 0($ra)

.extern sym1

.text

# Assembly statements containing machine-code instructions that
# require relocation operands:
ldq   $27, sym1($gp)!literal!1
jsr   $26, ($27), sym1!lituse_jsr!1
# as1 puts in an R_HINT for the jsr instruction
ldah  $gp, 0($ra)!gpdisp!2
lda   $gp, 0($gp)!gpdisp!2
```

In this example, the code with relocation operands does not provide any significant gains over the other code. This example is only provided to show the different coding methods.

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 two's 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. Table 2–3 lists the precedence rules from lowest to highest.

Table 2–3: Operator Precedence

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 2–4.

Table 2–4: 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 assignment (=) 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.
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	The type sdata is similar to the type 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.

Table 2–4: Data Types (cont.)

Type	Description
rdata and rconst	Any symbol defined while the `.rdata` or `.rconst` directives are in effect belongs to this category. The only difference between the types rdata and rconst is that the former is allowed to have dynamic relocations and the latter is not. (The types rdata and rconst are also similar to the type data but, unlike data, cannot 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`, `.rdata`, or `.rconst` 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` section (small bss section), 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 the `.bss` or `.sbss` sections 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, rconst, 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 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 symbol in a `.text` section, `.data` section, or `.bss` section, 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 symbol in a `.text` section, `.data` section, or `.bss` section, the 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 formats described in Table 2–5.

Table 2–5: 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.
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.

Table 2–5: Address Formats (cont.)

Format	Address Description
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.

3 Main Instruction Set

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, see the *Alpha Architecture Reference Manual*.

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

- Load and store instructions (Section 3.1)
- Arithmetic instructions (Section 3.2)
- Logical and shift instructions (Section 3.3)
- Relational instructions (Section 3.4)
- Move instructions (Section 3.5)
- Control instructions (Section 3.6)
- Byte-manipulation instructions (Section 3.7)
- Special-purpose instructions (Section 3.8)

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.
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.

Operand Specifier	Description
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 PALcode instruction.
s_reg, s_reg1, s_reg2	Source 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

Load and store instructions load immediate values and move data between memory and general registers. 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 divided into groups of instructions. The operands specified within a particular group apply to all of the instructions contained in that group.

Table 3–1: Load and Store Formats

Instruction	Mnemonic	Operands
Load Address	`lda`[a]	*d_reg, address*
Load Byte	`ldb`	
Load Byte Unsigned	`ldbu`	
Load Word	`ldw`	
Load Word Unsigned	`ldwu`	
Load Sign Extended Longword	`ldl`[a]	
Load Sign Extended Longword Locked	`ldl_l`[a]	
Load Quadword	`ldq`[a]	
Load Quadword Locked	`ldq_l`[a]	
Load Quadword Unaligned	`ldq_u`[a]	

Table 3–1: Load and Store Formats (cont.)

Instruction	Mnemonic	Operands
Unaligned Load Word	uldw	(See previous page)
Unaligned Load Word Unsigned	uldwu	
Unaligned Load Longword	uldl	
Unaligned Load Quadword	uldq	
Load Address High	ldah[a]	*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[a]	
Store Longword Conditional	stl_c[a]	
Store Quadword	stq[a]	
Store Quadword Conditional	stq_c[a]	
Store Quadword Unaligned	stq_u[a]	
Unaligned Store Word	ustw	
Unaligned Store Longword	ustl	
Unaligned Store Quadword	ustq	

[a] In addition to the normal operands that can be specified with this instruction, relocation operands can also be specified (see Section 2.6.4).

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 two's-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 — unless the setting of the .arch directive or the –arch flag on the cc or as command line causes the assembler to generate a single machine instruction in response to the ldbu 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. 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 — unless the setting of the .arch directive or the –arch flag on the cc or as command line causes the assembler to generate a single machine instruction in response to the ldwu 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.

Table 3–2: Load Instruction Descriptions (cont.)

Instruction	Description
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 an ldl_l instruction executes without generating an exception, 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. If a literal relocation type is specified in the ldq instruction, one machine instruction is generated and the symbol and offset are stored in the .lita section. Other relocation types generate a sequence of instructions and the symbol and offset are stored in that sequence.
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 generating an exception, 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: Load Instruction Descriptions (cont.)

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 two's-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 — unless the setting of the .arch directive or the –arch flag on the cc or as command line causes the assembler to generate a single machine instruction in response to the stb instruction.)

Table 3–3: Store Instruction Descriptions (cont.)

Instruction	Description
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 — unless the setting of the .arch directive or the −arch flag on the cc or as command line causes the assembler to generate a single machine instruction in response to the stw 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.
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

Arithmetic instructions perform arithmetic operations on values in registers. (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 divided into groups of instructions. The operands specified within a particular group apply to all of the instructions contained in that group.

Table 3–4: Arithmetic Instruction Formats

Instruction	Mnemonic	Operands
Clear	clr	*d_reg*
Absolute Value Longword	absl	*s_reg, d_reg* or *d_reg/s_reg* or *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 Byte	sextb	
Sign-Extension Longword	sextl	
Sign-Extension Word	sextw	

Table 3–4: Arithmetic Instruction Formats (cont.)

Instruction	Mnemonic	Operands
Add Longword (without overflow)	addl	*s_reg1, s_reg2, d_reg* or *d_reg/s_reg1, s_reg2* or *s_reg1, val_immed, d_reg* or *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 places 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 places 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 places 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 places 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 places 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 places the result in the destination register. An overflow exception is signaled if the value in the source register is -9223372036854775808.
Sign-Extension Byte (`sextb`)	Moves the least significant byte of the source register into the least significant byte of the destination register. Because the moved byte is a signed value, its sign bit is replicated to fill the other bytes in the destination register.
Sign-Extension Word (`sextw`)	Moves the two least significant bytes of the source register into the two least significant bytes of the destination register. Because the moved word is a signed value, its sign bit is replicated to fill the other bytes in the destination register.
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 places 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 places 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 places the result in the destination register. Overflow exceptions never occur.

Table 3–5: Arithmetic Instruction Descriptions (cont.)

Instruction	Description
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 places 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.
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 places either 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 places either 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 places either the 64-bit product of *s_reg1* and *s_reg2* or the immediate value in the destination register. Overflow is not reported.
Multiply Quadword (with overflow) (`mulqv`)	Computes the product of two signed 64-bit values. This instruction places either 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 places 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 places 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: Arithmetic Instruction Descriptions (cont.)

Instruction	Description
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 places 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 places the result in the destination register. If the true result's sign differs from the destination register's sign, an overflow exception is signaled.
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 places the high-order 64 bits of the 128-bit product in the destination register.
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 places the quotient in the destination register. The divl instruction rounds toward zero. If the divisor is zero, an error is signaled. Overflow is signaled when dividing -2147483648 by -1. A call_pal PAL_gentrap instruction may be issued for divide-by-zero and overflow exceptions.
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 places 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 the divlu instruction.)

Table 3–5: Arithmetic Instruction Descriptions (cont.)

Instruction	Description
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 places the quotient in the destination register. The divq instruction rounds toward zero. If the divisor is zero, an error is signaled. Overflow is signaled when dividing -9223372036854775808 by -1. A call_pal PAL_gentrap instruction may be issued for divide-by-zero and overflow exceptions. (The assembler uses temporary registers AT, t9, t10, t11, and t12 for the divq instruction.)
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 places 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 the divqu 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 places 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 the reml 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 places 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 the remlu instruction.)

