Alpha Assembly Language Guide

Randal E. Bryant
Carnegie Mellon University
Randy.Bryant@cs.cmu.edu

September 10, 1998

1. Overview

This document provides an overview of the Alpha instruction set and assembly language programming conventions. The Alpha architecture was formulated by Digital Equipment Corporation as a second generation reduced instruction set computer (RISC) architecture. It represents a careful balance between providing a sufficient range of instructions to encode common operations while avoiding a lot of features that could slow down the machine or lead to implementation difficulties in the future. Digital Equipment Corporation was subsequently acquired by Compaq in the Summer of 1998. Compaq is continuing to support Alpha. More complete documentation is available from Digital/Compaq [1, 2].

1.1. Data Types

The most notable feature of the Alpha is that it is a true 64-bit machine. All integer registers are 64 bits wide. Manipulating 64-bit addresses and 64-bit integers is fully supported. In addition, there is support for 32-bit integers.

For historical reasons, (dating back to the PDP-11, a 16-bit machine), Digital had an idiosyncratic terminology for word sizes. They consider a "word" to be 16-bits. Based on this, they refer to 32-bit

C declaration	Alpha Data Type	Size (Bytes)
`char`	Byte	1
`short`	Word	2
`int`	Long Word	4
`unsigned`	Long Word	4
`long int`	Quad Word	8
`long unsigned`	Quad Word	8
`char *`	Quad Word	8
`float`	S_Floating	4
`double`	T_Floating	8

Table 1: Sizes of standard data types

quantities as “long words” (the word size of the VAX.) They refer to 64-bit quantities as “quad words.” We are mostly interested in long words and quad words.

Table 1 shows the machine representations used for the primitive data types of C. Note that variables declared as `int`’s are stored as long (4-byte) words. If you want an 8-byte number, you need to declare it as `long`. All pointers (shown here as `char *`) are stored as 8-byte quad words. Don’t confuse the two uses of the word “long” here—Alpha long words are 4 bytes, but C `long int`’s are 8.

Within the machine, all integer registers hold quad words. Long words are converted to quad words by sign extension. That is, when converting from a long word lw to a quad word qw, the high order bit of lw is replicated as the most significant 33 bits of qw. For some strange reason, even data declared as `unsigned int` in C is stored in this “sign extended” form, even though the high order bit of an unsigned value is not a sign bit. As a consequence, the code generated to manipulate unsigned values is fairly clumsy.

1.2. Porting C Code to the Alpha

When porting C code originally developed on a 32-bit machine to an Alpha, the difference between the sizes for pointers and `int`’s is a common source of non-portability. Lots of code has been written assuming that you could store pointers in locations declared as `int`’s with no loss of information.

Another source of problems is with integer constants. By default, constants in C are assumed to be `int`’s. If you want to make them long, you need to add the suffix “`L`”. Without that suffix, the number is truncated to 32 bits and then sign extended to 64. Here are some examples illustrating this effect:

```
long int a = 0xFFFFFFFF7FFFFFFF;  /* 0x000000007FFFFFFFL */
long int b = 0xFFFFFFFF7FFFFFFFL; /* 0xFFFFFFFF7FFFFFFFL */
long int c = 0x0000000080000000;  /* 0xFFFFFFFF80000000L */
long int d = 0x0000000080000000L; /* 0x0000000080000000L */
```

Observe that values `b` and `d` are most likely what the programmer intended them to be. Values `a` and `c`, on the other hand are not, because the “`L`” suffix was omitted. Their high order bits are either all 0’s or all 1’s depending on bit 31 of the declared constant.

As another example, the statement

```
long int t = 1 << 32;
```

sets variable `t` to 0, which is most likely not the desired result. Since the constant `1` is interpreted as an `int`, the shift is performed to put the 1 in bit position 32. But this is converted to a `long int` by copying bit 31, which is 0, into bit positions 32 through 63. Instead, the expression should be written `1L << 32`, to ensure that the expression is evaluated using long integers. In fact, it is best to get into the habit of adding the “`L`” to the end of every constant. Such code will run reliably on both Alpha’s and on 32-bit machines.

When you want to print out 8-byte integers with `printf`, you need to use the directive `%ld`, rather than the standard `%d`. Similarly for printing in hexadecimal (`%lx`) and unsigned (`%lu`) formats.

Long Word	Quad Word	Description	Computation
`addl`	`addq`	Add	`c = a + b`
`s4addl`	`s4addq`	Scaled by 4 Add	`c = 4*a + b`
`s8addl`	`s8addq`	Scaled by 8 Add	`c = 8*a + b`
`subl`	`subq`	Subtract	`c = a - b`
`s4subl`	`s4subq`	Scaled by 4 Subtract	`c = 4*a - b`
`s8subl`	`s8subq`	Scaled by 8 Subtract	`c = 4*a - b`
`mull`	`mulq`	Multiply	`c = a * b`
`divl`	`divq`	Divide	`c = a / b`
`reml`	`remq`	Remainder	`c = a % b`

Table 2: Arithmetic Operations. Each instruction has a (4-byte) word and a quad (8-byte) word form.

2. Instructions

The Alpha instruction set is relatively simple. Arithmetic operations apply only to register data, i.e., both the source and destination operands must be register data. Explicit load and store operations are required to move data between memory and registers. Conditional branches can only test the relation between a register and the value zero.

2.1. Arithmetic Operations

Alpha supports integer operations for both 4-byte and 8-byte integers. The 8-byte versions treat the operands as full precision values. The 4-byte versions mimic the behavior one would obtain by executing the operations on a 32-bit machine. That is, they compute a value based on only the low order 4 bytes of the operands to generate a 4-byte value. They then sign extend this value to obtain the 8-byte result.

Table 2 lists the arithmetic instructions having both 4-byte and 8-byte versions (note the suffixes "`l`" and "`q`".) Arithmetic operations have three operands: two source and one destination. The destination is given as the rightmost operand, [in contrast to many other assembly languages where the destination is given as the leftmost operand.] Arithmetic operations can have one of two formats (shown for instruction `addq`):

$$\begin{array}{llll}\texttt{addq} & \texttt{R}_a, & \texttt{R}_b, & \texttt{R}_c \\ \texttt{addq} & \texttt{R}_a, & Lit_b, & \texttt{R}_c\end{array}$$

where $\texttt{R}_a$, $\texttt{R}_b$, and $\texttt{R}_c$ denote registers, and Lit_b denotes a "literal" constant between 0 and 255. The first two operands denote the operation sources: the first must be from register $\texttt{R}_a$, the second can either be from a register $\texttt{R}_b$ or a literal value Lit_b. The third operand denotes the destination, which must always be a register $\texttt{R}_c$. We will use the notation "`a`" and "`b`" to indicate the two source operands, where `a` is the value of register $\texttt{R}_a$, while `b` is either the value of $\texttt{R}_b$ or of Lit_b. Similarly, we will use the notation "`c`" to indicate the operation result, which will then be written to register $\texttt{R}_c$. Table 2 shows the effect of each of these instructions using C notation, with source operands `a` and `b`, and destination operand `c`.

For operations requiring constants that don't fit within the 8 bit limit of the standard operations, it is common to use instructions `lda` (load address) and `ldah` (load address high). These are documented in Section 2.4 describing load and store operations, even though they do not reference memory. Alternatively,

Instruction	Description	Computation
`cmpeq`	Equality	`c = (a == b)`
`cmple`	Less than or equal	`c = (a <= b)`
`cmplt`	Less than	`c = (a < b)`
`cmpule`	Unsigned less than or equal	`c = (ua <= ub)`
`cmpult`	Unsigned less than	`c = (ua < ub)`

Table 3: Comparison Operations

constants can be declared as part of the assembly program data and stored in memory. Load instructions can then put these values into registers.

The scaled operations, having prefixes "`s4`" and "`s8`" scale the first source value by a factor of 4 or 8. These are commonly used for array indexing.

The following are some examples of typical assembly code using arithmetic operations:

```
# Add $1 and $2 and store in $3
addq $1, $2, $3
# Register $8 points to integer array a
# Register $17 contains index i
# Want to set $9 to &a[i]:
s4addq $17, $8, $9
```

2.2. Comparison Operations

All comparisons operations operate on quad (8-byte) words. They have the same format as arithmetic operations. They result in destination register $\mathtt{R}_c$ being set to 1 (true) or 0 (false). Table 3 lists the different comparison possibilities. Note that the inequality tests have both signed and unsigned versions. These are indicated with C syntax using operands `a` and `b` as signed values and `ua` and `ub` as unsigned values.

2.3. Bit-Level and Logical Operations

All bit-level and shift operations operate on quad words. They have the same format as arithmetic instructions. Table 4 lists the different possibilities.

Left shift inserts 0's into the low order bit positions. Logical right shift inserts 0's into the high order bit positions (used for unsigned operands). Arithmetic right shift copies the high order bit of operand `a` into the new bit positions (used for signed operands).

Note that the shift amount must be between 0 and 63. Any larger number is reduced modulo 64 (by simply masking off all but the low order 6 bits).

Instruction	Description	Effect
`and`	And	`c = a & b`
`bic`	Bit Clear	`c = a & ~b`
`bis`	Bit Set	`c = a | b`
`eqv`	Logical Equivalence	`c = ~(a ^ b)`
`xor`	Exclusive-Or	`c = a ^ b`
`ornot`	Or-Not	`c = a | ~b`
`sra`	Shift Right Arithmetic	`c = a >> (b % 64)`
`sll`	Shift Left	`c = a << (b % 64)`
`srl`	Shift Right Logical	`c = ua >> (b % 64)`

Table 4: Bit-Level Operations

Instruction	Description	Bytes Accessed	Effective Address	Effect
`ldl`	Load Long	4	$EA = b + D$	`a = *EA`
`ldq`	Load Quad	8	$EA = b + D$	`a = *EA`
`stl`	Store Long	4	$EA = b + D$	`*EA = a`
`stq`	Store Quad	8	$EA = b + D$	`*EA = a`
`lda`	Load Address	0	$EA = b + D$	`a = EA`
`ldah`	Load Address High	0	$EA = b + D \cdot 65536$	`a = EA`
`ldq_u`	Load Quad Unaligned	8	$EA = (b + D)$ `& ~0x7`	`a = *EA`
`stq_u`	Store Quad Unaligned	8	$EA = (b + D)$ `& ~0x7`	`*EA = a`

Table 5: Load and Store Operations

2.4. Loads and Stores

Load and store operations are used to transfer data between registers and memory. Separate instructions are used to perform long (4-byte) word accesses and quad (8-byte) word accesses. In addition, instructions `lda` and `ldah` have the format of a load operation, but they do not cause any memory references.