Table 3–5: Arithmetic Instruction Descriptions (cont.)

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 places 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 the remq 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 places 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 the remqu instruction.)

3.3 Logical and Shift Instructions

Logical and shift instructions perform logical operations and shifts on values in registers.

Table 3–6 lists the mnemonics and operands for instructions that perform logical and shift operations. The table is divided into groups of instructions. The operands specified within a particular group apply to all of the instructions contained in that group.

Table 3–6: Logical and Shift Instruction Formats

Instruction	Mnemonic	Operands
Logical Complement — NOT	not	*s_reg, d_reg* or *d_reg/s_reg* or *val_immed, d_reg*

Table 3–6: Logical and Shift Instruction Formats (cont.)

Instruction	Mnemonic	Operands
Logical Product — AND	`and`	*s_reg1, s_reg2, d_reg* or *d_reg/s_reg1, s_reg2* or *s_reg1, val_immed, d_reg* or *d_reg/s_reg1, val_immed*
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 performs a complement operation on the contents of *s_reg1* and places the result in the destination register.
Logical Product — AND (`and`)	Computes the logical AND of two values. This instruction performs an AND operation between the contents of *s_reg1* and either the contents of *s_reg2* or the immediate value and then places the result in the destination register.
Logical Sum — OR (`bis`)	Computes the logical OR of two values. This instruction performs an OR operation between the contents of *s_reg1* and either the contents of *s_reg2* or the immediate value and then places the result in the destination register.
Logical Sum — OR (`or`)	Synonym for `bis`.
Logical Difference — XOR (`xor`)	Computes the XOR of two values. This instruction performs an XOR operation between the contents of *s_reg1* and either the contents of *s_reg2* or the immediate value and then places the result in the destination register.
Logical Product with Complement — ANDNOT (`bic`)	Computes the logical AND of two values. This instruction performs an AND operation between the contents of *s_reg1* and the one's complement of either the contents of *s_reg2* or the immediate value and then places the result in the destination register.

Table 3–7: Logical and Shift Instruction Descriptions (cont.)

Instruction	Description
Logical Product with Complement — ANDNOT (`andnot`)	Synonym for `bic`.
Logical Sum with Complement — ORNOT (`ornot`)	Computes the logical OR of two values. This instruction performs an OR operation between the contents of *s_reg1* and the one's complement of either the contents of *s_reg2* or the immediate value and then places the result in the destination register.
Logical Equivalence — XORNOT (`eqv`)	Computes the logical XOR of two values. This instruction performs an XOR operation between the contents of *s_reg1* and the one's complement of either the contents of *s_reg2* or the immediate value and then places 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. Register *s_reg1* contains the value to be shifted, and either the contents of *s_reg2* or the immediate value specifies the shift count. 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.
Shift Right Logical (`srl`)	Shifts the contents of a register to the right (toward the least significant bit) and inserts zeros in the vacated bit positions. Register *s_reg1* contains the value to be shifted, and either the contents of *s_reg2* or the immediate value specifies the shift count. 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 to the right (toward the least significant bit) and inserts the sign bit in the vacated bit position. Register *s_reg1* contains the value to be shifted, and either the contents of *s_reg2* or the immediate value specifies the shift count. 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

Relational instructions compare values in registers.

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.

Table 3–8: Relational Instruction Formats

Instruction	Mnemonic	Operands
Compare Signed Quadword Equal	cmpeq	*s_reg1, s_reg2, d_reg* or *d_reg/s_reg1, s_reg2* or *s_reg1, val_immed, d_reg* or *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 value in *s_reg1* equals the value in *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 value in *s_reg1* is less than the value in *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 value in *s_reg1* is less than or equal to the value in *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 value in *s_reg1* is less than either the value in *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 value in *s_reg1* is less than or equal to either the value in *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

Move instructions move data between registers.

Table 3–10 lists the mnemonics and operands for instructions that perform move operations. The table is divided into groups of instructions. The operands specified within a particular group apply to all of the instructions contained in that group.

Table 3–10: Move Instruction Formats

Instruction	Mnemonic	Operands
Move	mov	*s_reg, d_reg* or *val_immed, d_reg*
Move if Equal to Zero	cmoveq	*s_reg1, s_reg2, d_reg* or *d_reg/s_reg1, s_reg2* or *s_reg1, val_immed, d_reg* or *d_reg/s_reg1, val_immed*
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	

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.
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

Control instructions change the control flow of an assembly program. They affect the sequence in which instructions are executed by transferring control from one location in a program to another.

Table 3–12 lists the mnemonics and operands for instructions that perform control operations. The table is divided into groups of instructions. The operands specified within a particular group apply to all of the instructions contained in that group.

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

[a] In addition to the normal operands that can be specified with this instruction, relocation operands can also be specified (see Section 2.6.4).

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.
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.

Table 3–13: Control Instruction Descriptions (cont.)

Instruction	Description
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.
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`, and `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

Byte-manipulation instructions perform byte operations on values in registers.

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.

Table 3–14: Byte-Manipulation Instruction Formats

Instruction	Mnemonic	Operands
Compare Byte	`cmpbge`	*s_reg1, s_reg2, d_reg* or *d_reg/s_reg1, s_reg2* or *s_reg1, val_immed, d_reg* or *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.
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 places the result in the destination register. Bits 0-2 of register *s_reg2* or the immediate value specify the shift count.

Table 3–15: Byte-Manipulation Instruction Descriptions (cont.)

Instruction	Description
Insert Word Low (inswl)	Shifts the register *s_reg1* left by 0-7 bytes, inserts the word into a field of zeros, and then places 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 places 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 places 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 places 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 places 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 places 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.
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.

Table 3–15: Byte-Manipulation Instruction Descriptions (cont.)

Instruction	Description
Zero Bytes (`zap`)	Sets selected bytes of register *s_reg1* to zero and places 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 places 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

Special-purpose instructions perform miscellaneous tasks.

Table 3–16 lists the mnemonics and operands for instructions that perform special operations. The table is divided into groups of instructions. The operands specified within a particular group apply to all of the instructions contained in that group.

Table 3–16: Special-Purpose Instruction Formats

Instruction	Mnemonic	Operands
Call Privileged Architecture Library	`call_pal`	*palcode*
Architecture Mask	`amask`	*s_reg, d_reg* or *val_immed, d_reg*
Prefetch Data	`fetch`	*offset(b_reg)*
Prefetch Data, Modify Intent	`fetch_m`	
Read Process Cycle Counter	`rpcc`	*d_reg* or *d_reg, reg*
Implementation Version	`implver`	*d_reg*
No Operation	`nop`	*no_operands*
Universal No Operation	`unop`	
Trap Barrier	`trapb`	
Exception Barrier	`excb`	
Memory Barrier	`mb`	
Write Memory Barrier	`wmb`	
Count Leading Zero	`ctlz`	*s_reg, d_reg*

Table 3–16: Special-Purpose Instruction Formats (cont.)

Instruction	Mnemonic	Operands
Count Population	ctpop	(See previous page)
Count Trailing Zero	cttz	

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.
Architecture Mask (amask)	The value of the contents of s_reg or the immediate value represent a mask of architectural extensions that are being requested. Bits are cleared if they correspond to architectural extensions that are present, and the result is placed in the destination register.
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.
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. If *reg* is specified, the rpcc instruction is not issued until all previous instructions that generate a result in *reg* are completed. If R31 is specified as the *reg* operand, the *reg* operand is ignored and the rpcc instruction does not wait for any preceding computation.
Implementation Version (implver)	A small integer is placed in the destination register. This integer specifies the major implementation version of the processor on which it is executed. This information can be used to make code-scheduling or tuning decisions. The returned small integer can have the values 0, 1, or 2. 0 indicates EV4, EV45, LCA, and LCA-45 Alpha chips (that is, 21064, 21064A, 21066, 21068, and 21066A, respectively); 1 indicates an EV5 Alpha chip (21164); and 2 indicates an EV6 Alpha chip (21264).
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 are completed, 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 additional information on memory barriers.

Table 3–17: Special-Purpose Instruction Descriptions (cont.)

Instruction	Description
Write Memory Barrier (`wmb`)	Guarantees that all previous store instructions access memory before any store instructions issued after the `wmb` instruction.
Count Leading Zeros (`ctlz`)	Counts the number of leading zeros in *s_reg*, starting at the most significant bit position, and writes that count to *d_reg*.
Count Population (`ctpop`)	Counts the number of ones in *s_reg* and writes the count to *d_reg*.
Count Trailing Zeros (`cttz`)	Counts the number of trailing zeros in *s_reg*, starting at the least significant bit position, and writes the count to *d_reg*.

4

Floating-Point Instruction Set

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, see the *Alpha Architecture Reference Manual*.

This chapter addresses the following topics:

- Background information on floating-point operations — data types, the control register, exceptions, rounding modes, and qualifiers (Section 4.1)
- The instructions in the assembler's floating-point instruction set, which consists of the following classes:
  - Load and store instructions (Section 4.2)
  - Arithmetic instructions (Section 4.3)
  - Relational instructions (Section 4.4)
  - Move instructions (Section 4.5)
  - Control instructions (Section 4.6)
  - Special-purpose instructions (Section 4.7)

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.
2d_reg/ s_reg	One floating-point register that is used as both a destination register and a source register.
label	A label that identifies a location in a program.

Operand Specifier	Description
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 +INF or –INF.
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 to (=), 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

Topics addressed in the following sections include:

- 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

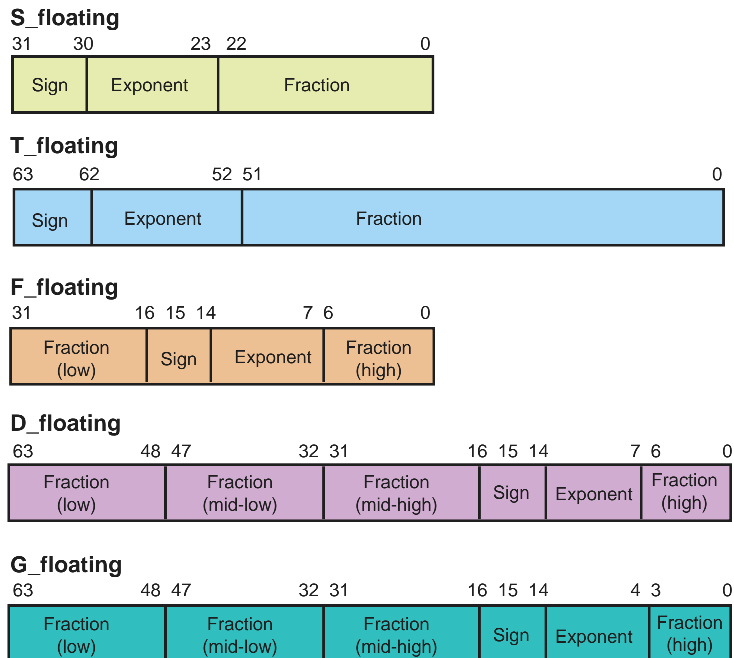
Floating-point instructions operate on the following data types:

- 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

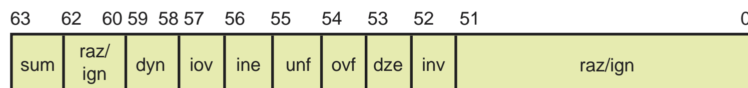


4.1.2 Floating-Point Control Register

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 — see Section 4.1.5 for information on instruction qualifiers) 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



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 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.
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 noninfinite. (The `cmpt`*`xx`* instruction 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 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 than the smallest finite number of the destination format. This trap can 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 can 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 that the destination format can represent. This trap can 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 noninfinite 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 being mapped to the one whose fraction ends in 0 (sometimes referred to as 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. It can be used with any of the IEEE rounding modes. (Dynamic rounding mode is described in Section 4.1.2.)

Alpha 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)

Trapping Mode	Qualifier
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)
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 values in the Number column are 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

Table 4–1: Qualifier Combinations for Floating-Point Instructions (cont.)

Number	Qualifiers
8	c
9	c, m, d, sui, suic, suim, suid

4.2 Floating-Point Load and Store Instructions

Floating-point load and store instructions load values and 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 divided into groups of functionally related instructions. The operands specified within a particular group apply to all of the instructions contained in that group.

Table 4–2: Load and Store Instruction Formats

Instruction	Mnemonic	Operands
Load F_floating	ldf[a]	*d_reg, address*
Load G_floating (Load D_floating)	ldg[a]	
Load S_floating (Load Longword)	lds[a]	
Load T_floating (Load Quadword)	ldt[a]	
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[a]	*s_reg, address*
Store G_floating (Store D_floating)	stg[a]	
Store S_floating (Store Longword)	sts[a]	
Store T_floating (Store Quadword)	stt[a]	

[a] In addition to the normal operands that can be specified with this instruction, relocation operands can also be specified (see Section 2.6.4).

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

The load and store instructions are grouped by function. See 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

Floating-point arithmetic 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 divided into groups of functionally related instructions. The operands specified within a particular group apply to all of the instructions contained in that group.

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* or *d_reg/s_reg*
Floating Negate	fneg	—	
Negate F_floating	negf	3	
Negate G_floating	negg	3	
Negate S_floating	negs	4	
Negate T_floating	negt	4	

Table 4–4: Arithmetic Instruction Formats (cont.)

Instruction	Mnemonic	Qualifiers	Operands
Add F_floating	addf	1	*s_reg1, s_reg2, d_reg* or *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 G_floating	subg	1	
Subtract S_floating	subs	2	
Subtract T_floating	subt	2	
Convert Quadword to Longword	cvtql	5	*s_reg, d_reg* or *d_reg/s_reg*
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. See Table 4–4 for the instruction names.

Table 4–5: Arithmetic Instruction Descriptions

Instruction	Description
Clear Instruction (`fclr`)	Clears the destination register.
Absolute Value Instruction (`fabs`)	Computes the absolute value of the contents of the source register and puts the floating-point result in the destination register.
Negate Instructions (`fneg`, `negf`, `negg`, `negs`, `negt`)	Computes the negative value of the contents of *s_reg* or *d_reg* and puts the specified precision floating-point result in *d_reg*.
Add Instructions (`addf`, `addg`, `adds`, `addt`)	Adds the contents of *s_reg* or *d_reg* to the contents of *s_reg2* and puts 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 –INF. For that rounding mode, the sum has a negative sign.
Divide Instructions (`divf`, `divg`, `divs`, `divt`)	Computes 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 results 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`)	Multiplies the contents of *s_reg1* or *d_reg* with the contents of *s_reg2* and puts the result in *d_reg*.
Subtract Instructions (`subf`, `subg`, `subs`, `subt`)	Subtracts the contents of *s_reg2* from the contents of *s_reg1* or *d_reg* and puts 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 –INF. For that rounding mode, the sum has a negative sign.
Conversion Between Integer Formats Instructions (`cvtql`, `cvtlq`)	Converts the integer contents of *s_reg* to the specified integer format and puts 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`)	Converts the floating-point contents of *s_reg* to the specified integer format and puts 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.

Table 4–5: Arithmetic Instruction Descriptions (cont.)

Instruction	Description
Conversion from Integer to Floating-Point Format Instructions (cvtqf, cvtqg, cvtqs, cvtqt)	Converts the integer contents of *s_reg* to the specified floating-point format and puts the result in *d_reg*.
Conversion Between Floating-Point Formats Instructions (cvtdg, cvtgd, cvtgf, cvtts, cvtst)	Converts the contents of *s_reg* to the specified precision, round according to the rounding mode, and puts 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

Floating-point 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.

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* or *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. See 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.
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

Floating-point move 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 divided into groups of functionally related instructions. The operands specified within a particular group apply to all of the instructions contained in that group.

Table 4–8: Move Instruction Formats

Instruction	Mnemonic	Operands
Floating Move	`fmov`	*s_reg, d_reg*
Copy Sign	`cpys`	*s_reg1, s_reg2, d_reg* or *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. See Table 4–8 for the instruction names.

Table 4–9: Move Instruction Descriptions

Instruction	Description
Move Instruction (fmov)	Moves the contents of *s_reg* to *d_reg*.
Copy Sign Instruction (cpys)	Fetches the sign bit of *s_reg1* or *d_reg*, combines it with the exponent and fraction of *s_reg2*, and copies the result to *d_reg*.
Copy Sign Negate Instruction (cpysn)	Fetches the sign bit of *s_reg1* or *d_reg*, complements it, combines it with the exponent and fraction of *s_reg2*, and copies the result to *d_reg*.
Copy Sign and Exponent Instruction (cpyse)	Fetches the sign and exponent of *s_reg1* or *d_reg*, combines them with the fraction of *s_reg2*, and copies the result to *d_reg*.
Move If Instructions (fcmoveq, fcmovne, fcmovlt, fcmovle, fcmovgt, fcmovge)	Compares the contents of *s_reg1* or *d_reg* against zero. If the specified condition is true, the contents of *s_reg2* are copied to *d_reg*; otherwise, *d_reg* is unchanged.

4.6 Floating-Point Control Instructions

Floating-point control 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 specified operands apply to all of the instructions listed in the table.

Table 4–10: Control Instruction Formats

Instruction	Mnemonic	Operands
Branch Equal to Zero	fbge	*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. See 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

Floating-point special-purpose 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`	(none)

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`)	Copies the value in the specified source register to the floating-point control register (FPCR).
Move from FPCR Instruction (`mt_fpcr`)	Copies 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.

5

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	`.err`
	`.file`
	`.lab`
	`.loc`
	`.option`

Location Control Directives	`.align`
	`.data`
	`.lit4`
	`.lit8`
	`.rconst`
	`.rdata`
	`.sdata`
	`.space`
	`.text`
	`.tlsdata`

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

Routine Entry Point Definition Directives	`.aent` `.ent`

Table 5–1: Summary of Assembler Directives (cont.)

Category	Directives

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

Repeat Block Directives	`.endr` `.repeat`
Assembler Option Directive	`.set`

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

Version Control Directive	`.ident` `.verstamp`
Scheduling and Architecture Subset Directives	`.arch` `.tune`

The following list contains descriptions of the assembly directives (in alphabetical order):

.aent *name*

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.

.align *expression*

Sets low-order bits in the location counter to zero. The value of *expression* establishes the number of bits to be set to zero. The maximum value for *expression* is 16 (which produces 64K alignment).

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

Normally, the `.word`, `.long`, `.quad`, `.float`, `.double`, `.extended`, `.d_floating`, `.f_floating`, `.g_floating`, `.s_floating`, `.t_floating`, and `.x_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
```