Load and store instructions have the following format, shown with instruction `ldq` (load quad word):

$$\texttt{ldq}\quad \texttt{R}_a,\quad \mathit{Disp}(\texttt{R}_b)$$

Operands $\texttt{R}_a$ and $\texttt{R}_b$ indicate registers, while $Disp$ is a constant *displacement* ranging between $-32,768$ and $+32,767$. In most cases $\texttt{R}_a$ indicates the destination (load) or source (store) of the data, while the combination of $\texttt{R}_b$ and $Disp$ indicates the memory location to access. Note that load instructions are the only Alpha instructions for which the destination is written on the left.

Table 5 describes the load and store operations. The column labeled "Effective Address" shows how the contents of register $\texttt{R}_b$, denoted b, and the value of the displacement $Disp$, denoted D are combined to generate an effective address EA. In most cases the values are simply added. For the `ldah` instruction, the value D is scaled by a factor of $2^{16} = 65,536$.

The column labeled "Effect" in Table 5 describes the behavior of the operation in C notation, where the effective address EA is represented by a pointer variable `EA`, and register $\mathtt{R}_a$ is represented by variable `a`. Load instructions read from the effective address and place the result in register $\mathtt{R}_a$. The quad word version `ldq` reads 8 bytes. The long word version `ldl` reads only 4 bytes and sign extends them to 8. Conversely, the store operations write the value in register $\mathtt{R}_a$ to the memory locations indicated by the effective address. The quad word version `stq` writes all 8 bytes, while the long word version writes only the low order 4 bytes of $\mathtt{R}_a$.

Instructions `lda` and `ldah` place the effective address in register $\mathtt{R}_a$ without accessing any memory locations. They are useful for setting addresses, for performing pointer arithmetic, and even for performing integer operations involving constants.

```
# Set $1 to absolute address 0x000F0FF0
# Use property that $31 is always 0
ldah $1, 15($31)   # 0xF
lda $1, 4080($1)  # 0x0FF0
# Compute p++ for integer pointer p in register $2
lda $2, 4($2)
# Compute x -= 17 for integer x in register $5
lda $5, -17($5)
```

The above load and store operations have an *alignment* restriction, limiting the set of legal effective address values. The general principle is when loading or storing a k-byte object, the effective address EA must be a multiple of k. In practice, this means that Alpha long word loads and stores must have EA be a multiple of 4, i.e., the 2 low-order bits of EA must be 0. Similarly quad word loads and stores must have EA be a multiple of 8, i.e., the 3 low-order bits of EA must be 0. This restriction is imposed to allow the memory system to operate as efficiently as possible. Compilers carefully organize all data structures and allocate storage in such a way that these alignment restrictions are obeyed.

Instructions `ldq_u` and `stq_u` are typically used when performing byte-level operations. They operate like ordinary load and store operations, except that they set the low order three bits of the effective address to 0 (shown in the table using the C trick of AND'ing EA with a mask `~0x7`). Thus if pointer `p`, stored in register `$16`, points to some character position in a string, then the instruction `ldq_u $1,0($16)` will load the quad word containing this character into register `$1`, but its position within the quad word can be anywhere from byte 0 to byte 7. As is discussed in Section 2.6, other instructions are provided to support accessing and operating on such byte data. These instructions, plus the byte-level manipulation instructions, provide a bridge between the hardware, which only supports aligned memory references, and the need of the compiler to generate efficient code for byte-level memory operations.

2.5. Conditional Moves

Conditional move operations provide a means of conditionally updating a register without using any branch operations. In modern machines such as Alpha, this can yield much better performance than the traditional technique of conditionally branching around the updating code.

These instructions have the same format as arithmetic operations, e.g., for instruction `cmoveq`:

`cmoveq`	$\mathtt{R}_a$,	$\mathtt{R}_b$,	$\mathtt{R}_c$
`cmoveq`	$\mathtt{R}_a$,	Lit_b,	$\mathtt{R}_c$

Instruction	Description	Move Condition
`cmoveq`	Conditional Move on Equal	`a == 0`
`cmovne`	Conditional Move on Not Equal	`a != 0`
`cmovgt`	Conditional Move on Greater Than	`a > 0`
`cmovge`	Conditional Move on Greater Than or Equal	`a >= 0`
`cmovlt`	Conditional Move on Less Than	`a < 0`
`cmovle`	Conditional Move on Less Than or Equal	`a <= 0`
`cmovlbc`	Conditional Move on Lower Bit Clear	`!(a & 0x1)`
`cmovlbs`	Conditional Move on Lower Bit Set	`a & 0x1`

Table 6: Conditional Move Instructions

Instruction	Description	`Byte(c, i)`, $0 \leq i \leq 7$
`extbl`	Extract Byte Low	`(i==0) ? Byte(a, b & 0x7) : 0`
`mskbl`	Mask Byte Low	`(b & 0x7 == i) ? 0 : Byte(a, i)`
`insbl`	Insert Byte Low	`(b & 0x7 == i) ? Byte(a, i) : 0`
`zap`	Zero Bytes	`Bit(b, i) ? 0 : Byte(a, i)`
`zapnot`	Zero Bytes Not	`Bit(b, i) ? Byte(a, i) ? 0`

Table 7: Byte Manipulation Instructions

Register $\mathtt{R}_a$ indicates the tested value, either register $\mathtt{R}_b$ or Lib_b indicates the source data, and register $\mathtt{R}_c$ designates the move destination. Whether or not the move takes place is based on the result of comparing register $\mathtt{R}_a$ to 0.

Table 6 lists the different conditional move instructions and the type of comparison performed. C expression syntax is used, where variable `a` denotes the contents of register $\mathtt{R}_a$. For example, the `cmoveq` instruction is equivalent to the following C code:

```
if (a == 0)
   c = b;
```

where variable `b` represents the source data and variable `c` represents the destination.

2.6. Byte Manipulation Operations

The Alpha does not directly support byte-level operations such as transferring single bytes between memory and registers. In principal, we could use the instructions already presented to realize byte-level manipulations, but a large amount of shifting and masking would be required. For example, consider the C operation `*dest = *src`, where both `dest` and `src` are of type `(char *)`. This operation must read the single byte pointed to by `src` and update the single byte pointed to by `dest`. Without special byte-manipulation instructions, this simple operation requires 17 Alpha instructions!

Table 7 documents some of the byte manipulation instructions. Each of these has the same form as an arithmetic instruction, with source operands `a` and `b` (where `b` may be either a register or a literal value), and result `c`. The right hand column indicates how each byte i of the result is computed, where we use the notation `Byte(`x`, `j`)` to denote byte j of quad word x. Our machines are configured to operate in "little endian" mode, with bytes numbered from 0 (least significant) to 7 (most significant).

Instructions `extbl`, `mskbl`, and `insbl` are designed to work in conjunction with instructions `ldq_u` and `stq_u` discussed in Section 2.4. Recall that these load and store instructions perform quad-word aligned loads and stores by ignoring the low order 3 bits of the effective address. These 3 byte manipulation instructions work in conjunction with these loads and stores to select and manipulate the byte within a quad word indicated by the low order 3 bits of operand `b`, as is indicated by the C expression `b & 0x7`. Instruction `extbl` selects the indicated byte, moves it to the low order byte of `c`, and sets the remaining bytes to 0. Instruction `mskbl` masks the indicated byte from source `a` while copying the rest. Instruction `insbl` shifts the low order byte of `a` into the indicated byte position while setting the remaining bytes of `c` to 0.

Thus, the C operation `*dest = *src` can be expressed by a 7 instruction sequence broken down as follows. We assume that operand `dest` is in register `$16`, while `src` is in register `$17`. We begin by getting the byte indicated by `src` into the low order byte of register `$1`:

```
ldq_u $1, 0($17)    # Get quad word containing *src
extbl $1, $17, $1   # Move *src into byte 0
```

Next, we load the quad word containing `*dest` into register `$2`:

```
ldq_u $2, 0($16)    # Get quad word containing *dest
```

We must replace the appropriate byte within this quad word with the low order byte of register `$1`. First, we move the new byte into the proper position:

```
insbl $1, $16, $1   # Move *src to byte position for *dest
```

Then we clear the byte in the destination quad word:

```
mskbl $2, $16, $2   # Clear byte position of *dest
```

We now merge these two with an `Or` operation:

```
bis   $1, $2,  $2   # Merge *src into quad word containing *dest
```

Finally, we store the updated quad word:

```
stq_u $2, 0($16)    # Store updated quad word containing *dest
```

Although this process seems laborious, it is superior to the 17 instruction sequence required without byte-level operations.

The final pair of instructions in Table 7 allow selective zeroing of bytes using a bit mask given by the low order 8 bits of operand `b`. Numbering these bits from 0 (least significant) to 7 (most significant), each bit indicates whether to copy the corresponding byte from operand `a` to the destination or to set this destination byte to 0. The two instructions: `zap` and `zapnot` differ only in the sense of the bit interpretations.