This is equivalent to:

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

.arch *model*

Specifies the version of the Alpha architecture that the assembler is to generate instructions for. The valid values for *model* are identical to those you can specify with the `-arch` flag on the `cc` command line. See `cc`(1) for details.

.ascii *string* [, *string*] ...

Assembles each *string* from the list into successive locations. The `.ascii` directive does not pad the string with null characters. You must put double quotation marks (") around each string. You can optionally use the backslash escape characters. For a list of the backslash characters, see Section 2.4.3.

.asciiz *string* [, *string*] ...

Assembles each *string* in the list into successive locations and adds a null character. You can optionally use the backslash escape characters. For a list of the backslash characters, see Section 2.4.3.

.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 for the `.byte` directive can optionally have the following 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 , expression1*[,*expression2*]

Unless defined elsewhere, *name* becomes a global common symbol at the head of a block of at least *expression1* 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. The *expression2* operand has the same effect on alignment as the operand for the `.align` directive.

.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 for the `.d_floating` directive can optionally have the following 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.

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

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

Synonym for `.t_floating`.

.edata `0`
.edata `1` *`lang-handler relocatable-expression`*
.edata `2` *`lang-handler constant-expression`*

Marks data related to exception handling.

If the flag is `0`, the assembler adds all subsequent data to the `.xdata` section.

If the flag is `1` or `2`, the assembler creates a function table entry for the next `.ent` directive. The function table entry contains the language-specific handler (*`lang-handler`*) and data (*`relocatable-expression`* or *`constant-expression`*).

.eflag *`flags`*

Encodes exception-related flags to be stored in the `PDSC_RPD_FLAGS` field of the procedure's run-time procedure descriptor. See the *Calling Standard for Alpha Systems* 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.

.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; the assembler ignores it.

.err

*For use only by compilers.*This directive causes the assembler to signal an error. Any compiler frontend that detects an error condition puts this directive in the input stream. When the assembler encounters a `.err` directive, it issues an error message and ceases to assemble the source file. This prevents the assembler from continuing to process a program that is incorrect.

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

Synonym for `.x_floating`.

.extern [(THREADS)] *name* [*number*]

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 and *number* is the expected size of the external object. If the THREADS argument is specified, the symbol is treated as a tls (thread local storage) 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 for the `.f_floating` directive can optionally have the following 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.

The `.f_floating` directive automatically aligns its data and preceding labels on 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 from which the assembly instructions that follow originated. 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-reg frame-size return_pc-reg* [*local_offset*]

Describes a stack frame. The first register is the frame register, and*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 for the `.g_floating` directive can optionally have the following 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.

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

.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 for the `.gprel32` directive can optionally have the following 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 (*addressVal*) and repetition count (*addressRep*) must be absolute.

The `.gprel32` directive automatically aligns its data and preceding labels on a longword boundary. You can disable this feature with the `.align 0` directive.

.ident *string*

Allows the specification of a string that the assembler stores in the `.o` file created during assembly. This string can be searched for in a `.o` file or in an executable using the `what(1)` command.

.lab *label_name*

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

.lcomm *name*, *expression1*[,*expression2*]

Gives the named symbol (*name*) a data type of `bss`. The assembler allocates the named symbol to the bss area, and *expression1* defines the named symbol's length. If a `.globl` directive also specifies the name, the assembler allocates the named symbol as an external symbol. The *expression2* operand has the same effect on alignment as the operand for the `.align` directive. If *expression2* is not specified, the alignment defaults to quadword alignment.

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.

.lit4

Allows 4-byte constants to be generated and placed in the `lit4` section. This directive is only valid for `.long` (with nonrelocatable expressions), `.f_floating`, `.float`, and `.s_floating`.

.lit8

Allows 8-byte constants to be generated and placed in the `lit8` section. This directive is only valid for `.quad` (with nonrelocatable expressions), `.d_floating`, `.g_floating`, `.double`, and `.t_floating`.

.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.

.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 for the `.long` directive can optionally have the following 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.

The `.long` directive automatically aligns its data and preceding labels on 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. The least significant bit corresponds to register `$0`. 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`.

.option *`options`*

For use only by compilers. Instructs the assembler to replace an optimization level that was specified on the command with the one specified in the options argument. Valid entries for this new optimization level are `O`, `O1` – `O4`.

.prologue *flag*

Marks the end of the prologue section of a procedure.

A *flag* of 0 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 1 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 1 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 for the `.quad` directive can optionally have the following 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.

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

.rconst

Instructs the assembler to add subsequent data into the `.rconst` section. (This is the same as the `.rdata` directive except that the entries cannot be relocatable.)

.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. Also note that nesting `.repeat` directives is not allowed.

.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`. Only one option can be specified by a single `.set` directive. The effects of the options are as follows:

- 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-language 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 following 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.

The `.s_floating` directive automatically aligns its data and preceding labels on 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 it. 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.)

.tlscomm *name*,*expression*

The *name* operand becomes a global tls common symbol at the head of a block of *expression* bytes of storage. This directive is analogous to the `.comm` directive.

.tlsdata

Directs the assembler to add all subsequent data to the `.tlsdata` section. This directive is analogous to the `.data` directive.

.tlslcomm *name*,*expression*

The *name* operand becomes a symbol of type `tlsbss`. The assembler allocates the symbol to the tlsbss section and the expression defines the named symbol's length. If a `.globl` directive also specifies the symbol name, the assembler allocates the named symbol as an external symbol.

Unlike non-tls symbols, thread local storage's `bss` data is allocated in only one area. There is no sbss area for tls symbols. This directive is analogous to the `.lcomm` directive.

```
.tlslcomm  b 8        /* TlsBss      stStatic */
.lcomm     B 8        /* SBss        stStatic */

.globl     c          /* TlsBss      stGlobal */
.tlslcomm  c 8
.globl     C          /* SBss        stGlobal */
.lcomm     C 8
```

.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 for the `.t_floating` directive can optionally have the following 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.

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

.tune *option*

Selects processor-specific instruction tuning for various implementations of the Alpha architecture. Regardless of the setting of the `.arch` directive, the generated code will run correctly on all implementations of the Alpha architecture. The valid values for *option* are identical to those you can specify with the `–arch` flag on the `cc` command line. See `cc`(1) for details.

.verstamp *major minor*

Specifies the major and minor version numbers; for example, version 0.15 would be `.verstamp 0 15`.

.weakext *name1* [,*name2*]

Sets *name1* to be a weak symbol during linking. If *name2* is specified, *name1* is created as a weak symbol with the same value as *name2*. 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 for the `.word` directive can optionally have the following 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.

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

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

Initializes memory to quad-precision (128-bit) IEEE floating-point numbers. The values of the expressions must be absolute.

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

expressionVal [: *expressionRep*]

The *expressionVal* is a 128-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.

The `.x_floating` directive automatically aligns its data and preceding labels on an octaword boundary. You can disable this feature with the `.align 0` directive.

6 Programming Considerations

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

The chapter addresses the following topics:

- Why your assembly programs should use the calling conventions observed by the C compiler (Section 6.1)
- An overview of the composition of executable programs (Section 6.2)
- The use of registers, section and location counters, and stack frames (Section 6.3)
- A technique for coding an interface between an assembly language procedure and a procedure written in a high-level language (Section 6.4)
- The default memory-allocation scheme used by the Alpha system (Section 6.5)

This chapter does not address coding issues related to performance or optimization. See Appendix A of the *Alpha Architecture Reference Manual* 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 Tru64 UNIX 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 64 KB can be addressed by a single memory-format instruction, the GOT is segmented into one or more sections of 64 KB or less.

In addition to providing efficient access to the GOT, the `gp` register is also used to access global data within ±2 GB 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 64 KB (that is, 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 (Section 6.3.1)
- Control of section and location counters with directives (Section 6.3.2)
- Stack frame requirements on entering and exiting a procedure (Section 6.3.3)

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 double-precision (64 bit) value. See 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.
`$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.

Table 6–1: Integer Registers (cont.)

Register Name	Software Name (from regdef.h)	Use
`$29` or `$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 (`$f0` has the real part and `$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`, `.rconst`, `.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.