Form	Description	Actual Implementation
`nop`	No operation	`bis $31, $31, $31`
–	Alternate no operation	`ldq_u $31, 0($sp)`
`mov $1, $2`	Move register	`bis $31, $1, $2`
`mov 17, $2`	Move literal	`bis $31, 17, $2`
`sextl $1, $2`	Move long word and sign-extend	`addl $31, $1, $2`

Table 8: Special Case Operations

Instruction	Description	Branch Condition
`beq`	Branch on Equal	`a == 0`
`bne`	Branch on Not Equal	`a != 0`
`bgt`	Branch on Greater Than	`a > 0`
`bge`	Branch on Greater Than or Equal	`a >= 0`
`blt`	Branch on Less Than	`a < 0`
`ble`	Branch on Less Than or Equal	`a <= 0`
`blbc`	Branch on Lower Bit Clear	`!(a & 0x1)`
`blbs`	Branch on Lower Bit Set	`a & 0x1`
`br`	Branch	`1`
`bsr`	Branch to Subroutine	`1`

Table 9: Branch Instructions

2.7. Useful Special Case Operations

In an attempt to make assembly language more readable, some commonly used patterns are given special names. These typically involve degenerate cases, such as copying from one register to another. They exploit the fact that integer register `$31` is always 0. Table 8 lists some typical cases and their translations into actual Alpha instructions. No-op instructions `nop` are commonly used to pad code to meet specified alignment requirements. The instruction `bis $31, $31, $31` has no effect, since the destination register `$31` never changes. Sometimes the compiler generates a second form of no-op shown in the second row of the table. It performs an unaligned load using the stack pointer `$sp` as the base register, but it has no effect since the destination register is `$31`. Move instructions `mov` are used to transfer from one register to another, or to set a register to a constant value. The sign extension operation `sextl` is the common method for converting from C `int`'s to `long int`'s. For example, it is common to see instructions of the form `sextl $16, $16` to convert an integer argument passed in register `$16` into the numerically equivalent quad word.

Note that the native C compiler CC makes use of these special names. The GCC compiler, on the other hand, generates the "implementation form" shown in the right hand column of Table 8, as does the disassembler `dis`. Understanding these special cases is important to being able to read the code generated by these programs.

2.8. Transfers of Control

Transfers of control come in two flavors: branches and jumps. Most branches are conditional—whether

S_floating	T_floating	Description	Computation
`adds`	`addt`	Add	`c = a + b`
`subs`	`subt`	Subtract	`c = a - b`
`muls`	`mult`	Multiply	`c = a * b`
`divs`	`divt`	Divide	`c = a / b`

Table 10: Floating Point Arithmetic Operations. Each instruction has a single precision (S_floating) and a double precision (T_floating) version.

or not they are taken depends on the result of comparing an operand register to 0. They have format (shown for `beq`):

$$\texttt{beq}\quad \texttt{R}_a\texttt{,}\ \mathit{Label}$$

where $\texttt{R}_a$ denotes the register being tested and *Label* is a label designating some position in the assembly code. The assembler automatically translates this label into an offset relative to the program counter. The upper part of Table 9 documents the different conditional branch types and the condition under which they are taken. C syntax is used with variable `a` denoting the contents of register $\texttt{R}_a$.

As shown in the lower part of Table 9, two special branch forms: `br` and `bsr` branch unconditionally. The unconditional branch `br` typically has register `$31` as operand $\texttt{R}_a$. The label then designates the branch target. The branch to subroutine instruction has the same format as other branches, but register argument $\texttt{R}_a$ is used in a totally different way. It designates where the current value of the program counter should be stored to allow the subroutine to return to the calling point. The convention is to use register `$26` for this purpose.

Jump instructions provide unconditional transfers of control with the target address specified by a register. We will use three different forms:

$$\begin{array}{llll} \texttt{jmp} & \texttt{\$31,} & (\texttt{R}_b)\texttt{,} & \mathit{Hint} \\ \texttt{jsr} & \texttt{R}_a\texttt{,} & (\texttt{R}_b)\texttt{,} & \mathit{Hint} \\ \texttt{ret} & \texttt{\$31,} & (\texttt{R}_b)\texttt{,} & \mathit{Hint} \end{array}$$

In all cases *Hint* is optional information inserted by the compiler to help the processor predict the jump target. The exact nature of these hints is not our concern.

The unconditional jump instruction `jmp` designates the jump target address in register $\texttt{R}_b$. The jump to subroutine instruction `jsr` gives the target in register $\texttt{R}_b$ and the register to store the current program counter as argument $\texttt{R}_a$. The convention is to use register `$26` for this purpose. The return instruction `ret` is functionally equivalent to a jump—it gives the target address as register $\texttt{R}_b$. By conventional this will be register `$26`, holding to program counter set by the preceding `bsr` or `jsr`. Both the `jmp` and the `ret` operations have an operand $\texttt{R}_a$, but this is typically register `$31`.

2.9. Floating Point

Floating point instructions use a set of 32 floating point registers, named `$f0` to `$f31`. Four floating point formats are supported, but we are only interested in two: the S_floating format implementing IEEE

Instruction	Description	Computation
`cmpteq`	Equality	`c = (a == b) ? 2.0 : 0.0`
`cmptle`	Less than or equal	`c = (a <= b) ? 2.0 : 0.0`
`cmptlt`	Less than	`c = (a < b) ? 2.0 : 0.0`

Table 11: Floating Point Comparison Operations

Instruction	Description	Bytes Accessed	Effective Address	Effect
`lds`	Load S_floating	4	$EA = b + D$	`a = *EA`
`ldt`	Load T_floating	8	$EA = b + D$	`a = *EA`
`sts`	Store S_floating	4	$EA = b + D$	`*EA = a`
`stt`	Store T_floating	8	$EA = b + D$	`*EA = a`

Table 12: Floating Point Load and Store Operations

single precision and the T_floating format implementing IEEE double precision. Each floating point register is 8 bytes, but it can hold either a single precision or a double precision value.

In general, the floating point operations mirror the behavior of a subset of the integer operations. For example, floating point arithmetic operations have just one format (shown for instruction `addt`):

$$\texttt{addt}\quad \texttt{F}_a,\quad \texttt{F}_b,\quad \texttt{F}_c$$

where $\texttt{F}_a$ and $\texttt{F}_b$ indicate the two source registers, and $\texttt{F}_c$ indicates the destination register. Table 10 lists the common arithmetic operations, using C variables `a` and `b` to denote the source operands and `c` to denote the destination.

Table 11 lists some of the floating point comparison operations. These set the destination register $\texttt{F}_c$ to 2.0 if the comparison holds and to 0.0 if it does not.

Floating point load and store instructions have the same format as their integer counterparts, except that they use floating point registers for data. For instruction `lds`, the format is:

$$\texttt{lds}\quad \texttt{F}_a,\quad \mathit{Disp}(\texttt{R}_b)$$

Operands $\texttt{F}_a$ and $\texttt{R}_b$ indicate registers, while Disp is a constant *displacement* ranging between -32,768 and +32,767. Floating point register $\texttt{F}_a$ indicates the destination (load) or source (store) of the data, while the combination of $\texttt{R}_b$ and Disp indicates the memory location to access. Table 12 lists the different instructions, their effective address calculations (matching the calculations for integer loads and stores), and the instruction effect. These load and store operations must have aligned addresses, i.e., single precision loads and stores must have EA be a multiple of 4, while double precision loads and stores must have EA be a multiple of 8.

As indicated in Table 13, there are also conditional moves for floating point values. These have the same format as arithmetic operations (shown here for `fcmoveq`):

Instruction	Description	Move Condition
`fcmoveq`	Conditional Move on Equal	`a == 0.0`
`fcmovne`	Conditional Move on Not Equal	`a != 0.0`
`fcmovgt`	Conditional Move on Greater Than	`a > 0.0`
`fcmovge`	Conditional Move on Greater Than or Equal	`a >= 0.0`
`fcmovlt`	Conditional Move on Less Than	`a < 0.0`
`fcmovle`	Conditional Move on Less Than or Equal	`a <= 0.0`

Table 13: Conditional Move Instructions

Instruction	Description	Branch Condition
`fbeq`	Branch on Equal	`a == 0.0`
`fbne`	Branch on Not Equal	`a != 0.0`
`fbgt`	Branch on Greater Than	`a > 0.0`
`fbge`	Branch on Greater Than or Equal	`a >= 0.0`
`fblt`	Branch on Less Than	`a < 0.0`
`fble`	Branch on Less Than or Equal	`a <= 0.0`

Table 14: Floating Point Branch Instructions

$$\texttt{fcmoveq}\quad \texttt{F}_a\texttt{,}\quad \texttt{F}_b\texttt{,}\quad \texttt{F}_c$$

The value in register $\texttt{F}_b$ is conditionally copied to register $\texttt{F}_c$ based on the result of comparing $\texttt{F}_a$ to 0.0.

Table 14 describes the conditional branch instructions for floating point. They have format similar to the integer branch instructions (shown for `fbeq`):

$$\texttt{fbeq}\quad \texttt{F}_a\texttt{,}\ \mathit{Label}$$

The decision of whether or not to branch is based on the result of comparing register $\texttt{F}_a$ to 0.0.

Finally, as Table 15 indicates, there is a set of operations for converting between the different numeric formats. Each of these has the same format (shown here for `cvtqs`):

Instruction	From	To
`cvtqs`	Quad integer	S_floating
`cvtqt`	Quad integer	T_floating
`cvtsq`	S_floating	Quad integer
`cvttq`	T_floating	Quad integer
`cvtts`	T_floating	S_floating
`cvtst`	S_floating	T_floating

Table 15: Floating Point Conversion Operations

Register Name	Software Name	Use
`$0`	`v0`	Returned value from integer functions
`$1`–`$8`	`t0`–`t7`	Temporaries
`$9`–`$14`	`s0`–`s5`	Callee saved
`$15`	`s6`	Callee saved
`$15` or `$fp`	`fp`	Frame pointer
`$16`–`$21`	`a0`–`a5`	Integer arguments
`$22`–`$25`	`t8`–`t11`	Temporaries
`$26`	`ra`	Return address
`$27`	`pv`	Address of current procedure
`$27`	`t12`	Temporary
`$28` or `$at`	`AT`	Reserved for assembler
`$29` or `$gp`	`gp`	Global pointer
`$30` or `$sp`	`sp`	Stack pointer
`$31`	`zero`	Always 0

Table 16: Integer Register Usage Conventions

`cvtqs` $\mathtt{F}_b$, $\mathtt{F}_c$

where $\mathtt{F}_b$ indicates the source register, and $\mathtt{F}_c$ indicates the destination register. Surprisingly, a floating point register is used even when the source or destination is a quad word integer. The 8 bytes of the floating point register are simply interpreted as a two's complement number in these cases. In fact, the only way to transfer data from a floating point register to an integer register is to store the floating point register to memory and then load the value from memory into an integer register.