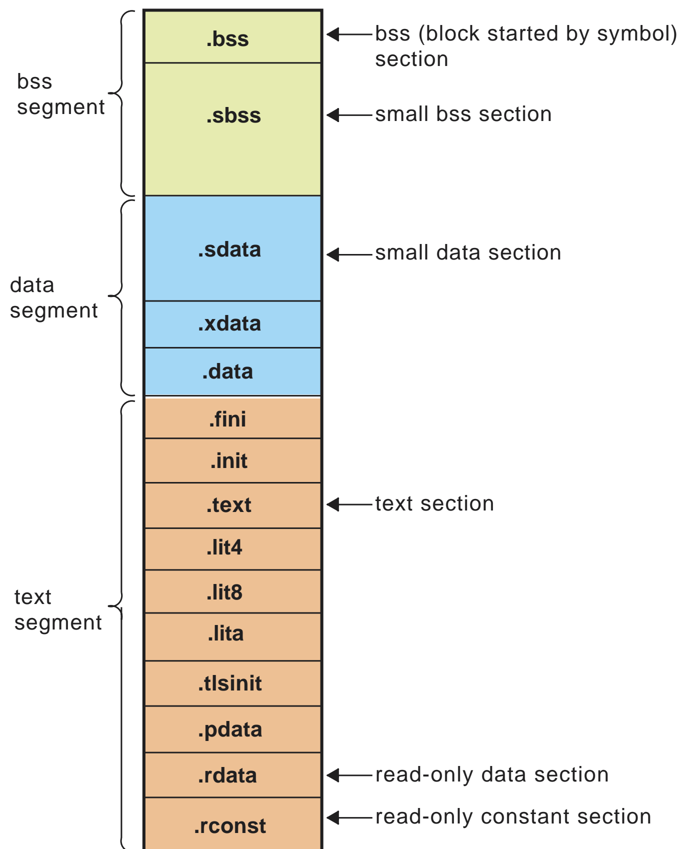
The bss (block started by symbol) section holds data items (usually variables) that are initialized to zero. If a `.lcomm` directive defines a variable, the assembler assigns that variable to either the `.bss` section or the `.sbss` (small bss) section, depending on the variable's size.

The default size for variables in the `.sbss` section 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 the `.sbss` section. 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: Sections and Location Counters for Nonshared Object Files



See the *Symbol Table / Object File Specification* manual for more information on section data. (This manual is available as an HTML or PDF document on the Tru64 UNIX Version 5.1 Documentation CD-ROM; it is not available in hardcopy.)

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 the code you must provide at the beginning of a procedure, step 7 describes how to pass parameters, and steps 8 through 12 describe the code you must provide at the end of a procedure:

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 area does not include space for the first six arguments because they are always passed in registers.

Note

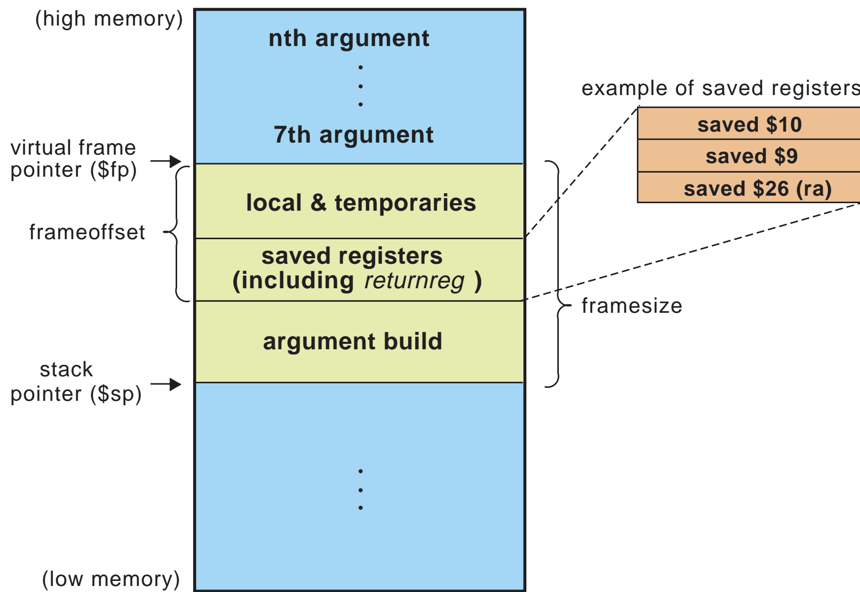
Once you have modified register `$sp`, you should not modify it again in the remainder of the procedure.

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 does not have a register allocated for it. It consists of the *framereg* (`$sp`, in most cases) added to the *framesize* (see step 3). Figure 6–2 shows 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`). The 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 for which you allocated space 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 settings 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 register (`$26`) is the first register to be saved; it must be saved at *framesize+frameoffset*`+0($sp)` for exception handling. 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 shows the order in which the registers in the preceding example would be saved.)

Then, save any floating-point registers for which you allocated space 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 single- or double-precision floating-point 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 zero.

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 used to write the procedures that interact with your program.

   General registers `$16` to `$21` and floating-point registers `$f16` to `$f21` are used for passing the first six arguments. All nonfloating-point arguments in the first six arguments are passed in general registers. All floating-point arguments in the first six arguments 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)
```

   (See 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 Coding Examples

The examples in this section show procedures written in C and the equivalent procedures written in assembly language.

Example 6–1 shows a nonleaf procedure. Note that it creates a stack frame 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. Note 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     $17, $17, $0
        cmplt   $0, $16, $1
        cmovne  $1, $16, $0
        ret     $31, ($26), 1
        .end    leaf
```

Example 6–3 shows a leaf procedure that requires stack space for local variables. Note 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: Leaf Procedure with Stack Space for Local Variables (cont.)

```
 #    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

The following steps show an approach to use in writing an assembly language procedure that calls `atof`(3), a procedure written in C that converts ASCII characters to numbers:

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`. The comments in the file 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

The following steps show an approach to use in writing an assembly language procedure that can be called by a procedure written in a high-level language:

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`. The comments in the file show how the parameters are passed, the execution of the call, and how the returned values are retrieved:

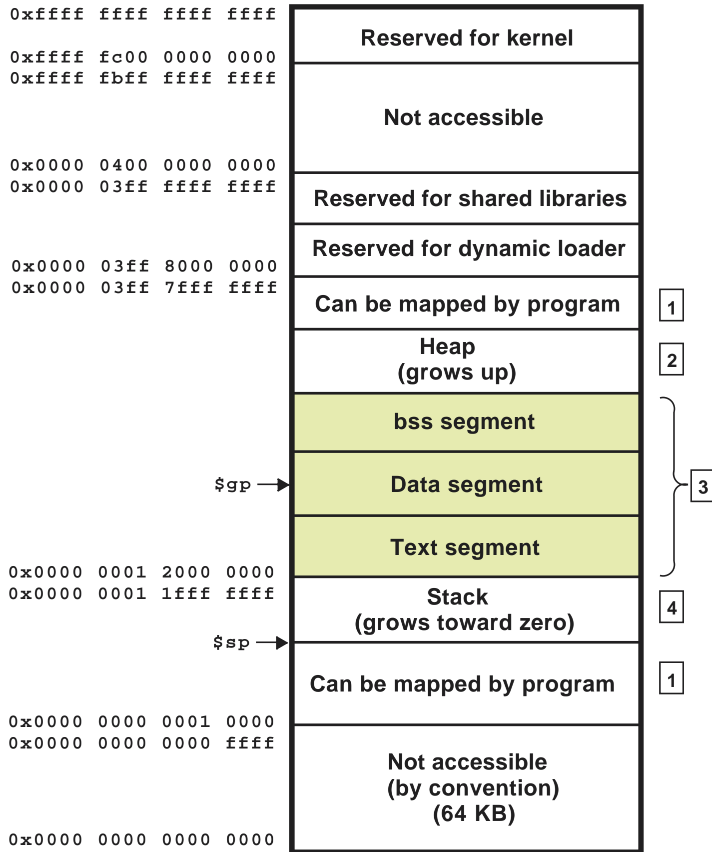
   ```
           .comm   global_r 4
           .comm   global_i 4
           .comm   global_s 10
           .comm   global_b 4
           .text
           .globl  callee
    #   10   {
           .ent    callee 2
   callee:
           ldgp    $gp, 0($27)
           .frame  $sp, 0, $26, 0
           .prologue       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 system gives every process two storage areas that can grow without bounds. A process exceeds virtual storage only when the sum of the two areas exceeds virtual storage space. By default, the linker and assembler use the scheme shown in Figure 6–3.

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



1. This area is not allocated until a user requests it. (The same behavior is observed in System V shared memory regions.)
2. The heap is reserved for `sbrk` and `brk` system calls, and it is not always present.
3. See the *Symbol Table / Object File Specification* manual for details on the sections contained within the bss, data, and text segments. (This manual is available as an HTML or PDF document on the Tru64 UNIX Version 5.1 Documentation CD-ROM; it is not available in hardcopy.)
4. The stack is used for local data in C programs.