Even in programs that do not use floating point operations, one commonly sees the generated machine code using floating point registers as destinations of loads and sources of stores. In these cases, the registers are just being used as locations capable of holding 4- or 8-byte values. The compiler is evidentally trying to get better performance by using both the integer and the floating point register sets.

3. Programming Conventions

The Alpha hardware provides only low-level support for handling tasks such as setting up procedure calls, maintaining the calling stack, and allocating space for data structures. Built on top of this low-level support is a set of conventions that all compiler writers and assembly code generators are supposed to follow. Having uniform conventions makes it possible to link together code generated from different sources, e.g., to have your C code be able to use routines from the standard Unix libraries. In addition, it enables tools such as debuggers, profilers, and performance monitors to work on code independent of how it was generated.

In this section we summarize some of the key features of the Alpha programming conventions. More extensive documentation can be found in [1].

3.1. Register Usage

Alpha has 32 integer registers, identified as `$0` up to `$31`. As far as the hardware goes, only one of these is special—register `$31` is always equal to 0. Even if it is the destination of an operation, its value never changes. Thus, register `$31` is often used as a "sink" when the value of some operation is not required, such as for the `nop` operation of Table 8.

The remaining integer registers are partitioned into different groups with different uses, as shown in Table 16. Registers can be identified in several ways: by their numbers `$0`–`$31`, by special names `$fp`, `$at`, `$gp`, `$sp`, or by "software" names that are supposed to more clearly identify their usage conventions. These software names are actually just macro definitions from file `regdef.h`. We will generally refer to registers by number, since this is seen in the `.s` files generated by the C compiler. The disassembler program `dis` generates the software names by default but can be made to generate numeric identifiers using the command line flag `-h`.

Observe that some lines in the table refer to the same register, indicating overlapping usages. For example, register `$27` generally holds the starting address of the currently executing procedure, but it can also be used just for temporary storage.

Register `$0` is used to return an integer (or pointer) value to the calling procedure. Registers `$1` to `$8` and `$22` to `$25` can be used by a procedure for arbitrary temporary values. However, if procedure A calls procedure B, there is no guarantee that the values in these registers will be unchanged when B returns back to A. Registers `$9` to `$14` are "callee saved" registers. That means that any procedure using them must first save the old values on the stack and then restore them before it returns. Thus, if A calls B, it can be assured that these register values are unchanged when B returns back to A, either because B did not alter them, or because B saved, altered, and later restored them.

Register `$15` can be used as a "frame pointer," indicating the start of the current stack frame. Most of the time this is not done, however—all stack addressing is done relative to the stack pointer.

Registers `$16` to `$21` are used to pass integer (or pointer) arguments to a procedure. If needed, more arguments can be passed on the program stack.

Register `$26` generally holds the address to which the currently-executing procedure should return. Register `$27` generally points to the currently executing procedure. The typical way for procedure A to call procedure B is to load the starting address of B into register `$27` and then execute the instruction:

```
    jsr $26, ($27)
```

Procedure B then returns to A by executing the instruction

```
    ret ($26)
```

Register `$29` is used as a "global pointer," indicating a region in memory where global data and linkage information is maintained.

Register `$30` is used as the "stack pointer," indicating the address of the top element of the stack. The stack grows toward lower addresses.

Table 17 indicates the usage conventions for the floating point registers. Like the integer registers, only floating point register `$f31` has any special hardware-implemented features—it is always equal to

Register Name	Use
`$f0`	Returned value from floating point functions
`$f1`	Returned imaginary value from complex functions
`$f2`–`$f9`	Callee saved
`$f10`–`$f15`	Temporaries
`$f16`–`$f21`	Floating point arguments
`$f22`–`$f30`	Temporaries
`$f31`	Always 0.0

Table 17: Floating Point Register Usage Conventions

0.0. The remainder are partitioned by convention into return values (`$f0`–`$f1`), callee saved (`$f2`–`$f9`), temporaries (`$f10`–`$f15` and `$f22`–`$f30`), and procedure arguments (`$f16`–`$f21`).

3.2. Stack Frames

The program stack is used as the working storage for procedures. Each procedure requiring local storage to hold local data or linkage information allocates a stack frame upon entry and deallocates it before returning.

Not all procedures require stack space. As long as a procedure does not call any other procedures and can fit all the data it requires in registers, it need not allocate a frame.

Figure 1 shows the general form of a stack frame. Note that the stack grows toward lower addresses, so the top of the stack is actually the lowest address. We will draw the frame with the "top" of the stack on the bottom of the figure. The figure illustrates the most general stack frame form—not all procedures require all parts.

We will refer to the parts of the frame relative to the currently active procedure, i.e., the one who's frame is at the top of the stack. If this procedure had more arguments than could be passed in the integer or floating point registers (e.g., if it had more than 6 integer or pointer arguments), the remaining arguments would be on the stack as part of the caller's frame. The "frame pointer" indicates the top of the caller's frame. Typically, this frame pointer is "virtual", meaning that it's value is defined in terms of some offset relative to the stack pointer. The first portion of the frame holds any local or temporary data that cannot be held in registers. This will typically include local arrays and any local variable to which a pointer must be generated. The next part of the frame provides storage for any registers that the procedure needs to save. Typically this includes the return address pointer (stored first) and the old values of any callee save registers to be used by the current procedure. Finally, the top part of the frame consists of temporary space to be used in building the arguments to procedures to be called by the current procedure. This area is required only if the current procedure calls another procedure with more arguments than can be passed through registers.

A final requirement is that the frame size must be a multiple of 16 bytes. This requirement is satisfied by padding the region for locals and temporaries as needed.

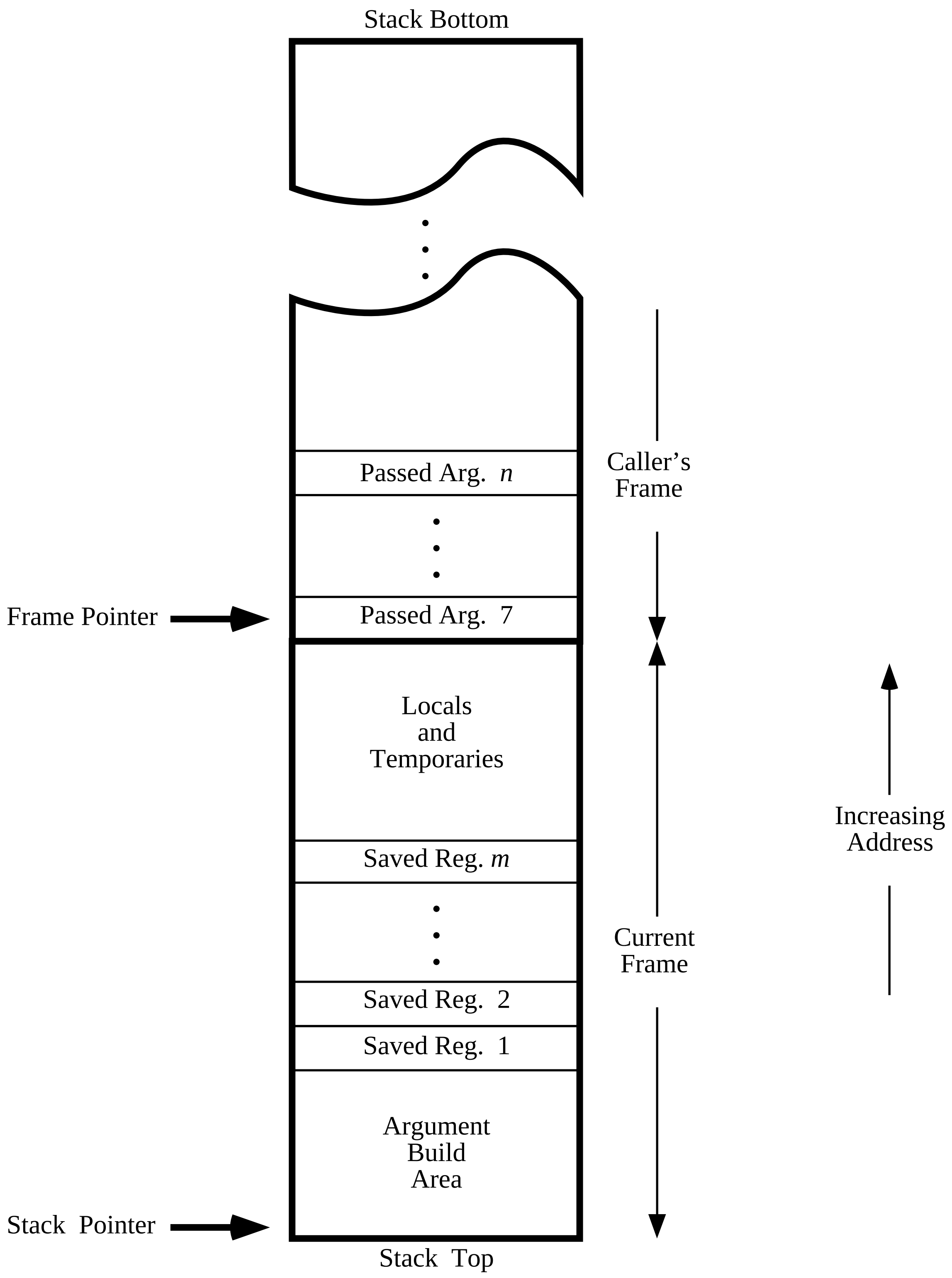

Figure 1: Stack Frame Structure

References

[1] *Alpha Assembly Programmer's Guide*, Digital Equipment Corporation, 1996. Available electronically as `alpha-asm.pdf` (Adobe Acrobat) or `alpha-asm.ps` (Postscript). These documents are stored in the directory:

```
/afs/cs.cmu.edu/academic/class/15213-f98/doc/
```

[2] R. L. Sites, R. T. Witek, *Alpha AXP Architecture*, 3rd edition, Digital Press, 1996. Available electronically as `alpha-ref.pdf` (Adobe Acrobat). These documents are stored in the directory:

```
/afs/cs.cmu.edu/academic/class/15213-f98/doc/
```