A

Instruction Summaries

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.

Operand Specifier	Description
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.

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`[a]	*d_reg, address*
Load Byte	`ldb`	
Load Byte Unsigned	`ldbu`	
Load Word	`ldw`	
Load Word Unsigned	`ldwu`	
Load Sign Extended Longword	`ldl`[a]	
Load Sign Extended Longword Locked	`ldl_l`[a]	
Load Quadword	`ldq`[a]	
Load Quadword Locked	`ldq_l`[a]	
Load Quadword Unaligned	`ldq_u`[a]	
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`[a]	
Store Longword Conditional	`stl_c`[a]	
Store Quadword	`stq`[a]	

Table A–1: Main Instruction Set Summary (cont.)

Instruction	Mnemonic	Operands
Store Quadword Conditional	stq_c[a]	(See previous page)
Store Quadword Unaligned	stq_u[a]	
Store Unaligned Word	ustw	
Store Unaligned Longword	ustl	
Store Unaligned Quadword	ustq	
Load Address High	ldah[a]	*d_reg,offset* (*b_reg*)
Load Global Pointer	ldgp	
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* or *label*
Branch to Subroutine	bsr	
Jump	jmp[a]	*d_reg,(s_reg),jhint* or *d_reg,(s_reg)* or *(s_reg),jhint* or *(s_reg)* or *d_reg,address* or *address*
Jump to Subroutine	jsr[a]	
Return from Subroutine	ret	*d_reg,(s_reg),rhint* or *d_reg,(s_reg)* or *d_reg,rhint* or *d_reg* or *(s_reg),rhint* or *(s_reg)* or *rhint* or *no_operands*
Jump to Subroutine Return	jsr_coroutine[a]	
Architecture Mask	amask	*s_reg,d_reg* or *val_immed,d_reg*
Clear	clr	*d_reg*
Implementation Version	implver	

Table A–1: Main Instruction Set Summary (cont.)

Instruction	Mnemonic	Operands
Absolute Value Longword	absl	*s_reg,d_reg* or *d_reg/s_reg* or *val_immed,d_reg*
Absolute Value Quadword	absq	
Move	mov	
Negate Longword (without overflow)	negl	
Negate Longword (with overflow)	neglv	
Negate Quadword (without overflow)	negq	
Negate Quadword (with overflow)	negqv	
Logical Complement (NOT)	not	
Sign-Extension Byte	sextb	
Sign-Extension Longword	sextl	
Sign-Extension Word	sextw	
Add Longword (without overflow)	addl	*s_reg1,s_reg2,d_reg* or *d_reg/s_reg1,s_reg2* or *s_reg1,val_immed,d_reg* or *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	
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	

Table A–1: Main Instruction Set Summary (cont.)

Instruction	Mnemonic	Operands
Subtract Quadword (without overflow)	subq	(See previous page)
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	
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	
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	
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	

Table A–1: Main Instruction Set Summary (cont.)

Instruction	Mnemonic	Operands
Move if Low Bit Set	cmovlbs	(See previous page)
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	
Call Privileged Architecture Library	call_pal	*palcode*
Prefetch Data	fetch	*offset* (*b_reg*)
Prefetch Data, Modify Intent	fetch_m	
Read Process Cycle Counter	rpcc	*d_reg* or *d_reg, reg*

Table A–1: Main Instruction Set Summary (cont.)

Instruction	Mnemonic	Operands
No Operation	nop	*no_operands*
Universal No Operation	unop	
Trap Barrier	trapb	
Exception Barrier	excb	
Memory Barrier	mb	
Write Memory Barrier	wmb	
Count Leading Zeros	ctlz	*s_reg, d_reg*
Count Population	ctpop	
Count Trailing Zeros	cttz	

[a] In addition to the normal operands that can be specified with this instruction, relocation operands can also be specified (see Section 2.6.4).

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. (The notes also identify the instructions on which relocation operands can be specified.)

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[a]	*d_reg,address*
Load G_Floating (Load D_Floating)	ldg[a]	
Load S_Floating (Load Longword)	lds[a]	
Load T_Floating (Load Quadword)	ldt[a]	
Store F_Floating	stf[a]	*s_reg,address*
Store G_Floating (Store D_Floating)	stg[a]	
Store S_Floating (Store Longword)	sts[a]	
Store T_Floating (Store Quadword)	stt[a]	
Load Immediate F_Floating	ldif	*d_reg, val_expr*
Load Immediate D_Floating	ldid	

Table A–2: Floating-Point Instruction Set Summary (cont.)

Instruction	Mnemonic	Operands
Load Immediate G_Floating	ldig	
Load Immediate S_Floating	ldis	
Load Immediate T_Floating	ldit	(See previous page)
Branch Equal to Zero	fbeq	*s_reg, label* or *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	
Floating Clear	fclr	*d_reg*
Floating Move	fmov	*s_reg, d_reg* or *d_reg/s_reg*
Floating Negate	fneg	
Floating Absolute Value	fabs	
Negate F_Floating	negf[b]	
Negate G_Floating	negg[b]	
Negate S_Floating	negs[c]	
Negate T_Floating	negt[c]	
Copy Sign	cpys	*s_reg1, s_reg2, d_reg* or *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	
Add F_Floating	addf[d]	
Add G_Floating	addg[d]	
Add S_Floating	adds[e]	
Add T_Floating	addt[e]	
Compare G_Floating Equal	cmpgeq[b]	

Table A–2: Floating-Point Instruction Set Summary (cont.)

Instruction	Mnemonic	Operands
Compare G_Floating Less Than	cmpglt[b]	
Compare G_Floating Less Than or Equal	cmpgle[b]	
Compare T_Floating Equal	cmpteq[c]	(See previous page)
Compare T_Floating Less Than	cmptlt[c]	
Compare T_Floating Unordered	cmptun[c]	
Compare T_Floating Less Than or Equal	cmptle[c]	
Divide F_Floating	divf[d]	
Divide G_Floating	divg[d]	
Divide S_Floating	divs[e]	
Divide T_Floating	divt[e]	
Multiply F_Floating	mulf[d]	
Multiply G_Floating	mulg[d]	
Multiply S_Floating	muls[e]	
Multiply T_Floating	mult[e]	
Subtract F_Floating	subf[d]	
Subtract G_Floating	subg[d]	
Subtract S_Floating	subs[e]	
Subtract T_Floating	subt[e]	
Convert Quadword to Longword	cvtql[f]	*s_reg, d_reg* or *d_reg/s_reg*
Convert Longword to Quadword	cvtlq	
Convert G_Floating to Quadword	cvtgq[g]	
Convert T_Floating to Quadword	cvttq[h]	
Convert Quadword to F_Floating	cvtqf[i]	
Convert Quadword to G_Floating	cvtqg[i]	
Convert Quadword to S_Floating	cvtqs[j]	
Convert Quadword to T_Floating	cvtqt[j]	
Convert D_Floating to G_Floating	cvtdg[d]	
Convert G_Floating to D_Floating	cvtgd[d]	
Convert G_Floating to F_Floating	cvtgf[d]	
Convert T_Floating to S_Floating	cvtts[e]	
Convert S_Floating to T_Floating	cvtst[b]	
Move From FP Control Register	mf_fpcr	*d_reg*

Table A–2: Floating-Point Instruction Set Summary (cont.)

Instruction	Mnemonic	Operands
Move To FP Control Register	mt_fpcr	*s_reg*
Floating No Operation	fnop	*no_operands*

[a] In addition to the normal operands that can be specified with this instruction, relocation operands can also be specified (see Section 2.6.4).
[b] s
[c] su
[d] c, u, uc, s, sc, su, suc
[e] c, m, d, u, uc, um, ud, su, suc, sum, sud, sui, suic, suim, suid
[f] sv, v
[g] c, v, vc, s, sc, sv, svc
[h] c, v, vc, sv, svc, svi, svic, d, vd, svd, svid
[i] c
[j] c, m, d, sui, suic, suim, suid

See the text immediately preceding Table A–2 for a description of the table notes.

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

B

32-Bit Considerations

The Alpha architecture is a quadword (64-bit) architecture, with limited backward compatibility for longword (32-bit) operations. The Alpha architecture's 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.

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, use the following instruction sequence:

```
sll  $rx, xx, $ry    # noncanonical result
addl $ry, 0, $ry     # sign-extend bit-31
```

To perform a logical shift right, use the following instruction sequence:

```
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 nonzero.

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, use the following instruction sequence:

```
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, use the following instruction sequence:

```
addl $rx, 0, $ry     # sign-extend bit-31
```

C

Basic Machine Definition

The assembly language instructions described in this manual 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 instructions 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`[a]
`ldw`	`AT,t9`
`ldwu`	`AT,t9`[a]
`stb`	`AT,t9,t10`[a]
`stw`	`AT,t9,t10`[a]
`ustw`	`AT,t9,t10,t11,t12`
`ustl`	`AT,t9,t10,t11,t12`
`ustq`	`AT,t9,t10,t11,t12`
`uldw`	`AT,t9,t10`

Instruction	Registers Used
uldwu	AT,t9,t10
uldl	AT,t9,t10
uldq	AT,t9,t10
divl	AT,t9,t10,t11,t12
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

[a] Use of registers depends on the setting of the .arch directive or the –arch flag on the cc command line.

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

On most processors that implement the Alpha architecture, loading and storing unaligned data or data less than 32 bits is done with multiple machine-code instructions. Except on EV56 Alpha processors, the following assembler instructions generate multiple machine-code 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.

On EV56 Alpha processors, the following instructions from the preceding list generate a single instruction:

- Load Byte Unsigned (`ldbu`)
- Load Word Unsigned (`ldwu`)

- Store Byte (stb)
- Store Word (stw)

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. Such assembler instructions are sometimes referred to as pseudoinstructions. 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
sextl	$rx,$ry	addl	$rx,0,$ry
unop		ldq_u	$31,0($sp)
xornot	$rx,$ry,$rz	eqv	$rx,$ry,$rz

D

PALcode Instruction Summaries

This appendix summarizes the Privileged Architecture Library (PALcode) instructions that are required to support an Alpha 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.
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.

Table D–2: Privileged PALcode Instructions (cont.)

Symbolic Name	Number	Operation and Description
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, 3–8, 3–10
absq instruction, 3–8, 3–10
addf instruction, 4–11, 4–12
addg instruction, 4–11, 4–12
addl instruction, 3–9, 3–10
addlv instruction, 3–9, 3–10
addq instruction, 3–9, 3–10
addqv instruction, 3–9, 3–11
address formats, 2–12
addresses
  special handling, C–2
adds instruction, 4–11, 4–12
addt instruction, 4–11, 4–12
.aent directive, 5–2
.align directive, 5–3
amask instruction, 3–24, 3–25
and instruction, 3–15
andnot instruction, 3–15, 3–16
.arch directive, 5–3
arithmetic instructions
  floating-point instruction set, 4–10
  main instruction set, 3–8
.ascii directive, 5–3
.asciiz directive, 5–3
assembler directives, 5–1

B

backslash escape character, 2–3
beq instruction, 3–19
bge instruction, 3–19, 3–20
bgt instruction, 3–19, 3–20
bic instruction, 3–15
big-endian
  byte ordering, 1–2
bis instruction, 3–15
blbc instruction, 3–19, 3–20
blbs instruction, 3–19, 3–20
ble instruction, 3–19, 3–20
blt instruction, 3–19
bne instruction, 3–19
br instruction, 3–19, 3–20
bsr instruction, 3–19, 3–20
.bss section, 6–4
.byte directive, 5–4
byte ordering
  big-endian, 1–2
  little-endian, 1–2
byte-manipulation instructions
  main instruction set, 3–21

C

C programs
  -S compilation option, 6–14
call_pal instruction, 3–24, 3–25
calling conventions, 6–1
chopped rounding (IEEE), 4–6
chopped rounding (VAX), 4–6
clr instruction, 3–8
cmoveq instruction, 3–18
cmovge instruction, 3–18
cmovgt instruction, 3–18
cmovlbc instruction, 3–18
cmovlbs instruction, 3–18
cmovle instruction, 3–18
cmovlt instruction, 3–18
cmovne instruction, 3–18
cmpbge instruction, 3–21, 3–22

cmpeq instruction, 3–17
cmpgeq instruction, 4–13, 4–14
cmpgle instruction, 4–13, 4–14
cmpglt instruction, 4–13, 4–14
cmple instruction, 3–17
cmplt instruction, 3–17
cmpteq instruction, 4–13, 4–14
cmptle instruction, 4–13, 4–14
cmptlt instruction, 4–13, 4–14
cmptun instruction, 4–13, 4–14
cmpule instruction, 3–17
cmpult instruction, 3–17
code optimization, 6–1
.comm directive, 5–4
comments, 2–1
compilation options
  -S option, 6–14
constant
  floating-point, 2–2
  scalar, 2–2
  string, 2–3
control instructions
  floating-point instruction set, 4–15
  main instruction set, 3–18
counters, 6–4
cpys instruction, 4–14, 4–15
cpyse instruction, 4–14, 4–15
cpysn instruction, 4–14, 4–15
ctlz instruction, 3–24, 3–26
ctpop instruction, 3–25, 3–26
cttz instruction, 3–25, 3–26
cvtdg instruction, 4–11, 4–13
cvtgd instruction, 4–11, 4–13
cvtgf instruction, 4–11, 4–13
cvtgq instruction, 4–11, 4–12
cvtlq instruction, 4–11, 4–12
cvtqf instruction, 4–11, 4–13
cvtqg instruction, 4–11, 4–13
cvtql instruction, 4–11, 4–12
cvtqs instruction, 4–11, 4–13
cvtqt instruction, 4–11, 4–13
cvtst instruction, 4–11, 4–13
cvttq instruction, 4–11, 4–12
cvtts instruction, 4–11, 4–13

D

.d_floating directive, 5–4
.data directive, 5–4
data type
  types supported, 2–10
directive
  assembler directives, 5–1
divf instruction, 4–11, 4–12
divg instruction, 4–11, 4–12
divl instruction, 3–9, 3–12
divlu instruction, 3–9, 3–12
divq instruction, 3–9, 3–13
divqu instruction, 3–9, 3–13
divs instruction, 4–11, 4–12
divt instruction, 4–11, 4–12
.double directive, 5–4
dynamic rounding mode, 4–3

E

.edata directive, 5–5
.eflag directive, 5–5
.end directive, 5–5
.endr directive, 5–5
.ent directive, 5–5
eqv instruction, 3–15, 3–16
.err directive, 5–5
escape character, backslash, 2–3
excb instruction, 3–24
exception
  floating-point, 1–5
  main processor, 1–5
expression operator, 2–8
expressions
  operator precedence rules, 2–9
  type propagation rules, 2–11
extbl instruction, 3–21, 3–22
.extended directive, 5–6
.extern directive, 5–6
extlh instruction, 3–21, 3–22

extll instruction, 3–21, 3–22
extqh instruction, 3–21, 3–22
extql instruction, 3–21, 3–22
extwh instruction, 3–21, 3–22
extwl instruction, 3–21, 3–22

F

.f_floating directive, 5–6
fabs instruction, 4–10, 4–12
fbeq instruction, 4–16
fbge instruction, 4–15, 4–16
fbgt instruction, 4–15, 4–16
fble instruction, 4–15, 4–16
fblt instruction, 4–15, 4–16
fbne instruction, 4–15, 4–16
fclr instruction, 4–10, 4–12
fcmoveq instruction, 4–14, 4–15
fcmovge instruction, 4–14, 4–15
fcmovgt instruction, 4–14, 4–15
fcmovle instruction, 4–14, 4–15
fcmovlt instruction, 4–14, 4–15
fcmovne instruction, 4–14, 4–15
fetch instruction, 3–24, 3–25
fetch_m instruction, 3–24, 3–25
.file directive, 5–6
.float directive, 5–6
floating-point constant, 2–2
floating-point control register
(*See* FPCR)
floating-point directives
.d_floating (VAX D_floating), 5–4
.f_floating (VAX F_floating), 5–6
.g_floating (VAX G_floating), 5–7
.s_floating (IEEE single precision), 5–12
.t_floating (IEE double precision), 5–13
.x_floating (IEE quad precision), 5–14
floating-point exception traps, 4–4
floating-point instruction qualifiers
rounding mode qualifiers, 4–7
trapping mode qualifiers, 4–7
floating-point instruction set, 4–1
floating-point instructions
arithmetic instructions, 4–10
control instructions, 4–15
load instructions, 4–9
move instructions, 4–14
relational instructions, 4–13
special-purpose instructions, 4–16
store instructions, 4–9
floating-point rounding modes, 4–5
.fmask directive, 5–6
fmov instruction, 4–14, 4–15
fneg instruction, 4–10, 4–12
fnop instruction, 4–16
FPCR, 4–3
.frame directive, 5–7

G

.g_floating directive, 5–7
global offset table
(*See* GOT)
.globl directive, 5–7
GOT, 6–2
.gprel32 directive, 5–7
.ident directive, 5–8

I

identifier
syntax, 2–1
immediate values, C–3
implicit register use, C–1
implver instruction, 3–24, 3–25
infinity

rounding toward plus or minus infinity, 4–6
insbl instruction, 3–21, 3–22
inslh instruction, 3–21, 3–23
insll instruction, 3–21, 3–23
insqh instruction, 3–21, 3–23
insql instruction, 3–21, 3–23
instruction qualifiers, floating-point
rounding mode qualifiers, 4–7
trapping mode qualifiers, 4–7
instruction set summaries, A–1
inswh instruction, 3–21, 3–23
inswl instruction, 3–21, 3–23
integer arithmetic instructions, C–4

J

jmp instruction, 3–19, 3–20
jsr instruction, 3–19, 3–20
jsr_coroutine instruction, 3–19, 3–20

K

keyword statement, 2–5

L

.lab directive, 5–8
label definition, 2–5
language interfaces, 6–2
.lcomm directive, 5–8, 6–4
lda instruction, 3–2, 3–4
ldah instruction, 3–3, 3–6
ldb instruction, 3–2, 3–4
ldbu instruction, 3–2, 3–4
ldf instruction, 4–9, 4–10
ldg instruction, 4–9, 4–10
ldgp instruction, 3–3, 3–6
ldid instruction, 4–9, 4–10
ldif instruction, 4–9, 4–10
ldig instruction, 4–9, 4–10
ldil instruction, 3–3, 3–6
ldiq instruction, 3–3, 3–6
ldis instruction, 4–9, 4–10
ldit instruction, 4–9, 4–10
ldl instruction, 3–2, 3–4t
ldl_l instruction, 3–2, 3–5
ldq instruction, 3–2, 3–5
ldq_l instruction, 3–2, 3–5
ldq_u instruction, 3–2, 3–5
lds instruction, 4–9, 4–10
ldt instruction, 4–9, 4–10
ldw instruction, 3–2, 3–4
ldwu instruction, 3–2, 3–4
linkage conventions
examples, 6–10
general, 6–3
language interfaces, 6–14
memory allocation, 6–17
.lit4 directive, 5–8
.lit8 directive, 5–9
.lita section, 6–5
little-endian
byte ordering, 1–2
load and store instructions, C–3
load instructions
floating-point instruction set, 4–9
main instruction set, 3–2
.loc directive, 5–9
logical instructions
descriptions of, 3–15
formats, 3–14
.long directive, 5–9

M

.mask directive, 5–9
mb instruction, 3–24, 3–25
mf_fpcr instruction, 4–16
minus infinity
rounding toward (IEEE), 4–6
mnemonic
definition, 2–5
mov instruction, 3–18

move instructions
floating-point instruction set, 4–14
main instruction set, 3–17
mskbl instruction, 3–21, 3–23
msklh instruction, 3–21, 3–23
mskll instruction, 3–21, 3–23
mskqh instruction, 3–21, 3–23
mskql instruction, 3–21, 3–23
mskwh instruction, 3–21, 3–23
mskwl instruction, 3–21, 3–23
mt_fpcr instruction, 4–16
mulf instruction, 4–11, 4–12
mulg instruction, 4–11, 4–12
mull instruction, 3–9, 3–11
mullv instruction, 3–9, 3–11
mulq instruction, 3–9, 3–11
mulqv instruction, 3–9, 3–11
muls instruction, 4–11, 4–12
mult instruction, 4–11, 4–12

N

negf instruction, 4–10, 4–12
negg instruction, 4–10, 4–12
negl instruction, 3–8, 3–10
neglv instruction, 3–8, 3–10
negq instruction, 3–8, 3–10
negqv instruction, 3–8, 3–10
negs instruction, 4–10, 4–12
negt instruction, 4–10, 4–12
nop instruction, 3–24, 3–25
normal rounding (IEEE)
unbiased round to nearest, 4–6
normal rounding (VAX)
biased, 4–5
not instruction, 3–14, 3–15
null statement, 2–5

O

operator evaluation order
precedence rules, 2–9
operator, expression, 2–8
optimization
optimizing assembly code, 6–1
.option directive, 5–9
or instruction, 3–15
ornot instruction, 3–15, 3–16

P

PALcode
instruction summaries, D–1
performance
optimizing assembly code, 6–1
plus infinity
rounding toward (IEEE), 4–6
precedence rules
operator evaluation order, 2–9
program model
memory layout, 6–2
program optimization, 6–1
.prologue directive, 5–10
pseudoinstructions, C–4

Q

.quad directive, 5–10

R

.rconst directive, 5–10
.rdata directive, 5–10
register use, 6–3
registers
floating-point, 1–2, 6–4t
general, 1–1
integer, 1–1, 6–3
relational instructions
floating-point instruction set, 4–13
main instruction set, 3–16
relocation operand
syntax and use, 2–6

reml instruction, 3–9, 3–13
remlu instruction, 3–9, 3–13
remq instruction, 3–9, 3–14
remqu instruction, 3–9, 3–14
.repeat directive, 5–10
ret instruction, 3–19, 3–20
rounding mode
  chopped rounding (IEEE), 4–6
  chopped rounding (VAX), 4–6
  dynamic rounding qualifier, 4–3
  floating-point instruction qualifiers, 4–7
  floating-point rounding modes, 4–5
  FPCR control, 4–3
  normal rounding (IEEE, unbiased), 4–6
  normal rounding (VAX, biased), 4–5
  rounding toward minus infinity (IEEE), 4–6
  rounding toward plus infinity (IEEE), 4–6
rpcc instruction, 3–24, 3–25

S

-S compilation option, 6–14
.s files, 6–14
.s_floating directive, 5–12
s4addl instruction, 3–9, 3–11
s4addq instruction, 3–9, 3–11
s4subl instruction, 3–9, 3–12
s4subq instruction, 3–9, 3–12
s8addl instruction, 3–9, 3–11
s8addq instruction, 3–9, 3–11
s8subl instruction, 3–9, 3–12
s8subq instruction, 3–9, 3–12
.save_ra directive, 5–11
.sbss section, 6–4
scalar constant, 2–2
.sdata directive, 5–11
.set directive, 5–11
sextb instruction, 3–8, 3–10
sextl instruction, 3–8, 3–10
sextw instruction, 3–8, 3–10
shift instructions
  descriptions of, 3–15
  formats, 3–14
sll instruction, 3–15, 3–16
.space directive, 5–12
special-purpose instructions
  floating-point instruction set, 4–16
  main instruction set, 3–24
sra instruction, 3–15, 3–16
srl instruction, 3–15, 3–16
stack organization, 6–7
statement, 2–5
stb instruction, 3–3, 3–6
stf instruction, 4–9, 4–10t
stg instruction, 4–9, 4–10t
stl instruction, 3–3, 3–7
stl_c instruction, 3–3, 3–7
store instructions
  floating-point instruction set, 4–9
  main instruction set, 3–2
stq instruction, 3–3, 3–7
stq_c instruction, 3–3, 3–7
stq_u instruction, 3–3, 3–7
string constant, 2–3
.struct directive, 5–12
sts instruction, 4–9, 4–10t
stt instruction, 4–9, 4–10t
stw instruction, 3–3, 3–7
subf instruction, 4–11, 4–12
subg instruction, 4–11, 4–12
subl instruction, 3–9, 3–11
sublv instruction, 3–9, 3–11
subq instruction, 3–9, 3–12
subqv instruction, 3–9, 3–12
subs instruction, 4–11, 4–12
subt instruction, 4–11, 4–12
symbolic equate, 5–12

T

.t_floating directive, 5–13
.text directive, 5–13

thread local storage
(*See* .tls* directives)
.tlscomm directive, 5–13
.tlsdata directive, 5–13
.tlslcomm directive, 5–13
trapb instruction, 3–24, 3–25
trapping mode
floating-point instruction qualifiers, 4–7
.tune directive, 5–14
type propagation rules
in expressions, 2–11

U

uldl instruction, 3–3, 3–6
uldq instruction, 3–3, 3–6
uldw instruction, 3–3, 3–5
uldwu instruction, 3–3, 3–5
umulh instruction, 3–9, 3–12
unop instruction, 3–24, 3–25
ustl instruction, 3–3, 3–7
ustq instruction, 3–3, 3–7
ustw instruction, 3–3, 3–7

V

.verstamp directive, 5–14

W

.weakext directive, 5–14
wmb instruction, 3–24, 3–26
.word directive, 5–14

X

.x_floating directive, 5–14
xor instruction, 3–15
xornot instruction, 3–15, 3–16

Z

zap instruction, 3–21, 3–24
zapnot instruction, 3–21, 3–24